Vrije Universiteit Brussel – Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes – France
and
Monash University – Australia
2001

Dynamic Service Discovery across Technology Boundaries

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: David Würth

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promoters: Christine Mingins (Monash University)

Abstract

The Web Service model enables the creation of services which can be exposed to the Internet. By using only open standards as underlying technologies, this model promises a new way of building service-oriented applications. The Web Service model promises seamless integration for heterogeneous platforms and languages, hence providing interworking across technology boundaries. This model is based on XML and Internet-based protocols. A survey of the state of the art of the Web Service model is provided.

The Jini model provides also a service-oriented approach, which is, from an architectural and conceptual point of view, similar to the Web Service model. But being based on the object-oriented paradigm and on the Java language environment, makes the Jini approach very different.

In this thesis we shall introduce and implement a Web Service-to-Jini bridge. The implementation of the bridge is used to prove the promises made by the Web Service model. In addition, the use in conjunction with another service-based approach, namely Jini, enables a comparison of two models, sharing conceptual and architectural similarities. Though, being based on completely different paradigms and technologies.

Table of Contents

Introduction

The thesis describes an approach to a Web Services and Jini interworking approach. This approach provides transparent interworking between applications exposed as Web Services and Jini services. This is provided by building a software bridge, which enables Web Services to be exposed as Jini services. The Web Service paradigm with its underlying technologies, promises seamless integration for heterogeneous platforms and languages, hence providing interworking across technology boundaries. The purpose of the bridge is to use Web Service technology as base, describing the emerging issues and problems. This evaluation of Web Services, a relative new development approach, will test the maturity of this approach. In addition, the use in conjunction with another service-based approach, namely Jini, enables a comparison of two models, sharing conceptual and architectural similarities. Though, being based on completely different paradigms and technologies.

Web Services are a new component-based paradigm. It comprise a conceptual and architectural foundation. Major software companies like Microsoft, IBM, and SUN, are providing application frameworks and their product palettes with Web Service support. The underlying technologies adhere all to specifications and open standards. The competition will be who is providing the better tools and development environments for Web Service development. We take a look at what development frameworks and tools are available today and to what extent they keep to the specifications. Quality of implementation and ease of use are important points to be considered. Further, another interesting question emerging here is, to what extent Web Services will be portable amongst different development platforms. Because the Web Service model is a very recent model, most of the specifications are still work in progress.

Web Services describe a distributed software architecture of service components. They can be integrated a run time, thus producing dynamic and flexible applications. Web Services can be expressed and linked using XML. The messaging passing protocol is based on XML. The use of XML entails platform neutrality. The component-based approach is based on the fact that Web Services are language independent. The use of XML messaging and dynamic discovery facilities enhance the current-day component model, by promoting a very loosely coupled model and the runtime integration of components. Whereas, nowadays component models are language dependent and runtime dependent. This service-based paradigm seems to be meant to supersede the object-based paradigm. The question arising here is if Web Service are only a concern of commercial nature or if they really leverage the existing models.

The building blocks which are enabling the concept of Web Services, are SOAP, WSDL, and UDDI. These three technologies are closely linked to XML and HTTP. The current problem involves the integration and coordination between the Web Services, above all concerning Web Services built within different vendor platforms. One major issue in this thesis is to evaluate the interoperability of different implementations of Web Service technologies. The approach taken is to use the aforementioned implementation of a bridge, which enables Jini services to access dynamically Web Services, implemented and exposed by the .NET application framework. This environment provides all the issues emerging concerning the integration across network and technology boundaries.

The implementation of distributed software is changing from a object-oriented approach to a documented-oriented approach using HTTP and XML. One of the emerging standards, SOAP, replaces the remote procedure call technology by its own XML based messaging model. The SOAP protocol is supposed to a lightweight protocol to support the exchange of information in a distributed environment. Because it need to fit the needs of all involved platforms and clients, it must be completely general. This prevents SOAP to be specialized for certain kinds of communication requirement. Is the SOAP protocol holding its promise to be lightweight and how does it cope with its need to be general?

Jini, building on an object-oriented model, entails the notion of types, hence providing a distributed, type safe model. In contrast, Web Services, which consist rather of documents than of object types, support the notion of types through WSDL. WSDL is a general purpose XML language for the describing the interface, protocol bindings, and deployment bindings of network services. The description of Web Services with WSDL can be compared with the definition of CORBA objects using an Interface Definition Language (IDL). The Web Service model WSDL embodies the notion of types. A question is if WSDL can assure type safety ?

The gist of the thesis is to provide the knowledge on Web Services, Jini, and the elaboration of its differences. This carries out a clear understanding of the different approaches used and the underlying technologies. These entails the evaluation of the models, having a critical look on it, and proposing enhancements. Moreover, it is our objective to evaluate these technologies in a practical environment, using the newest tools. Further, the thesis is going to answer the questions which have been outlined in this introduction.

As test basis for this evaluation serves the Microsoft .NET platform and the Java platform. In the .NET architecture, Web Services play a major role. Sun's Jini technology has a lot in common with the Web Services approach advocated by Microsoft, including the idea of services that are published on the network and registered in searchable directories. Jini's system is Java-technology centered. Theoretically, every language translating to Java bytecodes can be used to create Jini services. Even if Jini is Java-centric, its architecture promotes a communication which is protocol independent. Protocol design, like any engineering design, is often a trade-off between efficiency and generality. In systems that are designed around a one-size-fits-all protocol, such decisions need to favor generality. Is the demand of SOAP a limiting factor of Web Services?

One part of the thesis is to compare these two models by elaborating the main differences between a document-based approach and a object-based approach. The two approaches share some semblance, mainly concerning the key concepts and the architectural model. Both are based on the notion of services. Services can be made public and a center. Another important component is the lookup service which serves as a repository for services. The technology infrastructure defines, in both cases, a means to dynamically discover services. But, the underlying technologies and programming models differ considerably. The thesis takes the dynamic discovery mechanism to show the main differences emerging. What are the main differences and what is their effect on the respective model?

Part A the Road to Web Services

1 Web Service Defined

This section tries to dissect the term Web Service. Because there is no official definition of this term a myriad of different definitions are appearing. This comes as no surprise concerning the amount of interest triggered by the Web Service approach. Most of the big software companies, such as Microsoft and IBM, and many smaller ones are trying to establish themselves as provider of Web Service solutions and tools. Though the proliferation of companies focusing on Web Services is immense there is only a little difference amongst their view on what a Web Service is.

Our purpose is to give first an abstract view on Web Services and its relevant features, resulting in a concise definition. Then moving to more subjective definitions of Web Services issued by some major companies in this area, like Microsoft, IBM, and Sun Microsystems.

1.1 Abstract View on Web Services

Web Services are a description of specific functionality delivered via standard Internet protocols. The main purpose is to provide a mechanism for other services or applications to use this exposed functionality. In order to expose Web Services some problems have to be addressed. Because of the diversity of information systems which can take part in a Web Service environment, problems arising are, how to enable a uniform access to diverse information sources and operating systems, and how to deliver content to different output channels or devices. A channel can be thought of the Web, Business-to-Business communication, mobile applications, and so forth. The idea behind Web Services is to provide an abstraction from the complexity of interacting with multiple information sources, communication protocols, and delivery channels in a standardized way.

To describe the content of a Web Service in a standardized and abstract way, XML comes to rescue. Services are defined using XML as a standard syntax resulting in an uniform information access. The description of services embraces the definition of various properties of the information source to be accessed and the syntax of the service request and response is described. This results in a well-defined service contract. The usage of XML as a description language for services provides a self-describing, structured representation of data that can be implemented broadly. Every system or language providing the means to handle XML constructs can be a part in a Web Service architecture.

Services can be published in service registries enabling a global access to them. To find and use these services a flexible mechanism for service discovery is needed. Other services or applications are able to lookup dynamically services at run-time in these registries.

To summarize, a Web Service is an abstract description of application functionality mainly exposed to the Internet or Intranet. The standardized description of its functionality and the dynamic access are crucial ingredients. To provide a uniform access , they are accessible through standard Internet protocols. Web Service descriptions are carried out using XML as a description language. Standard Internet protocols, like HTTP, and XML, as a standardized syntax, are the cornerstones of Web Services.

1.2 What others are saying

To probe further in the Web Service concept it is necessary to take a more practical view on them. This is done by presenting definitions of four major software companies strongly involved in promoting Web Services. These companies, namely Microsoft, IBM, Oracle, and Sun Microsystems, agree on the importance of the Web Service approach and are currently in the process of consolidating their Web Service strategy.

The companies have submitted following definitions of Web Services:

- *Microsoft.* A Web service is a unit of application logic providing data and services to other applications. Applications access Web services via ubiquitous Web protocols and data formats such as HTTP, XML, and SOAP, with no need to worry about how each Web service is implemented. Web services combine the best aspects of component-based development and the Web, and are a cornerstone of the Microsoft .NET programming model. [KIRT]
- *IBM.* Web services are self-describing, self-contained, modular applications that can be mixed and matched with other Web services to create innovative products, processes, and value chains. Web services are Internet applications that fulfill a specific task or a set of tasks that work with many other web services in an interoperable manner to carry out their part of a complex work flow or a business transaction. [KREGER]
- *Oracle.* Fundamentally, a Web Service is a piece of logic that applications can access over a network via a standardized XML-based interface, in a platform-independent and language-neutral way. Web Services are designed to be published, discovered, and invoked dynamically in a distributed computing environment. By facilitating real-time programmatic interaction between applications over the Internet, Web Services allow companies to more easily exchange information, leverage information resources, and integrate business processes. [ORACLE]
- *Sun Microsystems.* A Web service describes specific business functionality exposed by a company, usually through an Internet connection, for the purpose of providing a way for another company or software program to use the service. [KAO]

These definitions express characteristics and basic concepts of the Web Service model. Moreover, the definitions entail promises and marketing-driven issues, which are a natural part of a company's strategy to promote their ideas and products. The below sketched points

are an elaboration of the most important statements made in the definitions. This points are examined with the purpose to find out if the promises are hold. The main points are:

- Description and access through ubiquitous Web protocols and data formats.
- Component-based development.
- Web Services represent self-describing, self-contained, modular application.
- Interoperability with other services to achieve a specific task.
- Platform-independency and language-neutrality.
- Integration of heterogeneous information systems, with the purpose to facilitate the data exchange between businesses.

In the remainder of this chapter the Web Service model is described in detail by elaborating the basic concepts and presenting the underlying technologies.

1.3 The Web Service Component Model

The Web Service model bears a close semblance to the current, common definition of component models. The idea in this section is to elaborate this statement by comparing Web Services to object-based and component-based concepts and to contrast the Web Service model with current-day, well known component models.

The combination of services, based and open standard protocols and XML standards, is a new paradigm in the way that applications and its collaborations are modeled. This "service-oriented paradigm" provides a conceptual and architectural foundation which can be implemented using a variety of platforms and programming languages. These standards, on which a web service system is built upon, allows for an implementation-neutral approach. This standard-based approach promises to facilitate the integration of heterogeneous information systems, hence resulting in a cross-language and cross-platform interoperability.

The Web Service system introduces fundamental concepts and ideas which are very similar to object-oriented systems, such as message passing, dynamic binding, and encapsulation. Though, the Web Services model goes beyond the object-based approach. It can be seen as a logical evolution from object-based systems to service-based system. Web Service systems foster significant decoupling and dynamic binding of components. All components in a system are represented as services. They encapsulate behavior and publish a messaging API to other collaborating components on the network. Services are looked up by applications using service discovery for dynamic binding of collaborations. Web Services can be dynamically composed into applications by means of a capabilities-based look-up at runtime, instead of the traditional static binding. The dynamic nature of the collaborations yields implementations that are platform- and programming language neutral, and communications mechanism independent. Further, a Web Service implementation is not restricted to a specific language paradigm, such as object-orientation. For instance, a service can be built using a script language.

Comparing the Web Service model to other so called component models, such as Javasoft's Enterprise JavaBeans (EJB) or Microsoft's Component Object Model (COM), it seems to have more in common with the characteristics defined for component models. The term "so called" is used here, because usually they are denoted as component model by their creators. But they are only partially adhering to the common understanding of a component model. Even if there is still a lot of debate on how to define what a component really is, there is a general agreement on what a component should look like. Betrand Meyer and Clemens Szyperski, both respected persons in the context of component technologies, agree in [MEYER] that components should adhere to at least the following list of characteristics: a component may be used by other software elements (clients); may be used by clients without the intervention of the component's developers; includes a specification of all dependencies (hardware and software platform, versions, other components); includes a precise specification of the functionalities it offers; is usable on the sole basis of that specification; is composable with other components; can be integrated into a system quickly and smoothly. The above mentioned component architectures have the restrictions of being language-dependent or runtime-dependent. Taking this into consideration, Web Services are leveraging components to the next level and Web Services represent the next generation, because they are not limited by these restrictions.

Like components, Web services represent black-box functionality that can be reused without worrying about how the service is implemented. Web Services provide well-defined interfaces, so called contracts, that describe the service provided. Developers can assemble applications by composing several services which can be located on the same machine or which are accessible on a remote machine. "A major thrust for the design of components is design for independence. The idea is simple: If a component is constructed with minimal, built-in dependencies on its environment, it is likely to be widely usable. Design for independence is central to the notion of components. [SZYP]" This idea expressed by Szyperski is especially true for Web Services because of their above mentioned independence concerning the implementing language and used operating system. Further, the exposition of their functionality through a standardized service description language.

Moreover, unlike current component technologies, Web Services do not use object model specific protocols such as Microsoft's Distributed Component Object Model (DCOM), Javasoft's Remote Method Invocation (RMI), which is using the Java Remote Method Protocol (JRMP), or the Object Management Group's Common Object Architecture (CORBA), using the IIOP, that require specific, homogeneous infrastructures on both the client and server machines. These low-level connection standards are, in the case of DCOM and RMI product-drive, and in CORBA's case is standard-driven.
Implementations tightly coupled to specific component technologies provide a tailored connection protocol by exploiting the underlying environment, hence resulting in a "proprietary standard" perfectly fitting the needs of its environment. Those protocols are acceptable in a controlled environment, but they become impractical on the Web. As technology changes over time, it becomes difficult to guarantee a single, unified infrastructure, resulting in heterogeneous systems. The emerging issue here is how to guarantee interoperability. Web Services take a different approach. They communicate using ubiquitous Web protocols and data formats such as HTTP and XML. Any system supporting

these Web standards will be able to support Web Services. Hence, the Web Service model uses a communication protocol which is mediated by lightweight, vendor-neutral communication technologies. This technologies allow every network-enabled system to interact.

Web Services are executables which can be linked to using XML messaging. A Web Service interface is defined strictly in terms of messages the Web Service accepts and generates. Clients of services can be implemented on any platform in any programming language, presupposed they can create and consume the message defined for the Web Service interface. This message-oriented approach is key to this component-based model, resulting in the above mentioned characteristics of being language, platform, and object-model agnostic. A Web Service can be implemented using the full feature set of any programming language, object model, and platform. Moreover, a Web Service can be consumed by applications implemented in any language for any platform. As long as the component contract that explains the service's functionality and the message sequences and protocols it expects is honored, the implementations of Web Services and Web Service consumers can vary independently without affecting the application at the other end of the conversation.

The specification of functional aspects of a Web Service is described declaratively with a description language providing metadata to empower dynamic service discovery. The service metadata contains enough information about the service's capabilities and further, it is specifying which communication protocols it supports. This feature leads to another important characteristic of the model. The feature is the possibility to compose services at run-time. This notion of execution-time building of applications is very different to the common development time deployment of components. It is a much different aspect of binding. In object oriented terms the notion of late binding is introduced. Web Services introduce the notion of an "extreme late binding". A Web Service can be composed at run-time, as opposed to objects and their collaborations adhering to a development-time model. What characterizes the Web Service architecture is that the model is very loosely coupled, using XML messaging and HTTP-based protocols to talk to each other. Therefore, Web Services can be bound to each other either at execution time or deployment time, as opposed to development time, allowing just-in-time integration. This just-in-time integration is achieved by providing a mechanism to discover components which provide a specific functionality. Because this functionality is published with a service description language, a client can find this service at run-time.

1.4 Web Service Key Specifications and Technologies

This section is to educate the reader on the major technologies entailing Web Services, and describing their contribution to the Web Service model. In the following sections these technologies are explored in detail. This investigation shall discover if the technologies are feasible and appropriate concerning the model.

The previous sections introduced Web Services and its component model. The next step is to give an overview on the underlying technologies. These technologies enable the building of

service components and are a crucial part of the technical framework. Later sections give a more detailed description of these technologies.

There are a few key specifications and key technologies very likely to be encountered when building and consuming Web Services. These specifications and technologies address some distinct requirements for service-based development. The issues addressed are as follows:

- the need for a standard way to represent data
- a common, extensible, message format
- a uniform, extensible, service description language
- a way to discover services located on a particular Web site.

XML seems to be the obvious choice for a standard way to represent data. Most Web Service-related specifications use XML for data representation, as well as XML Schemas to describe data types. The important point is that XML is the basis for a distributed object-like model which includes messages, state, methods, and object-oriented interfaces. In section 2.7 "XML Schemas"(25) is shown that XML Schema represents a feasible approach to reflect an object-based system into a document approach, but has its limits in representing some object-based related issues, such as distributed references. Another important point is, supporting XML with its widespread use and tool support, results in chances that developers have some previous knowledge of XML.

As a message format, the Simple Object Access Protocol (SOAP) turns to be out, to be the most supported communication model, in the scope of Web Services to date. As already mentioned before, Web Services are protocol independent. Hence, other protocols can be conceived as part of a Web Service architecture. SOAP defines a lightweight protocol for information exchange. Part of the SOAP specification defines a set of rules for how to use XML to represent data. Other parts of the SOAP specification define an extensible message format, conventions for representing remote procedure calls (RPC) using the SOAP message format, and bindings to the HTTP protocol.

SOAP is still just an RPC, calling low-level functions and leaving most communication definitions to a developer. In order to make it easier to use, there is a need for a format to describe services that can be invoked by SOAP. Following from this, given a Web Service, it is necessary to have a standard to document what messages the Web Service accepts and generates, that is to document the Web Service contract. A standard mechanism makes it easier for developers and developer tools to create and interpret contracts. The Web Service Description Language (WSDL) is an XML-based contract language jointly developed by Microsoft and IBM. WSDL can be seen as a complement to SOAP, as it facilitates interoperability between Web Services. Like the Interface Definition Language (IDL), which acts as a service describer with CORBA, WSDL is an XML syntax to describe Web Services. Most of the current SOAP implementations support this description language. Applications, that use SOAP as their communication protocol, can self-configure exchanges between Web Services, while hiding most of the low-level technical details.

Developers will also need some way to discover Web Services. In many cases the developer will not know the Internet address where services can be found. Universal, Description, Discovery, and Integration (UDDI) specifies a mechanism for Web Service providers to advertise the existence of their Web Services and for Web Service consumers to locate Web Services of interest. The discovery aspect appearing can not be seen alone with UDDI. WSDL is a major part of the evolving technology around the discovery aspects by providing the necessary service metadata. UDDI offers a technical framework that is independent from platforms and totally open, so that enterprises can find one another, define how they will interact on Internet, and define how information should be shared using a worldwide registration system. The result of this project will be, that enterprises will be able to enter the business-to-business world by using a standard approach. The effort behind UDDI is strongly business oriented. The investigation of UDDI yielded, that this mechanism is very important in a business related use of Web Services, for instance for discovering a service providing a specific business functionality. The idea expressed by UDDI is to built business registries, where every company can register its supported services. Client can discover services using the registries as a kind of yellow pages. Different roles can be conceived here. That is the service provider, who provides a service interface for a software asset that manages a specific set of tasks. A service provider node can represent the services of a business entity or it can simply represent the service interface for a reusable subsystem. A second role is the service requestor, who wants to discover and invoke other software services to provide a business solution. The third role can be seen as a service broker. A service broker acts as a repository or yellow pages for software interfaces that are published by service providers. A business entity or an independent operator can represent a service broker.

The bottom line is, that component models, with the purpose to be used throughout the Web, needs to be augmented with a few other platform devices, which maintain the ubiquity and simplicity of the Web, to constitute a more functional platform. The full-function web service platform can be thought of as XML plus HTTP, as the standard base of the platform. SOAP, WSDL, and UDDI are extensions, build on these standards, to provide a higher-level programmatic access to Web Services. The Web Service model is promoting these standards that are independent both of programming languages and of operating systems to ensure an absolutely open environment.

1.5 Open Standards

What can be derived from the above mentioned idea of Web Services, that the model is only going to work, if open standards are provided and supported by Web Service development tools. For this sort of openness to work, standardized formats and methods are crucial ingredients. Concerning the above mentioned technologies SOAP and WSDL, they were submitted to the World Wide Web Consortium (W3C), and actually still a work in progress, which means that there may be changes or updates. But this shows the commitment, the involved companies like Microsoft and IBM have, to promote Web Services. SOAP and WSDL, were submitted, amongst others, by Microsoft and IBM. Further, the industry is

attempting to take advantage of Internet Task Force (IETF) standards such as Extensible Markup Language (XML) and the Hypertext Transport Protocol (HTTP).

The key to Web Service interoperability is reliance solely on Web standards. However, simply agreeing that Web Services should be accessed through standard Web protocols is not sufficient to make things interoperable. Web Service development tools have to adhere only to these standards. As experienced before, for instance in the case of CORBA, companies tend to compromise standards with proprietary extension, to gain a business advantage. This time the companies seem to agree on the importance of adhering to standards and interoperability.

To conclude, what makes Web Services model approach very different and interesting is its commitment to open standards. To date, the submitted specifications are relatively unstable, but robust enough to predict their future. Only the fact, that Microsoft and IBM are both promoters of Web Services, show the impact Web Services will have in the nearer future. But everything relies on interoperability issues.

2 The Simple Object Access Protocol

This chapter explains the Simple Object Access Protocol (SOAP) [W3CSOAP] and its main constituent technologies. It is necessary to understand its underlying technologies to get the whole picture of SOAP. SOAP has recently gained a lot of marketing momentum, and is promoted by Microsoft and IBM. They agree, that SOAP should become the main protocol used in conjunction with Web Services. It is the purpose in this section to present the underlying technologies SOAP is based upon, and to provide an evaluation of SOAP. This investigation should discover, if the SOAP specification provides appropriate functionality to be used in a Web Service architecture. Further, it is examined if existing tools support adhere to the specification and, above all, if they are interoperable.

2.1 Introduction

The Simple Object Access Protocol (SOAP) is a simple and lightweight protocol, specified with a minimal set of conventions for exchanging structured and typed information between peers. SOAP is lightweight in the sense that it does not itself define any application semantics such as a programming model or implementation specific semantics. Rather, it defines a packaging model and an data encoding standard to express application semantics. The specification is submitted to the W3C and is currently work in progress [W3CSOAP]. One main purpose of SOAP is to provide only a minimal set of conventions, resulting in a protocol, that can be used in a variety of communication models, ranging from remote procedure call mechanisms to message-oriented models. SOAP represents the base technology of the decentralized and distributed Web Service environment. The protocol is based on the Extensible Markup Language (XML), and as the investigation has shown, often used in conjunction with the Hypertext Transport Protocol (HTTP). Web Service development tools, such as Microsoft .NET and the Web Service Tool Kit (WSTK) from IBM, support at least this combination.

The protocol utilizes this Internet-based protocol as its RPC-style transport mechanism, to deliver method requests which are encoded as XML. This combination of XML and HTTP yields a protocol which is platform-neutral and language-independent. This statement is based on the experience gained implementing the bridge described in Part C The Web Service-to-Jini bridge" (71). Though HTTP is often mentioned as the transport layer for SOAP, it is transport independent. Other Internet-based protocols, such as FTP and SMTP, can also be used. The current specification defines protocol bindings only for HTTP and HTTP Extension Framework. Current development tools, such as Microsoft .NET and IBM's Web Service Tool Kit, allow SOAP in conjunction with HTTP only.

The SOAP message exchange model consists of one-way transmissions from sender to receiver which can be combined to be used as a request/response pattern to simulate an RPC-style exchange mechanism. SOAP messages rely on XML Namespaces and the XML Schema definition language. The XML encoding makes SOAP messages simple to read and parseable by humans and machines alike. The technical requirements to use SOAP are very

few. All is needed is an XML parser, XML Namespace and XML Schema compliant, used in combination with an operating system supporting current Internet protocols, above all the HTTP protocol. Thus, SOAP can be used in various languages running on multiple platforms.

SOAP's main purpose is to facilitate the integration of heterogeneous information systems, hence enabling interoperability among applications and platforms. SOAP's interoperability arises from its characteristics mentioned above.

2.2 Current-day middleware

SOAP can be seen as another middleware which is taking its place amongst a plethora of already existing middleware technologies, such as CORBA and DCOM. But SOAP is promising to have a major advantage compared to current-day middleware, and that is interoperability. The promise of being platform and language independent, and further providing an Internet-based protocol makes SOAP much different. Hence, the justification for this new distributed computing model is cross-platform and cross-programming language interoperability. In chapter 5 (71) we will evaluate SOAP's promise of interoperability, which we conceive as a very important issue of SOAP implementations. If SOAP implementations are failing to provide seamless integration and interoperability, SOAP will fail as technology. The remainder of this section will focus on the shortcomings of previous solutions and future requirements for successful solutions.

At this time, the are three dominant Remote Procedure Call (RPC) protocols used in the industry. These are DCOM , Java's RMI, and CORBA's Internet Inter-ORB Protocol (IIOP). We will focus on DCOM and CORBA here. Even if both approaches have conceptual similarities, for instance, both protocols use endpoint identifiers to identify a target object on the server-side, and both use method identifiers to determine the signature of the method to be invocated, there are differences, which impact crucially interoperability.

One major difference is the encoding styles used for the parameter values in the payload. In DCOM, the payload is written in a format known as Network Data Representation (NDR). In IIOP/GIOP, the payload is written using Common Data Representation (CDR) format. Both NDR and CDR deal with the differing data representations used on various platforms. However, in [BOX] is investigated that there are some minor differences between these two formats that make them incompatible with one another.

Another key distinction is how CORBA and DCOM uniquely indentify server-side endpoints and objects across network boundaries. CORBA/IIOP uses a representation called Interoperable Object Reference (IOR). IOR's contain addressing information, which can be transmitted across network boundaries, so that any CORBA product can resolve this reference to an object endpoint. DCOM provides a similar representation called OBJREF. This unique identifier combines distributed reference binding with endpoint identification. Both approaches are crucial for uniquely identifying distributed CORBA or DCOM resources. Different research projects have shown that different representations of network resources

are a main hindrance for interoperability. For instance, in [GISOLFI] is elaborated that "IORs do not correlate to OBJREFs, which results in an interoperability problem between CORBA and DCOM applications." And [BOX] points out that these protocols tend to be not interoperable.

A shortcoming both approaches are sharing is that both are based on a single vendor solution. Using the protocols to a maximum advantage, entails, in the case of CORBA, vendor-specific extensions which are very unlikely to work unless all applications are built against the same ORB product. Though both protocols have been implemented on a variety of platforms and products, the reality is, that a given deployment needs to use a single-vendor's implementation. With DCOM, that comes as no surprise, the Windows operating system above Windows NT, has to be used. DCOM was ported to LINUX, but with some major functionality lost, as expressed in [BOX]. CORBA interoperability has been an issue for several years now. The reality has been that vendors compete on ORB implementations, hence there is no motivation from a business perspective to achieve interoperability. The result is that every machine runs the same ORB product. Even if it is possible to get two CORBA products to call one another using IIOP, many of the higher level services, such as security and transaction management, are not generally interoperable at this time [BOX].

Moreover, both protocols are not suitable for communication across the Internet. Experience has shown that DCOM and CORBA are both reasonable protocols for server-to-server communications. However, both, DCOM and IIOP, have severe weaknesses for client-to-server communications, especially when the client machines are scattered across the Internet. The problem appearing, regarding Internet communication, are firewalls which separates the client and server machines. Firewalls are often so restrictive that the likelihood of either IIOP or DCOM packets getting the permission to pass is very low. This is due to the HTTP-based communication on the Internet. Many providers of middleware technology have HTTP tunneling support built-in in their products. Due to a lack of standard they are not interoperable and hence very sensitive to configuration of a Web server.

None of these issues impact the use of DCOM or IIOP within a server farm. Relying on a closely administered environment makes a consistent configuration possible. This empowers system administrators to use the full functionality provided by either product. The relatively small number of machines also helps to keep the costs of using commercial ORB products under control, as a smaller number of ORB licenses are needed. If IIOP is only spoken within the server farm, a smaller number of ORB licenses are needed. Finally, it is likely that all of the host machines in a server farm will have direct IP connectivity, removing the firewall-related problems of DCOM and IIOP.

The shortcomings of these approaches are the major strengths represented in SOAP. SOAP is a solution which is based on open standards that can truly support interoperability, as long as the implementing tool adhere to the standard. The major difference that SOAP has as a distributive computing solution, is that it is achieving interoperability through open standards at the specification level and the implementation level.
Concluding from the previously described characteristics and shortcomings of the different communication models, the place where there is a need for XML messaging mechanisms like

SOAP is on the Internet, where traffic in protocols like DCOM and CORBA IIOP is not supported. For one thing, firewalls not let them pass. SOAP harnesses existing Internet protocols that are already ubiquitous, meaning HTTP primarily, but also SMTP, FTP, and secure Web protocols such as the Secure Socket Layer (SSL). The binary distributed computing protocols also have an inherently limited audience. DCOM is limited to connecting Windows computers, while RMI is confined to Java platforms, and CORBA tends to be limited to high-end enterprise environments. They are tightly coupled to a particular environment and technology, whereas XML messaging is a loosely coupled solution. On the Internet, it is not predictable to what platform a connection will occur.

2.3 SOAP's Technical Background

To understand the simplicity of SOAP and its close relation to HTTP, an example is presented, showing SOAP's syntax structure and its inherent semantics. This technical examination of a SOAP call and response example, will provide the knowledge necessary to understand arising interoperability problems and negative side effects introduced by SOAP.

2.3.1 HTTP, the Internet RPC mechanism

The Hypertext Transfer Protocol (HTTP) is the most used application protocol used on the Internet. It is the only protocol which is tolerated by firewalls and supported by nearly every operating system and Web browser. The following investigation will show that HTTP is an RPC-like protocol that is simple to use. To date, it is widely deployed, and very likely to function in the face of firewalls. Protocols like IIOP or DCOM are commonly used in in-house networks, whereas HTTP provides the connecting means to the client. HTTP requests are typically handled by Web server software, but an increasing number of application server products are supporting HTTP as a native protocol in addition to DCOM and IIOP, such as IBM's WebSphere application server.

HTTP, like DCOM and IIOP, supports a request/response communication model over TCP/IP. An HTTP client connects to an HTTP server using TCP, usually using port 80 as standard port. After establishing the TCP connection, the client can send an HTTP request message to the server. The server then sends an HTTP response message back to the client after processing the request. Both the request and response messages can contain arbitrary payload information, typically tagged with the Content-Length and Content-Type HTTP headers.

The following is a legal HTTP request message:

```
POST /foobar HTTP/1.1
Host: 209.110.197.12
Content-Type: text/plain
Content-Length: 12

Text content
```

The HTTP headers are just plain text which makes them easy to use in conjunction with programming environments popular in Web development. A lot of programming languages can be used, for instance in combination with Common Gateway Interface (CGI), to implement the server-side application logic. The first line indicates that this is an HTTP POST request that conforms to the rules per HTTP 1.1. The target for the request is identified by an Uniform Resource Identifier (URI). The Request-URI is simply a token used by the HTTP server software to identify the target of the request. All these conventions are based on a standard specified by the Internet Engineering Task Force (IETF). The great success the Internet has gone through, is, among other things, based on the fact that it uses commonly accepted standards, such as HTTP. Among other things, the standard specifies a fixed number of HTTP methods. GET is the HTTP method used to navigate through the Web. POST is the most commonly used HTTP method for building Web-based applications. Unlike GET, POST allows arbitrary data to be sent from the client to the server. HTTP/1.1 added several features to its predecessor (HTTP/1.0), including support for a flexible data transfer and explicit support for keeping TCP connections alive across HTTP requests, which is a requirement for a response/request communication model introduced by SOAP. The next lines of the request indicate the size and type of the request payload. The Content-Length header specifies the number of bytes of payload information. The Content-Type identifier specifies the encoding style of the payload information. HTTP, much like DCE, allows the client and server to negotiate the transfer syntax used to encode information. Most DCE applications, such as DCOM, use NDR to encode its data. Most Web applications use text/html or other text-based syntaxes. The request then contains raw bytes whose syntax and length are identified by the Content-Length and Content-Type HTTP headers. In this example, the content is the 12-byte plain text string "Text content". The text-based HTTP payload information makes it very easy to understand and to use, compared to a binary protocol.

After processing the request, the HTTP server is expected to send an HTTP response back to the client. The response must contain a status code indicating the outcome of the request. The response can also contain arbitrary payload information much like the request message. The following is an HTTP response message:

```
200 OK
Content-Type: text/plain
Content-Length: 12

Message received
```

In this case, the server returned a status code of 200, which is the standard success code for HTTP. Had the server been unable to decode the request, it would have returned the following response instead of the one shown previously:

```
400 Bad Request
Content-Length: 0
```

Had the HTTP server decided that requests for the target URI should be temporarily redirected to a different URI, the following response would have been returned:

```
307 Temporarily Moved
Location: http://123.456.789.00/urn
Content-Length: 0
```

This response informs the client that the request could be satisfied by retransmit ting it to the endpoint identified in the Location HTTP header. All of the standardized status codes and headers part of the IETF standard. Very few of them relate directly to SOAP users, with one important exception. In HTTP/1.1, the underlying TCP connection is reused across multiple request/response pairs. The HTTP Connection header allows either the client or the server to close the underlying connection. By adding the following HTTP header to a request or response, both sides are required to shut down their TCP connections after processing the request.

```
Connection: close
```

This section has shown a subset of HTTP's functionality which provides the basis for a RPC-style mechanism SOAP. It is examined now, how SOAP is using this functionality to provide a Internet-based remote procedure call model, and conserving the ease of ease introduced by HTTP.

2.4 SOAP Structure

SOAP uses the aforementioned HTTP protocol to a great extent to provide an RPC mechanism with an easy to use, ubiquitous Internet-based protocol. One of the advantages of HTTP is its wide deployment and acceptance. As already mentioned above, SOAP is protocol independent, and supports an RPC-style and message based communication model. However, in the current state, SOAP is mostly used with HTTP over an RPC-style protocol. Current development tools, supporting the construction of Web Services, such as Microsoft .Net and IBM's WSTK, are providing SOAP bindings over HTTP. Therefore we will focus on this combination.

Remote procedure calls in SOAP are essentially client-server interactions over HTTP where the request and response comply with SOAP encoding rules. SOAP allows methods to be invoked against endpoints over HTTP. A SOAP endpoint is identified by a URL, just like any other HTTP-based resource. A SOAP method is uniquely identified by a namespace Uniform Resource Identifier (URI) and an NCName. A URI is simply a formatted string that uniquely

identifies a resource. The NCName maps to a symbolic name of the method. The Request-URI in HTTP is typically used at the server end to map to a class or an object, but this is not mandated by SOAP. The namespace URI scopes the method name, much like an interface name scopes a method in Java or CORBA. The common HTTP header is extended with an additional SOAPAction parameter. This HTTP header `SOAPAction` specifies the interface name and the name of the method to be called on the server.

The SOAP message is an XML document whose root element, the *Envelope*, specifies the overall structure of the message, its intended recipient, and other attributes of the message. SOAP specifies a remote procedure call convention, which includes the representation and format to be used for calls and responses. The HTTP payload for a SOAP method request contains the information needed to invoke the request. A method call is modeled as a compound data element consisting of a sequence of fields, one for each parameter. A return structure consists of the return value as well as the out and in/out parameters. SOAP encoding rules specify the serialization for primitive and application-defined datatypes.

HTTP is a fairly functional RPC protocol that provides most, of the functionality of IIOP or DCOM in terms of framing, connection management, and support for serialized object references. URLs are surprisingly close to IORs and OBJREFs in functionality. What HTTP lacks is a single standard format for representing the parameters of an RPC call. This is where XML comes into play. Like NDR and CDR, XML is a platform-neutral data representation mechanism. XML allows data to be serialized into a transmissible form that can be decoded on any platform, presupposed an XML parser is provided.

The format of data and the protocol used to exchange is a determining factor in the degree of interoperability among applications. The lack of a reliable and commonly understood data-exchange format has limited effective communication between heterogeneous systems, as in the case with the current-day protocols mentioned above. XML has emerged as a standard for representing data in a platform-independent way. XML is, in short, essentially a tree-oriented data representation language that is simple to generate and parse. Its simplicity and platform independence makes XML a perfect solution for an encoding standard.

Also, HTTP has emerged as a simple, universally supported protocol for exchanging data over the Internet. HTTP requests/replies are passed through firewalls and handled securely, supported by various security standards. Thus, moving XML data via HTTP is a way for distributed applications to communicate with each other. That is precisely what SOAP does. By expressing RPCs independent of platforms, it opens the possibility of implementing other architecture-specific protocols in SOAP. One use of SOAP which can be perceived is to use it as an intermediary protocol into which other protocols can be easily translated. The bridge described in chapter 5 uses SOAP to make calls from a Java environment to a Microsoft environment.

2.5 The Structure of a SOAP Document

To present the overall structure of a SOAP document a complete SOAP/HTTP request, with the content, encoded as XML, is shown. Although this particular set up uses HTTP to deliver SOAP messages, SOAP can ride on any other transport protocol. SMTP, the Internet email protocol, can be used to deliver SOAP messages. The header differs between transport layers, but the XML payload remains the same.

A SOAP request is sent as an `HTTP POST` with the content type set to text/xml and a field called `SOAPAction` set to either an empty string or the name of the SOAP method. The SOAPAction field allows a receiving Web server to detect incoming SOAP messages and provide appropriate means to handle them. For instance, for routing or filtering reasons. The SOAP specification does not mandate the `SOAPAction` field to contain a value. At the first look, this seems to cause no problem. Examining the SOAP specification shows that, on the one hand, the specification tries to be as flexible as possible by not imposing to many restrictions. As in the case of the `SOAPAction` field which leaves the interpretation to the implementers. But on the other hand, it is this flexibility which can compromise interoperability. For instance, there are tools which require the `SOAPAction` field is given a value. They use this value to dispatch the SOAP call to the appropriate method call. Apache SOAP is using the SOAPAction field for that reason. If another tool does create SOAP calls without this field value, which would be standard conformant, Apache SOAP would not be able to handle the call.

The structure of a request call defined with SOAP consists of the three parts. The first part is an *envelope* that defines a framework for describing what is in a message and how to process it. The envelope defines the various namespaces that are used by the rest of the SOAP message, typically including the SOAP Envelope namespace, XML Schema for Instances (XSI), and XML Schema for DataTypes (XSD). Second, a set of encoding rules is defined. This is used for expressing instances of application-defined datatypes. SOAP allows hierarchically structured queries and responses, and specifies serialization of primitive datatypes, such as string, numeric and date, and aggregates like arrays and vectors. New types may be defined using the `<complexType>` construct inside a schema definition. The default SOAP encoding scheme uses the <type> attribute, which can be found in the Schema for Instances namespace, to indicate an XSD type. XSD defines several basic types, including int, byte, short, boolean, string, float, double, date, time and URL. It also specifies a format for sending arrays. Because SOAP is intended to be platform and language neutral, XSD does not define formats for encoding objects or structures unique to a single language. XML Schemas are further investigated in section 2.7 ”XML Schemas” (25).

The last part defines a convention for representing remote procedure calls and responses. SOAP can potentially be used in combination with a variety of other protocols. However, in the current specification, the only bindings defined is how to use SOAP in combination with HTTP and HTTP Extension Framework.

The XML schema of an SOAP request is looking as follows:

```
<schema
 targetNamespace='urn:schemas-xmlsoap-org:soap.v1'>
  <element name='Envelope'>
    <type>
      <element name='Header' type='Header'
               minOccurs='0' />
      <element name='Body' type='Body'
               minOccurs='1' />
    </type>
  </element>
</schema>
```

This XML schema shows that a SOAP message needs a mandatory SOAP envelope, which contains the namespace definitions, followed by an SOAP header. The Header is an optional element that can carry auxiliary information such as authentication, or transaction management. Any element in a SOAP processing chain can add or delete items from the Header. Elements can also choose to ignore items if they are unknown. If a Header is present, it must be the first child of the Envelope. The third part is a mandatory SOAP body element. The Body is the main payload of the message. When SOAP is used to perform an RPC call, the Body contains a single element that contains the method name, arguments, and Web service target address. If a Header is present, the Body must be its immediate sibling, otherwise it must be the first child of the Envelope.

2.5.1 A SOAP Call

The next figure shows an example of a request/response SOAP message pair which uses HTTP to deliver the message. As already mentioned, SOAP can be delivered but any other protocol, such as SMTP or FTP. The XML payload will be the same but the headers are changing.

A SOAP message embedded in an HTTP request

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
       <m:GetLastTradePrice xmlns:m="Some-URI">
           <symbol>DIS</symbol>
       </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The HTTP protocol uses a simple POST request via the network. The next line denotes the target site. The content type has to be set to `text/xml` to provide the appropriate content type. The `SOAPAction` field enables a Web server to detect that this request is an SOAP message, so the server can filter the request or process other actions. The namespace-qualified tagname in the Body element matches the `SOAPAction` in the HTTP header exactly. This redundancy is to allow the HTTP-based infrastructure, such as firewalls or Web servers, to process the call without having to parse the whole document.

The `<body>` element contains the XML payload. Here all the information is provided which is necessary to invoke a Web service operation. In the SOAP `<envelope>` element, various namespaces are defined. The schemas defined in this namespaces can be used by the rest of the SOAP message, because they are defined globally. In this case, the typically included namespaces are presented. That is the namespace containing the vocabulary for the SOAP envelope elements. Further, the encoding standard that is going to be applied is added. Here it is the namespace for XML schemas.

2.5.2 The SOAP Response

A SOAP response message is depicted in the following:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
       <m:GetLastTradePriceResponse xmlns:m="Some-URI">
           <Price>34.5</Price>
       </m:GetLastTradePriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The figure above shows the SOAP response message created by some server-side process. The SOAP response is embedded in an HTTP response containing the results of the operation. The result will always be inside an element whose name matches the method suffixed by “Response”.

Upon receiving the SOAP request, the server-side software is expected to execute some code that creates the SOAP response. How this happens is completely outside the scope of the SOAP protocol. Possible server-side implementations are a CGI program containing a Perl script, a Java servlet which constructs the XML document, a CORBA server object, a .NET application, and so forth. This is part of SOAP’s independence of the platform and language

used. Once the server-side operation has executed, an HTTP response message will be returned to the client containing the results of the operation.

2.5.3 SOAP Exceptions

The SOAP protocol will create a SOAP exception if a server is not able to respond to a SOAP request. This can be due to several reasons, such as a generic HTTP errors. Different error sources are explicitly dealt with in the SOAP specification and different error code are included in the generated SOAP fault structure. A fault message is simply an instance of the following type encoded as the root element in the Body part of a SOAP message:

```
<schema
  targetNamespace='urn:schemas-xmlsoap-org:soap.v1'>
  <element name='Fault'>
    <type>
      <element name='faultcode' type='string' />
      <element name='faultstring' type='string' />
      <element name='runcode' type='string' />
      <element name='detail' />
    </type>
  </element>
</schema>
```

The faultcode attribute must contain either a well-known SOAP fault code as an integer or a namespace-qualified value that is application-specific. The faultstring accessor contains the human-readable description of the error that occurred. The runcode accessor contains a string, whose value is indicating whether the requested operation was actually performed prior to the error generation. The detail accessor is optional, and is used to contain an application-specific exception object. It is up to the receiving SOAP client, what actions to take.

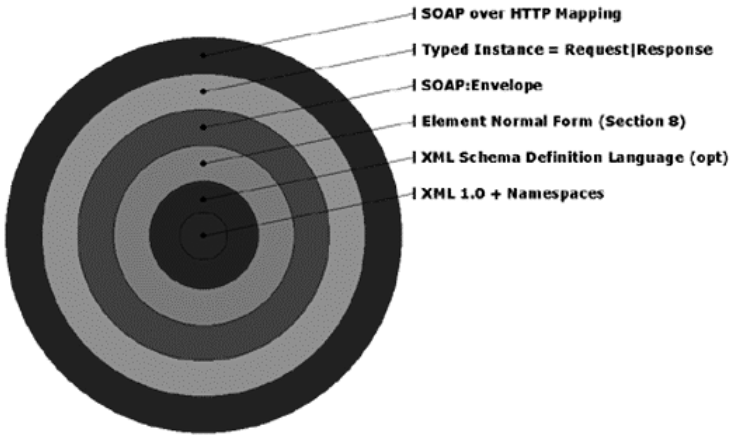
A SOAP request and response pair can be constructed as quite as easy as in the case with HTTP. A developer can introspect SOAP calls for debugging reasons, or can even built SOAP responses by hand, due to SOAP's simplicity.

2.6 SOAP's Constituent Parts

The SOAP specification is relatively short. This is due to the fact that SOAP can built on already existing standards. The SOAP specification is not demanding the use of specific standards, rather it leaves is open which technology to use, as long as they adhere to the rules defined by its own specification. The following figure acts as a reasonable decomposition of the SOAP protocol. These technologies were chosen because they reflect the implementation approach of many tools existing today.

The core of SOAP is the XML recommendation and XML Namespaces. This reflects the fact that SOAP is simply an application of XML. The next layer is the XML Schemas specification.

While SOAP does not mandate the use of XML Schemas, it was designed to allow them to act as its type description language. Comparing the encoding rules in the SOAP specification with the XML Schema specification, shows the close relationship between the two. Microsoft's and IBM's SOAP tool are using XML Schemas as encoding standard. XML Schemas and XML Namespaces are not SOAP-specific. Rather, these are two technologies that SOAP utilizes. SOAP introduces the element-normal-form encoding style which is described in the SOAP specification. Further the figure defines an request/response communication model and the use of SOAP over HTTP. As already mentioned, the technologies chosen here are not mandatory. Other means can be provided, such as SMTP as protocol, or a message-based communication protocol.



Two important technologies sketched in the figure above are XML Schemas and XML Namespaces. The combination of them can be used to encode method parameters, which are going to be sent through the wire. They are the means for mapping object-based constructs into an XML syntax, and providing an extensible and reusable mechanism to define new datatypes. The following section will probe further into this two technologies.

2.7 XML Schemas

This section will investigate whether XML Schemas [XMLSCHEMA0] provide the necessary mechanisms to map typed object graphs to an XML representation. The purpose is to examine what means are provided to represent object-oriented notions like inheritance, classes, and objects as XML constructs. The XML Schema specification is an essential part of the scope of Web Services. SOAP and the later described Web Service Description Language (WSDL) are both based on XML Schema.

SOAP needs a means to serialize and deserialize a graph of objects in a language independent and type-safe way. SOAP defines the rules and requirements for an encoding

schema in its specification. Even if XML Schema is not a mandatory part of this encoding schema in SOAP, it is very likely to be used for this purpose. The encoding schema sketched in the SOAP specification bears a close resemblance with the XML Schema specification.
One important issue concerning Web Services and the Web Service-to-Jini bridge to be built is that the underlying technologies should be able to map object-oriented types into a document-oriented model, and ensuring type safety across language boundaries.
We will use the following example schema to explain some of the important parts of the XML Schema specification. The purpose is to show how these rules can be harnessed to map object-oriented constructs into XML documents without losing type information.

2.7.1 XML Schema Instance Documents

The purpose of a schema is to define a class of XML documents. An XML document class can be seen as a template for so called instance documents. Instance documents conform to a particular XML schema. They have the same relationship as classes and objects in the object-oriented world. The next figure is showing and XML Schema instance document representing a purchase order. This example is part of the XML Primer [XMLSCHEMA0].

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
    <shipTo country="US">
        <name>Alice Smith</name>
        <street>123 Maple Street</street>
        <city>Mill Valley</city>
        <state>CA</state>
        <zip>90952</zip>
    </shipTo>
    <billTo country="US">
        <name>Robert Smith</name>
        <street>8 Oak Avenue</street>
        <city>Old Town</city>
        <state>PA</state>
        <zip>95819</zip>
    </billTo>
    <comment>Hurry, my lawn is going wild!</comment>
    <items>
        <item partNum="872-AA">
            <productName>Lawnmower</productName>
            <quantity>1</quantity>
            <USPrice>148.95</USPrice>
            <comment>Confirm this is electric</comment>
        </item>
        <item partNum="926-AA">
            <productName>Baby Monitor</productName>
            <quantity>1</quantity>
            <USPrice>39.98</USPrice>
            <shipDate>1999-05-21</shipDate>
        </item>
    </items>
</purchaseOrder>
```

The instance document contains a root element, `purchaseOrder`, and different subelements namely `shipTo`, `billTo`, `comment`, and `items`. The whole document represents a tree-like structure, hence the subelements can have subelements again, and so on. The leaves of this tree like structure are concrete values, such as numbers or strings. XML Schema datatypes can be distinguished in simple types and complex types. Simple types consist only of concrete values, whereas complex types can have other subelements such as child nodes or additional attributes. Attributes can carry only simple types.

2.7.2 XML Schema Class Documents

The class document defines a template for its instances, by specifying the types and elements which have to be used. The vocabulary used in the class schema is defined in different sources. A vocabulary defines XML schema types which can be reused. Some of the elements are defined directly in the instance document, others can be defined in a separate XML Schema. In this case other types, such as `<element>` or `<complexType>`, are part of the default vocabulary introduced in the XML Schema specification. All the XML Schema vocabulary is part of the `http://www.w3.org/2001/XMLSchema` namespace.
The following document shows the class document of the aforementioned instance document.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>

 <xsd:element name="comment" type="xsd:string"/>

 <xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
   <xsd:element name="shipTo" type="USAddress"/>
   <xsd:element name="billTo" type="USAddress"/>
   <xsd:element ref="comment" minOccurs="0"/>
   <xsd:element name="items"  type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
 </xsd:complexType>

 <xsd:complexType name="USAddress">
  <xsd:sequence>
   <xsd:element name="name"   type="xsd:string"/>
   <xsd:element name="street" type="xsd:string"/>
   <xsd:element name="city"   type="xsd:string"/>
   <xsd:element name="state"  type="xsd:string"/>
   <xsd:element name="zip"    type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN"
     fixed="US"/>
 </xsd:complexType>

 <xsd:complexType name="Items">
  <xsd:sequence>
   <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
```

```
      <xsd:complexType>
       <xsd:sequence>
        <xsd:element name="productName" type="xsd:string"/>
        <xsd:element name="quantity">
         <xsd:simpleType>
          <xsd:restriction base="xsd:positiveInteger">
           <xsd:maxExclusive value="100"/>
          </xsd:restriction>
         </xsd:simpleType>
        </xsd:element>
        <xsd:element name="USPrice"  type="xsd:decimal"/>
        <xsd:element ref="comment"   minOccurs="0"/>
        <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
       </xsd:sequence>
       <xsd:attribute name="partNum" type="SKU" use="required"/>
      </xsd:complexType>
     </xsd:element>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:schema>
```

The Schema document defines those elements used in the instance document. The document consists of elements and subelements, most notably `<complexType>`, and `<simpleType>`. These type definitions determine the appearance of elements and their content in instance documents.

The elements used in the schema are all prefixed with `xsd`. The schema document declares a namespace in the first line, namely `xmlns:xsd="http://www.w3.org/2001/XMLSchema"`. This document contains the default XML Schema vocabulary. The prefix `xsd` denotes the namespace. It is not mandatory to use exactly this prefix but it is commonly used. The built-in simple types, being a part of the default namespace, have the same prefix. The purpose of the association is to identify the elements and simple types as belonging to the vocabulary of the XML Schema language rather than the vocabulary of the schema author.

This XML schema provides the information needed to be translated in a Java class. A strategy would be to map all the complex types into Java classes and its subelements into variables. If a subelement represents another complex type, this will first be mapped into a Java class. To reflect the relationship between the element and its subelement, the Java class is holding a reference to it. Hence, the above example would result in the generation of three Java classes, namely the `PurchaseOrder`, `USAdress`, and `Items`. The `PurchaseOrder` class has two references to the `USAdress` class, and a reference to `Items`. XML Schema instance documents are then used to express object instances of this classes.

2.7.3 Definition of New Datatypes

What is needed is a flexible mechanism which can be extended by providing new data types. As already described above, there is a basic difference between complex types and simple types. Complex types allow elements in their content and may carry attributes, and simple types do not. The creation of a new type is called a definition. Defined types can then be

declared with their specific type name and namespace and be used in instance documents. Again, this is the same mechanism as in object-oriented languages. The purpose is to provide the possibility to extend the language vocabulary by defining new and reusable types, built on built-in or previously defined types.

We will focus first on the definition of complex types. There are three elements mainly used for this purpose. That is the <complexType>, the <element>, and the <attribute>. Further, an element can reference another element which is outside the scope of the type definition.

```
<xsd:complexType name="USAddress" >
  <xsd:sequence>
   <xsd:element name="name"   type="xsd:string"/>
   <xsd:element name="street" type="xsd:string"/>
   <xsd:element name="city"   type="xsd:string"/>
   <xsd:element name="state"  type="xsd:string"/>
   <xsd:element name="zip"    type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
 </xsd:complexType>
```

In a schema document, <element> declares an element, and attributes are declared using the <attribute> element. The example shows the USAddress type. The type is defined as a complex type, and within the definition of USAddress five element declarations and one attribute declaration are defined. An instance document using this new type has to follow exactly the predefined structure. The consequence of this definition is that any element with the type USAddress must consist of five elements and one attribute. These elements must be called name, street, city, state and zip as specified by the values of the declarations' name attributes, and the elements must appear in the same sequence in which they are declared. The subelements are all simple types but complex types can also contain other complex types or references to types. All the simple types are “strongly typed” to use this notion from object-oriented languages. Strong typing is absolutely necessary to map XML Schemas to a language like Java. The name type is defined as the built-in type string, which may be transformed to a Java string. The element zip is a decimal.

The next example contains element declarations involving complex types. The PurchaseOrderType contains a complex type. This type is declared like the other types, by using the type attributes.

```
<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
   <xsd:element name="shipTo" type="USAddress"/>
   <xsd:element name="billTo" type="USAddress"/>
   <xsd:element ref="comment" minOccurs="0"/>
   <xsd:element name="items"  type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
 </xsd:complexType>
```

The declaration of this type in an instance document follows the same rules as the declaration of a simple type. The elements named `shipTo` and `billTo`, each containing the five subelements that were declared as part of `USAddress`. The attribute declarations reference simple types. Attributes cannot contain other elements or other attributes but only simple types. It is also possible to reuse already existing elements. This is done through the `ref` attribute. The `ref` attribute makes it possible to use an existing element, rather than defining a new one.

```
<xsd:element ref="comment" minOccurs="0"/>
```

This declaration references an existing element, which was declared elsewhere in the schema. In general, the value of the `ref` attribute must reference a global element, meaning one that has been declared under the schema element rather than as part of a complex type definition. Because elements defined, for instance as part of a complex type, are local to this element. These scoping rules are similar to the scoping rules in object-oriented languages. Hence, global elements, and global attributes, are created by declarations that appear as the children of the schema element. Once declared, a global element or a global attribute can be referenced in one or more declarations using the `ref` attribute as described above. A declaration that references a global element enables the referenced element to appear in the instance document in the context of the referencing declaration. It is like defining a global attribute in a Java class.

In the example above the optional `minOccurs` attribute was introduced. XML Schema enables the definition of cardinality of elements. `MinOccurs` determines the minimum mandatory occurrences of an element. The maximum number of times an element may appear is determined by the value of a `maxOccurs` attribute in its declaration. This value may be a positive integer, or the term unbounded to indicate there is no maximum number of occurrences. The default value for both the `minOccurs` and the `maxOccurs` attributes is 1.

2.7.4 Derivation of Types to Create New Types

New simple types are defined by deriving them from existing simple types. This can be built-in types or derived simple types. The usual way to define a new simple type is by restricting an existing one. It is important to understand, that a derived type is handled like a subtype of the type defined in the restriction base. The notion of types and subtypes is lend from object-orientation. The consequence is, as shown later in this section, types can be substitute by its subtypes.

The elements `<simpleType>` and `<restriction>` are used for this purpose. The restriction element is used on an element to indicate the existing type, and to identify the so called facets that constrain the range of values. Facets are different kinds of restrictions that can be imposed on base types. To create a new type of integer called `myInteger` whose range of values is between 10 and 99, we base our definition on the built-in simple type integer, whose range of values also includes integers less than 10 and greater than 99. To define `myInteger`, we restrict the range of the integer base type by employing two facets called `minInclusive` and `maxInclusive`:

```
<xsd:simpleType name="myInteger">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="10"/>
    <xsd:maxInclusive value="99"/>
  </xsd:restriction>
</xsd:simpleType>
```

The following new type `ISBNType` is based on a string. The restriction facet used here is the pattern. The pattern allows the definition of regular expressions. In this case three different structural possibilities are provided. The three lines containing the pattern element can be seen as an "or" boolean expression. The first regular expression defines an ISBN number as a one digit followed by a dash, followed by a five digit number, followed by a dash, and so on. The other two regular expressions can be read in the same way.

```
<xsd:simpleType name="ISBNType">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="\d{1}-\d{5}-\d{3}-\d{1}"/>
        <xsd:pattern value="\d{1}-\d{3}-\d{5}-\d{1}"/>
        <xsd:pattern value="\d{1}-\d{2}-\d{6}-\d{1}"/>
    </xsd:restriction>
</xsd:simpleType>
```

XML Schema defines fifteen facets. Among these, the enumeration facet is shown in the next example. It is particularly useful because it can constrain the values of almost every simple type, except the boolean type. The enumeration facet limits a simple type to a set of distinct values.

```
<xsd:simpleType name="USState">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="AK"/>
    <xsd:enumeration value="AL"/>
    <xsd:enumeration value="AR"/>
    <!-- and so on ... -->
  </xsd:restriction>
</xsd:simpleType>
```

Only one value can be applied to this element in an instance document.

2.7.5 Documentation

One important approach for writing reusable schemas is to document them. XML Schema provides elements for human readable comments and processing instructions to handle for supporting tools. Human readable documentation can be defined in `<documentation>` elements, while information targeted at applications should be included in `<appinfo>` elements. Both elements must be included in an `<annotation>` element. They accept optional `<lang>` to indicate the language of the documentation. The source attribute is a URI reference that can be used to indicate the purpose of the `<appinfo>` to the processing application. These elements are defined in the XML Schema vocabulary.

2.7.6 Composing Schemas from Multiple Files

In object-orientation design, large and complex architectures are dealt with, by decomposing them into smaller, manageable software units, which are easier to handle. Further, it is common practice to reuse existing libraries. Following from that, there is a need in XML schemas to provide a mechanism to split up XML documents and to provide access to existing vocabularies. Hence, as schemas become larger, it is often desirable to divide their content among several schema documents. This is to reduce complexity and provide ease of maintenance. To split up a large schema or to use libraries of schemas XML Schema provides two mechanisms for including external schemas.

The first, `<include>`, is similar to a copy and paste of the definitions of the included schema. The demanded schema is only included. Hence, it is not possible to override any elements definitions of the included schema. The include mechanism enables you to use externally created schema components "as-is", that is, without any modification. It is used as follows:

```
<xsd:include schemaLocation="character.xsd"/>
```

Object-oriented languages like Java provide a similar mechanism to include other libraries. The Java language defines the import keyword to include packages containing Java classes. The classes inside the included packages can be used in the importing class. The second inclusion mechanism, `<redefine>`, is similar to include, except that it lets you redefine the declarations from the included schema.

```
<xsd:redefine schemaLocation="character12.xsd">
<xsd:simpleType name="nameType">
 <xsd:restriction base="xsd:string">
  <xsd:maxLength value="40"/>
 </xsd:restriction>
</xsd:simpleType>
</xsd:redefine>
```

Redefine shares the object-oriented notion of extending a class, thus providing a kind of inheritance for XML schema documents. It is important to notice that new defined types, based on an already existing type, are seen, in the scope of XML schemas, as subtypes of the derived types. For instance, XML parsers complying to the standard, are able to recognize a type as being a subtype of another type. Features like `<include>` and `<redefine>` are introduced to create libraries of schemas. These are features borrowed from object oriented design that can be used to create reusable schemas.

2.7.7 Substitution Groups

XML Schema provides a mechanism, called substitution groups, that allows elements to be used in place of other elements. To use this feature, elements have to be assigned to a special group of elements that are said to be substitutable for a particular named element called the head element. The head element can be substituted by all elements defining this specific head

element in their `substitutionGroup` attribute. One constraint is, that all elements in a substitution group must have the same type as the head element, or they can have a type that has been derived from the head element's type. It provides a mechanism for allowing elements to be used interchangeably. The substitution group is defined through referencing a common element, the head. The head element does not hold any specific declaration but must be global. Then they can all be used in place of the head element. In the following example the element "surname" can be used anywhere an element "name" has been defined. Substitution groups provide the flexibility to reflect the notion of types and subtypes in object-orientation.

```
<xsd:element name="name" type="xsd:string"/>
<xsd:element name="surname" type="xsd:string"
substitutionGroup="name" />
```

This concept is very important to provide the flexibility to map type substitutability from object-oriented languages. In object-orientation, objects adhering to the same type, can be used interchangeable. Late binding mechanisms resolve during run-time which object is actually going to be used.

2.7.8 Abstract and Final Types

In object-oriented languages an abstract class can be defined to force a developer to extend this class and to implement the missing parts. XML Schema introduces the abstract attribute to support the same functionality. The following example defines a generic "name-elt" element, which is the head of a substitution group. Because it is declared as abstract, analogously to abstract classes in object oriented languages, it can not be used directly. It has to be replaced by either name or surname everywhere it is referenced.

```
<xsd:element name="name-elt" type="xsd:string" abstract="true"/>
<xsd:element name="name" type="xsd:string" substitutionGroup="name-
elt" />
<xsd:element name="surname" type="xsd:string"
substitutionGroup="name-elt" />
```

The mechanism allows an element to change its type as long as it is a subtype of a specific abstract type. This is quite similar to the notion of types in object-orientation where an abstract class is defining the type of its subclasses. The derived classes can be used everywhere where their superclasses can be used, too.

After having shown a mechanism for defining abstract types, there might be the need to block the definition of subtypes. Where in Java the final keyword can be used to achieve this requirement, the XML Schema specification introduces the final attribute which can be used in an `<complexType>` or `<element>`. This attribute can take the values *restriction*, *extension* and *#all* to block derivation by restriction, extension or any derivation. The following example would, for instance, forbid any derivation of the `characterType` complex type.

```
<xsd:complexType name="characterType" final="#all">
```

The investigation of XML Schemas has shown, that they provide a powerful mechanism to represent object-oriented structures. The schema specification introduces plenty of object-based concepts, such as derivation of types, and substitutability, hence supporting the facilities needed to map class- or object graphs into an XML presentation. But, XML Schemas can not be seen as an object-oriented language approach. It is rather an XML serialization mechanism which is able to represent object-oriented constructs in a language independent way.

2.8 XML Namespaces

Understanding XML namespaces is essential to understanding and building reusable XML schemas. Namespaces have been a W3C standard since January 1999. The idea behind namespaces is to provide a mechanism to fully qualify XML element and attribute names. This is to prevent two elements having the same name but different meanings.

A schema contains a collection of type definitions and element declarations. These type definitions and element declarations are named. A schema has its own name, which is supposed to be unique. Hence, an element's qualified name can be built by adding the schema name to an element name. The schema name opens a unique namespace. The elements whose names belong to a particular namespace are called a target namespace. For instance, the qualified name of the `<complexType>` is `http://www.w3.org/2001/XMLSchema/complexType`. This is because the complexType element is residing in the `XMLSchema` namespace. But note, that a different `<complexType>` element can be defined in another namespace. The notion of namespaces provide a similar mechanism as packages in Java. Packages provide the naming scope for Java classes. The fully qualified name of a Java class consists of its package and its class name.

Target namespaces enable us to distinguish between definitions and declarations from different vocabularies. For example, target namespaces would enable us to distinguish between the declaration for an element in the XML Schema language vocabulary, and a declaration for an element in another user-defined vocabulary pertaining to a completely different context. The former is part of the `http://www.w3.org/2001/XMLSchema` target namespace, the latter is part of another target namespace, defined by the author.

For example, you might have an accounting application that uses an element called `<schedule>` to mean an accounting schedule. A time management application might use the same element `<schedule>` to mean a time schedule. Using both elements in an instance document can cause name ambiguities. The purpose might be to keep track of all project information including accounting and project timeline in the same document. Without additional information an application would not be able to distinguish between this two elements. Namespaces allow the provision of a fully qualified name which adds a prefix to each of its containing elements. For instance, each `<schedule>` element can be extended with a unique prefix that indicates to which application it belongs. For example `<accounting:schedule>` and `<time:schedule>` would make things unambiguous. In the

next example accounting and time are namespaces. Usually, the namespaces used in an XML schema can be added as attributes in the <schema> element.

```
<xsd:schema
     xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
        xmlns="http://example.org/ns/books/"
        targetNamespace="http://example.org/ns/books/"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified" >
```

Namespace declarations play an important role. The first namespace declaration in the XML schema above, is `xmlns:xsd="http://www.w3.org/2000/10/XMLSchema`. The `xmlns:xsd` prefix defines, that all elements pertaining to this namespace, have to be prefixed with `xsd` throughout the document. As already mentioned, this namespace contains all the elements which are part of the W3C XML Schema instructions. The name of the prefix used can be chosen arbitrarily.

If most of the elements in the document belong to the same namespace, it is not necessary to prefix each element name. Instead you can define a default namespace that applies to all non-prefixed elements and attributes. The syntax for defining a default namespace is `xmlns=”namespace”`. The defined default namespace in this case is `http://example.org/ns/books/`. This means elements belonging to this namespace are not prefixed. The target namespace defines the namespace in the current document. It enables to distinguish between definitions and declarations from different vocabularies. Further, it provides a unique name, which can be addressed by other schemas. These enables reuse of existing schemas.

2.8.1 Conclusion

XML Schemas with XML Namespaces provide support for building reusable, decomposed schemas, which can be organized as own vocabularies. As a encoding mechanism for SOAP, there is to date no comparable solution providing the a similar amount of functionality.

2.9 Critical look

SOAP has attracted a widespread corporate interest, which is reflected by the amount of tools emerged in the last few months. The promise of an remote procedure call and messaging mechanism being Internet-capable and enabling the integration of heterogeneous system platforms is causing interest in different kind of businesses. eBusiness solutions and business-to-business integration are needing a more flexible and standardized approach to consolidate their business. Naturally, companies providing SOAP tools, are tending to publish biased information on SOAP and marketing oriented statements. Therefore this section provides an objective view on SOAP which is a result of the investigation carried out on SOAP.

2.9.1 SOAP's problems

As described in the previous sections, SOAP is an XML messaging and RPC standard. As one benefit, it provides a framework for XML messaging without excessive complexity. This is mainly due to reusing already successful and widely spread technologies, which have already proved their ease of use, such as XML and HTTP. The SOAP specification is fairly concise compared with many other messaging standards, with much of the text dedicated to encoding. Because of its widespread industry support, simple design, and the use of SOAP in other XML standards, SOAP appears to be the XML messaging standard which is going to be used in the future. But the investigation has shown that SOAP does not fit in every environment. There are situations where a protocol like SOAP is not appropriate.

Two problems are occurring with SOAP which can be a hindrance to use SOAP in certain environments. The first problem is the performance of the serialization and deserialization mechanism. The second problem is the size of the message created by SOAP. Hence, the interesting issue emerging here is the performance of the text-based approach SOAP compared with binary protocols, such as CORBA, RMI, or DCOM.

With binary protocols, encoding arguments and return values is carried out using a binary data format. The client and server agree on the binary format, there is no need to add meta-information such as the names or types of the arguments. Whereas, SOAP has to provide quite a lot of additional information, such as the complete method signature with type information, as shown in the SOAP example previously. The advantage of binary protocols is that they achieve better performance, above all considering the serialization and deserialization time needed for their method arguments. But different binary encoding systems are hard to work together. In the case of CORBA's CDR and DCE's NDR format, which is used in DCOM, as already mentioned before, makes it impossible to work together. A text-based approach facilitates to process and to debug messages, because they are easier to read. A simple XML parser can be used to read or write a SOAP message. The negative side here is, that existing XML parsers are very expensive, in terms of code size, processing time and memory footprint. In the remainder of the section where the use of SOAP makes more sense, and where traditional middleware, such as CORBA and DCOM are the preferred mechanism to be used.

2.9.2 Conversion to a Network Representation

A serialization mechanism converts an object into a persistent state. SOAP uses XML as its serialization format. Deserialization converts objects from their persistent state to their representation in memory. Deserialization in SOAP involves parsing the XML representation of an object and instantiating the object using reflection. With binary protocols, such as Java RMI, the class structure of the object being deserialized is already known. The serialized code contains the information where to find the class, and provides the mechanism, in this case the class loader, to dynamically load the class information. On the other hand, in SOAP deserialization, the class structure is learned as the XML is parsed. This shows that SOAP's encoding scheme is limited in its functionality to provide a full object-based language. For

instance, SOAP does not define a means to represent remote references. The need to include class and type information coupled with the already large size of the XML representation of the serialized object makes the SOAP deserialization considerably less efficient.

The impact of this performance penalty might be a problem in a closed network environment where a fast server-to-server connection is needed. Taking into consideration the steps a SOAP call involves, a significant performance overhead would not be a surprise. SOAP message creation and parsing consists of several task of generating the envelope structure, filling it, verifying it, extracting the necessary parameters, and so on.
In an Internet environment, SOAP is revealing its main advantages, compared to binary formats. One advantage is that SOAP, being based on Internet standards, can pass firewalls easily. On the surface, it seems that an XML-based scheme would be intrinsically slower than that of a binary-based model, but it is not as straightforward as that. When SOAP is used for sending messages across the Internet, the time to encode/decode the messages at each endpoint is tiny compared with the time to transfer the bytes between endpoints, so using XML in this case is not significant.

Another question arising here is, if some parts of a SOAP call are representing an overhead, above all in an local network environment. SOAP needs additional meta-information about all types used in messages, using its default encoding rules. The sending of typing information in every message is an unnecessary overhead since this information is already implicit in the input and output schema definitions of the service, of which can be assumed the client is knowledgeable. SOAP implementations, like Microsoft's SOAP, are using a service description language to avoid to send additional information in SOAP calls. Apache SOAP has no support for a description language.

Further, including typing information within each message complicates the evolution of both clients and servers because changes in this information must be synchronously distributed and incorporated into their implementations.

2.9.3 Transmittable Data

Overall, SOAP provides many advantages. Unfortunately, its universality comes with a performance penalty. Because XML messages are textual the sizes of its messages are significantly larger than protocols which send binary data. In an environment, where large data sets have to be send as a part of a RPC call, the overall size of a SOAP call might compromise performance.

In [GOVIND] the performance relative to other communication protocols was tested. The throughput was compared for Sun RMI and Apache SOAP. The performance was compared to a transfer of serialized array and linked list data over a raw socket connection. In general SOAP is approximately ten times slower than Sun's implementation. Considering the relative sizes of data that must be sent for the same object, this comes as no surprise.

Further the investigation has shown that the size of serialized data types in SOAP is approximately ten times larger than in Sun native serialization. This increase in size is from the translation of binary data into text.

For example, in Java, each double takes 8 bytes. The string representation in XML of a double with 16 digits of precision takes at least 16 characters in addition to the 17 bytes for the tags `<double>` and `</double>`. Thus, each double serialized into XML could take at least 33 bytes. This results in an overhead that is at least a factor of four larger in the XML representation of a double array. Since SOAP uses XML for data representation, this overhead is intrinsic to the SOAP protocol and cannot be removed by choosing a better implementation.
Serializing Java objects into SOAP-encoded XML data takes approximately ten times more memory than the binary representation. Sun's native Java serialization-deserialization is closely tied to Java and hence providing a very efficient mechanism. Serialization and deserialization speeds for SOAP-based implementations are approximately 100 times slower and their throughputs are also a 100 times lower.

The most significant defect detected in [GOVIND] of using SOAP for RMI is performance. Sending the 8-byte double in XML, `<double>3.141592653589793E+000</double>`, requires 40 bytes of data. SOAP's data representation size in general is about 10 times the size of binary representations. Determining the precise performance penalty is important for deciding when SOAP is appropriate. Because the costs of serialization and deserialization of XML encoded messages is very high, the use of SOAP in a closed environment where server-to-server performance is critical, should not be considered yet. On the Internet, where the time spent for communication is much higher, SOAP is the preferable protocol to be used.

2.9.4 SOAP Interoperability

To date, over 50 SOAP toolkits have appeared. The challenge is to provide interoperability among the tools. SOAP's full value can only be realized if interoperability between the toolkits is ensured. Taking a look at the huge amount of marketing literature for SOAP issued since the first introduction of the specification, one of the main promises made is cross-platform interoperability, regardless of development platform or programming language. The companies are aware that interoperability is the driven force behind the Web Service technology, but examples like CORBA have shown, that companies tend to add proprietary mechanisms.

The challenge with the SOAP specification is, that it introduces many optional components, which may result in different interpretations. The ambiguity of the specification language can cause interoperability problems. The built-in flexibility of the SOAP specification yields implementations, which are SOAP compliant but may have made distinct assumptions on certain issues. Optional components are, for instance, it is optional if SOAP message parameters are provided type information directly inside the SOAP document, or if the type information is provided somewhere else. An implementation making the assumption that parameters are typed will not be able to process the untyped SOAP message. As [bridge] will

show, this is actually the problem with Apache SOAP and Microsoft SOAP. Further, a toolkit provider might decide to implement only a subset of the SOAP specification. There might be a mismatch in supported features amongst two different tools.

There is evidence that in the case of SOAP, the companies are committed to ensure interoperability. To help address this issue and to promote SOAP interoperability in general, the SOAPBuilders online group [ILAB] was founded. Group members are representants from large companies, such as Microsoft or IBM, to individual persons with their own implementation of SOAP. The group has introduced a mailing list to discuss interoperability issues, and has specified a test suite for testing interoperability. The SOAPBuilder Interoperability Lab (ILAB) is established to support the testing of SOAP implementations. The ILAB is using the test suite to investigate the interworking of tools and publishes the test results. This concerted effort shows that software industry has realized the importance of SOAP and that interoperability is a center-stage issue. The community work is a prove for the effort made in realizing the promise of SOAP toolkits from whatever provider to work seamlessly together. In the Part C The Web Service-to-Jini bridge chapter is it investigated if this effort was a success in the case of Apache SOAP and MS SOAP.

2.10 Conclusion

One of the big questions about using SOAP, or any XML-based mechanism for distributed computing is where it makes most sense. The investigation has shown that there will also be a place for other distributed technologies, such as DCOM, CORBA, and Java Remote Method Invocation (RMI). To many existing applications rely on these mechanism.

DCOM, CORBA and Java RMI are using binary protocols. As the last section has shown this means speedier transmission across the network and more instantaneous processing by the recipient. XML messages need more bandwidth and have to be run through a parser before processing.

However, it can be argued that those disadvantages are not so important, given the rapidly increasing speed of parsers and CPUs. The success of the Web, which is also based on relatively verbose protocols, can be taken as example. The Web has achieved a more widespread adoption than any competing network computing technology. Hence, the benefits of SOAP are, in the current situation, most obvious for applications that cross local network boundaries. Because it is not possible to control what technologies are going to be used outside of a network boundary.

3 The Web Service Description Language

This chapter describes the Web Service Description Language (WDSL). WSDL is a specification to describe Web Services and how to access them. The WSDL was submitted by Microsoft, IBM, and Ariba to the W3C, the standardization is still in process. The purpose is to provide a structured way to describe the communication protocols and message formats introduced by the Web Service model. This need is addressed by defining an XML grammar for describing the services and its exposed operations.

3.1 Introduction

A WSDL document entails the location, and the description of the operations it exposes. It provides a simple way for service providers to describe the basic format of requests to their systems regardless of the underlying protocol used, such as Simple Object Access Protocol and the data encoding used, such as XML Schema or Multipurpose Internet Messaging Extensions (MIME), in a standardised way. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an operation. Related concrete operations are combined into abstract services. WSDL is extensible and supports service descriptions and their messages regardless of what message formats or network protocols are used to communicate. Currently, the only bindings described in the WSDL specification are SOAP 1.1, HTTP GET/POST, and MIME. WSDL provides a way for service providers to describe the basic format of web service requests over different protocols or encoding. It describes what a Web service can do, where it resides, and how to invoke it.

In object-orientation, there is a generally understood notion of an interface. The interface is a concrete and immutable description of what the object looks like to external applications that are going to use the services of that object. In some languages, the interface takes the shape of an Interface Description Language (IDL). WSDL defines services as collections of network endpoints called ports in WSDL. The abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings. This allows the reuse of abstract definitions of messages, which are abstract descriptions of the data being exchanged, and port types, which are abstract collections of operations. The idea of describing a service in abstract form is similar to the use of an interface in an object-oriented context or CORBA IDL. The concrete protocol and data format specifications for a particular port type constitute a reusable binding. A port is defined by associating a network address with a reusable binding, a collection of ports define a service.

3.2 Current Implementations

Because WSDL, as with all the other technologies comprising the Web Service model, is a current technology, tools supporting WSDL are relatively uncommon. The most widely known ones are Microsoft.NET, part of the Microsoft .NET SDK, which is the most mature

and advanced tool set utilizing WSDL. The second one is the IBM Web Service Tool Kit (WSTK), which also provides tool support, but the implementation is less convincing compared to .NET. Microsoft .NET is actually in beta 2 phase, while the IBM tool has not passed the beta 1 phase. Nevertheless, .NET provides a complete implementation and tries to support development with Web Services and WSDL through automatic generation of WSDL-based code. IBM does not yet provide many features. WSTK implements only the SOAP binding, whereas in .NET HTTP POST/GET protocol can be used. WSDL can be used in an RPC- or message-based style. In its current state, WSTK is only RPC compliant. IBM is very committed to WSDL as being a party to the specification submission. Hence, it is only a question of time before IBM WSTK will add the missing features.

As mentioned above, a WSDL document conceptually divides the basic service description into two parts, which can be denoted as the service interface describing the abstract part of a Web Service, and a service implementation representing the binding to a concrete protocol and endpoint. This enables each part to be defined separately and independently, and reused by other parts. The specification leaves great freedom concerning how it can be interpreted by implementation parties. The consequence is, that in its current state, the .NET generated WSDL code and the WSTK one are not compatible. The reason is that the IBM tool reflects the separation of abstract and concrete description by creating two WSDL files. These two files are connected via the WSDL's import mechanism. Dot-NET's tool does not support the import statement yet. Further, the WSTK uses type definitions which are part of an older version of the XML Schema specification.

The next figure sketches the WSDL elements and categorizes them as being part of either the abstract or concrete description. Notice that the specification does not explicitly promote this separation. This is our view on how the distinction should be made. The WSTK makes the binding element part of the interface definition. The reasoning for this is not clear.

Service Interface Definition	**Service Implementation Definition**
Message	Service
Type	Port
PortType	Binding

A service interface definition is an abstract or reusable service definition that can be instantiated and referenced by multiple service implementation definitions. A service interface definition bears a close resemblance to an Interface Definition Language (IDL) or Java interface. This allows common standard service types to be defined, for instance for business entities, and implemented by multiple service implementers. This is analogous to defining an abstract interface in a programming language and having multiple concrete implementations. In CORBA, service interfaces have been defined by industry standards organizations, such as HL7 for the health industry. WSDL goes in the same direction.

The service interface contains WSDL elements that comprise the reusable, abstract portion of the service description. These are the *binding*, *portType*, *message* and *type* elements. The operations of the Web service are defined in the *portType* element. The operations define what XML messages can appear as input and output. An operation maps a method signature from a programming language into an XML representation. The message element specifies which XML data types constitute various parts of a message. It is used to define the input and output parameters of an operation. The use of complex data types within the message is described in the *types* element. Due to the usage of namespaces, user defined types can be imported from external files. The service implementation definition describes how a particular service interface is implemented by a given service provider. A Web service is modeled as a service element. A service element contains a collection of port elements. A port associates an endpoint, for example, a network address location or URL with a binding element from a service interface definition. The binding element describes the protocol, data format, security and other attributes for a particular service interface.

The service provider would develop a service implementation definition document that describes the WSDL service, port and address location elements that describe the network address of the provider's Web service and other implementation-specific details. The service interface definition together with the service implementation definition makes up a complete WSDL definition of the service. This pair contains sufficient information to describe to the service requestor how to invoke and interact with the Web service. The service requestor can require other information about the service provider's endpoint. This information is provided by the complete Web service description of the service.

A WSDL document is very verbose and contains all the information needed to access Web Service endpoints. In the bridge chapter we shall use Microsoft .NET generated WSDL documents to harness the provided meta-data to generate Java interfaces and proxies to prove its capabilities to provide a type-safe means to describe services.

3.3 Defining Web Services with WSDL

The following, very simple, Java class is used as an example to show what a WSDL document looks like. The Java class is going to be exposed as a Web Service. The service provides two operations.

```
public class Weather {
  public float getTemp(String city) {
    float temp = WeatherService.getTemp(city));
    return temp;
  }

  public void setTemp( String city, float temp ) {
    WeatherService.setTemp(city, temp);
  }
}
```

The class is going to be represented as a Web service and both methods are exposed as operations on that service. It is the decision of the developer which methods will be exposed. It should be noticed, that the development steps of a class which is going to be exposed as a Web Service, are the same as the steps involved in a local implementation.
The following parts present the WSDL document representing the above shown Java class. The <service> element defines a physical location for a communication end-point. It uses the port type and binding specified earlier, and basically gives the Web address or URI for a particular provider of the described service.

To describe this service, you use the WSDL <definitions> element. All WSDL elements belong to the WSDL namespace, which is defined as http://schemas.xmlsoap.org/wsdl/. As an example, consider a service that you call weatherService; the service would be defined using WSDL as follows:

```
<definitions name ='weatherservice'
 xmlns='http://schemas.xmlsoap.org/wsdl/'>
 <service name='WeatherService' >
 ……
 </service>
</definitions>
```

The <definitions> element is the root element of the WSDL document. The whole XML structure is enclosed in the <definitions> element that describes a set of related services. The WSDL namespace is declared as the default namespace for the document. So all elements belong to this namespace unless they have another namespace prefix. WSDL relies heavily on XML namespaces. The XML namespaces given in the <definitions> element's target namespace attribute is by default attached to all the names used for the other top-level WSDL elements. Developers can use qualified names to refer to these elements using prefixes from the particular namespace declarations in scope. This is the usual way to use namespaces in an XML document. The XML namespaces mechanism is used for the same purpose as in SOAP documents - that is for disambiguating names in the character data of XML specifications. XML namespaces are also used to connect WSDL elements to the data-typing provided in the <types> element. The <types> element allows the specification of low-level data-types for the message or procedure contents. Different mechanisms are permitted through namespace extensibility, but XML schemas are likely to be the choice for most users. WSDL provides a system for importing data-type specifications located as separate resources. The current implementations use XML Schemas as a mechanism to define types. There could be several such resources in cases of complex messages in multiple usage domains. In the current situation, we are not aware of a better means to provide XML datatype definitions. Since the last version change the XML Schema specification has been quite robust and provides a mature technology.

Each service is defined using a service element. Inside the service element, you specify the different ports on which this service is accessible. A port specifies the service address, for example, `http://localhost:8080/soap/rpcrouter` The port definition would be like this:

```
<port name='Weather_ServiceBinding' binding='Weather_Service' >
    <soap:address
       location=' http://localhost:8080/soap/rpcrouter ' />
</port>
```

Each port has a unique name and a binding attribute. The binding element is part of the service implementation definition and explained later in this section. When using SOAP as the communication protocol, the port element contains a `<soap:address/>` element with the actual service address. Here, the SOAP namespace prefix refers to the namespace `http://schemas.xmlsoap.org/wsdl/soap/`. This namespace is used for SOAP-specific elements within WSDL. Such elements are also known as WSDL SOAP extension elements. There are different SOAP specific elements used in this WSDL document, which will be introduced when necessary.

The communication protocol does not have to be SOAP. WSDL allows for different bindings, such as SOAP and HTTP. For instance, if your Web service is exposed via HTTP GET, the port element would contain an `<http:address/>` element similar to this:

```
<http:address
location="http://localhost/demos/wsdl/devxpert/weatherGET.asp"/>
```

A Web service may be accessible on many ports. For example, you might make your service available via SOAP and HTTP GET and possibly even via SMTP. For this Web service, you would have three ports each one with a different name. Microsoft .NET provides this flexibility. It is possible to add different port bindings, hence providing different access points for clients. The IBM WSTK is very restricted with respect to port binding definitions. They provide only one port for all deployed Web Services. This port hosts a process which dispatches Web Service requests to the specific implementing Java class/method. Further, as already mentioned above, only the SOAP binding is provided.

3.3.1 Defining Request and Response Messages

The next step is to define the service's request and response messages. A message is described in a protocol independent manner, that is to say, a message may be used with SOAP, HTTP GET, or any other protocol. To use Web services in a remote procedure call (RPC) model, two messages must be described. There is the input or request message, which is sent from the client to the service. The server then sends the output or response message. In the case of SOAP, the message refers to the payload of the SOAP request or response. That is, the message does not include the SOAP envelope, or the headers. In the WSDL specification there is no naming convention for messages specified. Messages can be named arbitrarily using their name attribute. Usually WSDL documents are generated automatically by a tool that will probably follow its own naming convention for messages.

The <message> element defines the data format of each individual transmission in the communication. In our example, this is a simple statement that the body of the message is a particular element from the schema in the types section. The breaking of a transmission into message parts depends on the logical view of the data. For instance, if the transmission is a remote procedure call, the message might be divided into multiple parts, one of which is the procedure name and meta-data and the rest being the procedure parameters. In an RPC call two message have to be provided.

To describe the message structures, you use the WSDL <message> element. Each <message> contains zero or more <part> elements. A <part> corresponds to a parameter or a return value in the RPC call. The request message will contain all in/out parameters and the response message will contain all out parameters as well as the return value if the service returns something. The part name order reflects the parameter order. Each <part> must have the same name and data type as the parameter it represents. This is a SOAP predefined naming rule which is part of the SOAP specification and not WSDL specific. For example, the getTemp method would correspond to two messages. A request message sent from client to service and a response message sent back to the client:

```
<message name='IngetTempRequest'>
  <part name='city' type='xsd:string'/>
</message>
<message name='OutgetTempResponse'>
  <part name='Result' type='xsd:float'/>
</message>
```

The naming of the parameters names and method names are not specified. That means, that different tools will generate different outputs. Notice that the data types are prefixed with the xsd namespace prefix, as it was declared earlier in the document. As already mentioned in a previous section, XML Schema defines many data types that can be used to define the message parts. The extensive list of XSD types is usually sufficient for all simple data types needed. However, if a service uses user defined types, those types have to be defined in WSDL. These new defined types have to be added in the <types> section.

3.3.2 Port Types and Operations

If using the RPC model, the messages have to be tied together to represent a request-response pair corresponding to the specific method call they are representing. For this purpose, WSDL provides the operation element. Operations are defined using the <operation> element. An operation specifies which message is the input and which message is the output like this:

```
<operation name='getTemp' parameterOrder='city'>
  <input message='IngetTempRequest' />
  <output message='OutgetTempResponse' />
</operation>
```

Inside the <operation> element <input> and <output> elements are defined. Each refers to the corresponding message by its fully qualified name, for instance IngetTempRequest. The

<operation> element groups messages that form a single logical operation. For instance, in our case, we can have an getTemp request which triggers an getTemp response, or in case of error or exception, a fault response. This particular exchange is grouped together into a WSDL port type. As you can see, the relationship to messages is made by qualified name reference. A port type element is the abstract representation of all methods of a class which are going to be exposed as part of a Web Service. The collection of all operations exposed by your service is called a portType and is defined using the WSDL <portType> element like this:

```
<portType name='WeatherSoapPort'>
  <operation name='getTemp'>
    <input message='IngetTempResponse' />
    <output message='OutgetTempResponse' />
  </operation>
      <!-- other operations would go here -->
</portType>
```

So the <operation> element is a child of <portType>. The portTypes name can be chosen arbitrarily. There are four forms of operations with built-in support in WSDL. These are one-way, request-response, solicit-response, and notification. The latter two are simply the "inverse" of the first two, the only difference being whether the end point in question is on the receiving or sending end of the initial message. Basically, WSDL supports unidirectional and bi-directional, such as request-response port types. Faults are only supported in the bi-directional port types.

3.3.3 Transition from Abstract to Concrete

The WSDL document moves now from the abstract and logical, represented through messages and port types, to the concrete and physical, with some reference between the two. The <binding> element content is the connection between the logical and physical model. A binding definition takes the operation defined through the abstract port type and connects it to a concrete description of how it is transmitted through SOAP. Here is where the earlier mentioned SOAP extensions to WSDL are used. For instance, SOAP needs to know how the data is going to be encoded. The SOAP specification contains predefined rules for encoding which must be added to the WSDL document. WSDL also provides bindings to HTTP and MIME, and full extensibility to other protocols.

The example binding specifies the Weather_ServiceBinding as having the SOAP RPC style as the transport mechanism. The style can be RPC or Document, the former indicating a more procedural communication pattern and the latter is a message-based model. Further, the binding also specifies the network transport as HTTP. SOAP can be transmitted by other means, such as SMTP. The <soap:operation> elements maps the individual messages in the port type to definition of SOAP endpoints running on the server side. It is also important to define a SoapAction parameter, required for SOAP over HTTP. The given value must be used in the HTTP headers of the actual messages in order to be identifiable as SOAP messages.

The following code extract shows the transition occurring from abstract data types, messages, and operations to concrete physical representation of messages bound to a transport protocol. To define the concrete aspects of operations, the WSDL <binding> element is used:

```
<binding name='Weather_ServiceBinding' type='Weather_Service' >
    …
</binding>
```

The name of the binding can be chosen arbitrarily. However, the standard defines that the same name as for the binding attribute on the <port> element has to be used. The <binding> element contains a WSDL SOAP extension element called <soap:binding> which is used to specify the transport protocol. As already mentioned above, SOAP can be used over HTTP, SMTP, or possibly other transport bindings will be defined soon. For example:

```
<soap:binding style='rpc'
              transport='http://schemas.xmlsoap.org/soap/http' />
```

Then for each operation that this service exposes, the value of the SOAPAction HTTP header is specified. The SOAPAction is an HTTP header that the client sends when it invokes the service. The SOAP server uses this header to determine the service. The SOAPAction value is specified like this:

```
  <binding name='Weather_ServiceBinding' type='Weather_Service' >
    <soap:binding style='rpc'
transport='http://schemas.xmlsoap.org/soap/http' />
    <operation name='getTemp' >
      <soap:operation
soapAction='urn:weather_service' />
                      ....
    </operation>
  </binding>
```

Basically, an <operation> element is added with the same name as the operation you defined earlier. Within this <operation> you add a <soap:operation> with the soapAction attribute. Finally, the encoding of the input and output messages of this operation is attached, the complete binding looks like this:

```
  <binding name='Weather_ServiceBinding'
type='wsdlns:Weather_Service' >
    <soap:binding style='rpc'
transport='http://schemas.xmlsoap.org/soap/http' />
    <operation name='geTemp' >
      <soap:operation
soapAction='urn:weather_service' />
      <input>
        <soap:body use='encoded' namespace='urn:weather_service'

encodingStyle='http://schemas.xmlsoap.org/soap/encoding/' />
      </input>
      <output>
        <soap:body use='encoded' namespace='urn:weather_service'
```

```
encodingStyle='http://schemas.xmlsoap.org/soap/encoding/' />
      </output>
    </operation>
  </binding>
```

Within the <operation> you add an <input> and an <output> element and use a <soap:body> element within each to specify how the data is encoded. The URI http://schemas.xmlsoap.org/soap/encoding/ indicates the SOAP encoding style as described in the SOAP 1.1 specification.

Overall this example is quite simple. It describes communication consisting of short SOAP transmissions, with one input string and one output in each operation. WSDL could just as easily define multiple port types consisting of a myriad of messages.

3.4 Conclusion

WSDL is a very important building-block in the realm of Web Services. Above all, the combination with an XML messaging protocol, such as SOAP, makes it an indispensable mechanism for providing run-time information of services. Without a description language like WSDL, Web Services would loose their dynamic nature. Because Web Services are based on the idea of run-time binding of components there is the requirement of an standardized description language.

Using the SOAP protocol without the means of publishing a service description, a client-service interaction would have been carried out statically. The client needs to know the address of the service and its unique identifier. Further, to construct the SOAP call, the client needs to know the signature of the operation to be invoked. With WSDL, a service client can discover a Web Service dynamically.

Moreover, WSDL is an important part in ensuring SOAP interoperability. For instance, the Web Service-to-Jini bridge built in the scope of this thesis can dynamically generate SOAP calls using Apache SOAP. The Apache SOAP client is calling Web Services using Microsoft SOAP. The bridge is using the metadata provided in the WSDL file.

The interoperability problem between the two SOAP implementation is due to the fact that Apache SOAP is building its SOAP calls and response, providing type information for all parameters used in an operation. In contrast, Microsoft SOAP is not adding type information to its generated SOAP constructs. Rather, the type information provided on the WSDL document generated automatically, when implementing a Web Service. Hence, this implementation is tightly connected to WSDL. Apache SOAP does not support WSDL, and hence has to provide the type information, resulting in larger documents. A further investigation of this issue will be provided later.

The WSDL specification succeeds in keeping the language quite simple. Like the other technologies involved with the Web Service model, it is easy to use. Because WSDL

documents contains a thorough description of interfaces to Web Services they can be used to automatically generate access code to the services. A developer of Web Services does not have to be concerned about how to write or consume Web Service descriptions, there will be provided tools for doing that.

WSDL is a good example for how companies, in this case Microsoft, IBM, and Ariba, can combine their effort and submit a standard, without trying to compromise the standard. The WSDL specification borrows thoughtfully from other efforts trying to reuse already existing technologies, like the Service Description Language (SDL) defined by Microsoft.

Part B The Road to Jini

4 Introduction

This chapter introduces the Jini model. It will examine Jini's approach to a service-based architecture. The purpose is to compare the Web Service model and the Jini model throughout this chapter. Though, both models have from an architectural and conceptual point of view a lot of concepts and mechanisms in common, they are based on two different paradigms. Jini is a Java-centric model, hence tightly coupled to an object-oriented environment, whereas the Web Service model is built on an document or text-based approach, adhering to open standards. The idea is to describe Jini and its functionality, so that it allows to compare to the Web Service model.

4.1 Jini's View on Networking

This section describes the driving forces behind Jini. These forces are fundamental for understanding the Jini architecture and infrastructure which are explained in a later section. The analysis below elaborates why Jini is not only another distributed environment, but rather introduces a new kind of distributed paradigm.

4.1.1 Jini's Approach to Distributed Computing

Jini's vision is "to turn the network into the client's computer by supplying the client with a federation of remote "plug and play" devices and services in a dynamic configuration that is personalized for each client" [WALDO2]. The principle of "plug and play" devices and services yields to a system of spontaneous and decentralized networking. Such distributed systems require a new way of thinking in contrast to centralized systems that promote a more static networking model. Programs built according to the Jini idea are provided the means to function and survive in highly dynamic network environments. This entails the ability to adapt their behavior to the requirements of the current context.

Jini assumes the network is unreliable. Building a system on realistic expectations of the infrastructure is a straightforward concept to explain to developers but may cause doubts to customers who do not completely understand the ideas behind this approach. The creators of Jini argue that the problem with current system architectures, which are built on a distributed client/server model, is that they are often built on the assumption that network failure is not taken into consideration resulting in unreliable systems. The result is, if one system breaks, a chain reaction is caused affecting other systems, too. [WALDO]

Moreover, the Jini creators see the technology progressing towards a system, where each participant, can communicate with other participants in a network, and discover dynamically services. This decentralized organization of the network requires a design that allows updates and changes to individual components without the requirement to shutdown the

whole network. Unlike a single machine, a large network cannot be shut down without great difficulty, and updating the entire network is more difficult still. So the Jini system allows upgrades and updates to be installed and used by the components being networked without requiring that the network be shut down or all individual components to be updated. In a client/server system, when a server has to be changed or taken from the network, the whole system will stop to work.

Hence, Jini is built on the assumption that the components which are part of the system are changing, as are their interactions. Moreover, a part of the Jini vision is that everybody can be part of a Jini network, resulting in Jini being a very open system allowing a lot of different devices and services to participate. Anything with a processor, some memory, and a network connection can be part of the Jini environment. That encompasses devices like printers and other peripherals, and the community of wireless devices, becoming more important, such as cell phones.

The programming model introduced reflects another credo of the Jini vision. It is believed that it is difficult to develop large pieces of reliable software. The suggestion is instead to work on making small pieces of software reliable and then assemble them into a system that is resilient when one of the smaller pieces fails. "In this way we can build reliable systems out of non-reliable parts" [STEIN]. Jini allows for this approach.

The Jini infrastructure, combined with the Java environment's ability to move code safely, allows the system to represent this kind of spontaneous networking. Services and clients can join or leave a network federation anytime. More important, new and enhanced services can be introduced to extend the functionality of the networked federation.

4.1.2 Federation and Centralized Control

Jini introduces the notion of a federated systems in contrast to the commonly used centralized system, such as the client/server approach. A Jini federation entails, broadly spoken, all the services registered with a particular set of lookup services and the clients using these lookup services to find registered services. The term federation was chosen consciously, because it denotes a system that empowers the clients to be self-dependent and self-contained. There is no central entity on which the clients are dependent. That means that the purpose of the Jini system is to provide a minimal set of rules to allow clients and services to find each other and interact. The Jini system imposes only a small set of conventions to make this kind of loosely coupled model possible.

The mechanisms a Jini federation provides to achieve this aim are the lookup service and the discovery protocol. These mechanisms dictate how its members join, leave, and find one another. Also by dictating the use of the Java language type system, the Jini federation defines how services are identified. The Jini programming model codifies certain common styles of object interaction. Finally, by requiring that the proxy code for a service be downloadable, the Jini federation dictates how the services and their clients manage change in their implementations and the way they are extended. The usage of a federated model

instead of a usual centralized model reflects the ideas expressed in the Jini vision. Using a federated model instead of the usual model of centralized control, as in distributed operating systems, was a conscious decision by the Jini designers. While centralized systems can be optimized for some cases, changing them over time is difficult. More importantly, centralized systems do not scale well, and as Jini federations begin to emerge, they have to scale to very large numbers. The federated approach allows such scaling in ways not available to centralized systems. Jini's architectural model supports peer-to-peer communication with variable size clients and variable size services.

4.1.3 What Makes Jini Different

The Jini vision entails consequences which are mapped to its architecture and programming model. One important point is that, as already mentioned above, Jini is built on the assumption that the network is unreliable.

As investigated in [WALDO], distributed computing is fundamentally different from non-distributed computing. The usually mentioned four differences include latency, memory access, partial failures, and concurrency. Most of the current distributed systems, such as CORBA or DCOM, attempt to hide some of these problems, whereas the Jini approach is aware of the possibility of failures. Rather, it aims at preventing these failures by providing the necessary tools and methods for effectively building software that adequately addresses these differences and is able to survive in the face of problems caused by distribution. The problem will become even worse considering the future changes in networking Jini is envisioning. The involvement of ad hoc networks and other loosely coupled systems that are distributed and decentralized, put a heavy burden on the distributed models existing today. The lack of centralized control implies that services be created on the fly without intervention of a central point of control.

In such a federated system the client has the capabilities to configure and manage itself, without human intervention. The services can automatically enter the network and communicate with other devices or services. These networks are also self-healing in that devices that leave the network for any reason do not affect the remaining devices' operation. A Jini client that loses contact with a server can recover and continue processing. This notion of self-healing is a very important part of the Jini system.

The reliability of a server process is another important factor in distributed computing. Clients are dependent on the functionality a server exposes. Reliability measures how well a device or network performs in the presence of disturbances. It is quite common that a process may start and stop, or crash completely. Because Jini is aware of this kind of problem the Jini technology can handle these failures because it expects devices to randomly move in and out of the network.

The Jini architecture parallels the way machines communicate over the Internet. On the Internet, any two machines will have multiple communication routes between them, and the Internet hides the exact path from the user. If a problem occurs with one path, Internet

routing protocols select and use another path. When a server becomes unavailable, it disrupts the client communication with that server process. When this happens in a Jini-based system, the client automatically goes looking for an alternate server. Once it locates another server process, the Jini client can reconnect. If no server is available, the Jini client waits or informs the user. This functionality is built into Jini technology and remains transparent to users.

4.2 Architectural Overview

This section does not provide a programming tutorial. Rather, an overview is given, focusing on the architectural aspects and technical requirements. The constituent parts of Jini are introduced. This introduction provides a knowledge base for better understanding the following sections. Further, some Jini specific terms are explained.

The Jini technology infrastructure is built around the notion of services. Services are entities that provide some specific functionality, which can be looked up by clients. A service in the Jini architecture is required to be defined in terms of a data type for the Java programming language. A data type can be implemented in various ways. Moreover, a service can represent an implementation of different types, hence providing a variety of functionality to clients. Types are a standard concept in object-oriented software techniques. Jini brings these object-oriented techniques seamlessly onto the network. Other distributed object technologies, such as RMI or CORBA, promise a similar object-oriented approach for distributed systems. We will show that Jini goes beyond these approaches.

Programmatically services are defined through a Java interface. Clients of a service need only know this interface to use it. An implementation of such an interface is called *proxy* in Jini parlance. Implementation details of the service are hidden completely from the client. Most of the network related issues are handled, transparently to the client, by the Jini technology infrastructure. Obviously, there must be at some point a way where the Jini client has to make a call to a network object, but we will see that in Jini this need is much less pervasive than in other current-day middleware.

Jini technology is built upon the Java platform. The most important point here is that the Java platform enables dynamic loading of code written in Java, that is sourcecode, as well as bytecodes. The Jini system does not only establish a remote procedure call mechanism, like in other client-server strategies, such as RMI or DCOM. Jini goes beyond simply moving messages by moving objects through the wire. A client of a Jini service receives a proxy, which represents an object with data and behavior. The proxy, which is downloaded into the client, makes the interaction with the service possible. Proxies are "real" implementations of the service interface, thus supporting all the Java language features a local implementation would provide. Clients holding such proxies can use the Java programming language's reflection capability, for example, to inspect the object. Proxies hide the implementation details of the protocol used for client-service communication and provide a simple method-call interface to the remote service, simplifying considerably the work necessary for a service's integration into a client.

To organize interactions between clients and services lookup services are necessary. The Jini lookup service is the central component of Jini's runtime infrastructure. Locating Jini services is carried out by using services provided by the Jini infrastructure. It enables service providers to publish their services and enables clients to locate and use services. Services are required to register with all available lookup services, and clients should utilize all available lookup services for their search. The Jini specification dictates this because running multiple lookup services provides the system with redundancy and some resilience against failure. As already mentioned in a prior section the notion of failure is a natural part of Jini. Jini's runtime environment is built around the awareness that partial failure is quite common in a network environment.

To summarize the core ideas, the Jini system exploits the Java environment to construct federations of services that can be accessed by clients. The definition of services is carried out through Java interfaces. The implementing objects are called proxies. This service-specific implementation needs to be written in the Java programming language, or at least the Java bytecodes are necessary. The service client will only know the interface of the service, hence being independent from implementation issues. Further, the Java environment's capability to move objects across distributed Java Virtual Machines (JVM) is used to a great extent. Proxies will be uploaded into a lookup service by a service provider. Service implementations will then be downloaded into a requesting client. This short introduction to some of the important parts of Jini, serves as starting point to a more detailed look at Jini.

4.3 Components Making Up the System

This section describes the different components making up the system. The focus of the system is to make the network a more dynamic entity that better reflects the dynamic nature of the federation by enabling the ability to add and delete services flexibly, hence supporting an ad hoc network. The technologies to support this idea of spontaneous networking are part of the Jini infrastructure and programming model.

The Jini specification defines a set of components that are supposed to built up the system. These components are the building blocks that constitute Jini's vision of a distributed environment. These components can be segmented into three categories. The first component is the infrastructure for federating services in a distributed system. Second, the programming model that supports and enables the production of distributed services. Finally there is the service component that can be made part of a federated Jini system and offers functionality to any other member of the federation.

The infrastructure provides the means to publish reliable services in Jini, hence making them available to clients. An important part of the Jini infrastructure is the discovery and join protocol. These protocols allow services to publish themselves, and enables clients and services to discover other services. Another major part of the infrastructure is the lookup service, which serves as a repository of services. The lookup service acts as a global storage mechanism for "Jini objects", thus working as a resource manager for service objects. These objects represent the proxies to the service that placed the code into the lookup service. They

can be downloaded to a client as part of a lookup operation. The notion of reliability requires an infrastructure that can assure that only safe code is transmitted to clients. Therefore the security system of the Java environment is extended to a distributed security system. This security system is integrated into RMI, thus extending the Java platform's security model to encompass distributed systems. The mechanisms introduced here can be seen as the backbone of the Jini system. In a later section the security mechanisms are explained in more detail and examined if they are sufficient for an Jini environment.

The programming model represents another component in the Jini system. The model provides a programmatic access to the Jini environment functionality. The Jini system is, compared to the base Java environment, enhanced with different concepts and mechanisms. They are mainly introduced to constitute the aforementioned vision of a flexible, distributed system. One enhancement is the event and notification interface, which is an extension of the Java Beans event model, to the distributed environment that enables event-based communication between Jini services. Further, services can use functionality provided in the leasing interface, which defines a way of allocating and freeing resources using a model which is based on duration. The leasing concept is very important in the scope of the self-healing approach. Moreover, it also provides a transaction interface to ensure data integrity.

The third category comprises services. Services, as already mentioned above, appear programmatically as objects written in the Java programming language, perhaps made up of other objects. A service has defined a contract which describes the operations that can be requested by an application or another service. The type of the service determines the interfaces that make up that service and also define the set of methods that can be used to access the service. A single service may be composed by using other services to achieve a specific task.

The Jini specification defines these three different categories of component for the Jini system. They all can be seen as the building blocks to realize Jini's vision of an ad hoc network. Some of the key concepts that are part of the three components mentioned above, are described in more detail in the following:

- *Service.* The most important concept within the Jini architecture is that of a *service*. A service is an entity that can be used by a variety of other entities, such as another service or a client program. A Jini system is not built on the assumption that there are sets of clients and servers communicating. Instead, a Jini system consists of services that can be composed dynamically to perform a particular task. Services may make use of other services, and a client of one service may itself be a service with clients of its own. The dynamic nature of a Jini system enables services to be added or withdrawn from a federation at any time, yielding in an ad hoc networking approach. Jini systems provide mechanisms for service construction, lookup, communication, and use in a distributed system. Services in a Jini system communicate with each through a predefined *service protocol*. This protocol is represented as a set of interfaces written in the Java programming language. The base Jini system defines other protocols that define other important service interactions.

- *Lookup Service.* Services are found and resolved by a *lookup service*. The lookup service is the central bootstrapping mechanism for the system and provides the major point of contact between the system and users of the system. A service client needs to get access to a lookup service, in order to find an appropriate service. A lookup service maps interfaces indicating the functionality provided by a service to sets of objects that implement the service, hence acting like a method call broker. In addition, descriptive entries associated with a service allow more fine-grained selection of services. Of course, references to a Jini Lookup service may be placed in these other naming and directory services, providing a means for clients of those services to gain access to a Jini system. A service is added to a lookup service by a pair of protocols called *discovery* and *join.* The discovery protocol enables to find the lookup service, and the *join* protocol provides the functionality to join a lookup service.
- *Java Remote Method Invocation (RMI).* Communication between services can be accomplished using Java RMI[SUNRMI]. The infrastructure to support communication through RMI is a part of the Jini technology infrastructure. RMI is a distributed mechanism that provides the functionality to find, activate, and garbage collect remote objects groups. Fundamentally, RMI is a Java-programming-language-enabled extension to traditional remote procedure call mechanisms. The RMI system not only allows data to be passed through the wire but full objects, including code. Much of the simplicity of the Jini system is enabled by this ability to move code around the network in a form that is encapsulated as an object.
- *Security.* The design of the security model for Jini technology is built on RMI security extensions. RMI, being an RPC system does not provide the right mechanisms to ensure security in an Jini environment.
- *Leasing.* A lease-based model is introduced to manage access to services. A lease empowers a client to access a service over a certain time period. Each lease is negotiated between the user of the service and the provider of the service as part of the service protocol. A client can prevent the expiration of a lease by renewing it, if he wants to use the service after the lease expires. If the is not renewed in time the resource is being freed. There can be different reasons for not extending a lease, perhaps the resource is no longer needed, the client or network fails, or the lease is not permitted to be renewed. A lease can be defined as exclusive to insure that no one else may take a lease on the resource during the period of the lease. Non-exclusive leases allow multiple users to share a resource. Leases can be seen as the resource manager of the system. They take care that unused resources are freed and returned to the resource pool.
- *Transactions.* A series of operations, either within a single service or spanning multiple services, can be wrapped in a transaction. The Jini Transaction interfaces supply a service protocol needed to coordinate a two-phase commit. How transactions are implemented—and indeed, the very semantics of the notion of a transaction is left up to the service using the interfaces.
- *Events.* The Jini architecture supports distributed events. An object may allow other objects to register interest in events in the object and receive a notification of the occurrence of such an event. This enables distributed event-based programs to be written

with a variety of reliability and scalability guarantees. A service can leave an event object on another service. If this service subsequently encounters a situation the service was interested in, the client will be notified.

4.4 Fallacies of the Network and Self-Healing Approach.

What makes Jini fundamentally different is that it is built on the assumption that networks are not reliable and partial failures can occur. Distributed systems are different from non-distributed systems. There are situations in which different parts of a cooperating group are unable to communicate, either because the connection between the members in the group has failed or because one of the members of the group has crashed. This partial failure can happen at any time and can be intermittent or long-lasting.

The possibility of partial failure makes the construction of distributed systems difficult and error-prone. In such a system components of the system provide resources or other services to each other. In many of the non-distributed systems resources and services are granted until explicitly freed or given up, others, such as Java, provide automatic memory management mechanisms. Systems, which are not providing such mechanisms are open to failures if there is no means provided to successfully make the explicit calls that cancel the use of the resource or system. This kind of failure can result in resources never being freed, in services being delivered long after the recipient of the service has forgotten that the service was requested, and in unnecessary high resource consumption.

To avoid these problems, as already mentioned in the previous section, the notion of a lease is introduced. From the fault tolerance point of view, the concept of leases is perhaps the most important concept of the ones promoted in Jini. A lease represents a dynamic reservation of a remote resource. By adding the lease approach into the Jini specification, the Jini system establishes itself as a self-healing system. Rather than granting services or resources until that grant has been explicitly cancelled by the party to which the grant was made, a leased resource or service grant is time based. When the time for the lease has expired, the service ends and the resource is freed, hence preventing unnecessary resource consumption. The time period for the lease is negotiated between the client and the service when the lease is first granted. A request/response form of communication between the client and the service is established. Leases may be renewed or cancelled before they expire by the holder of the lease, but in the case of no action taken by the lease holder, which includes the case of a network or participant failure, the lease simply expires. When a lease expires, both the holder of the lease and the grantor of the lease know that the service or resource has been reclaimed.

Although the notion of a lease was originally brought into the system as a way of dealing with partial failure, the technique is also useful for dealing with another problem faced by distributed systems. Distributed systems tend to be long-lived. In addition, since distributed systems are often providing resources that are shared by numerous clients in an uncoordinated fashion, such systems are much more difficult to shut down for maintenance purposes than systems that reside on a single machine. Hence, if a system has to be shut

down because of a maintenance issue, the lease-based approach enables a much easier way to shut down the system temporarily.

Like other service distribution architectures, Jini has the concept of events. However, in Jini a registration for receiving events is considered to reserve resources at the event source. Consequently, acting according to the Jini resource reservation principle, the event source creates a lease that is given to the event receiver. In this way Jini assures that the event source will eventually stop sending events to parties that have crashed or gone away.

As Waldo argues in [WALDO], most distributed computing approaches have aimed at simulating centralized systems by trying to "hide" faults and errors caused by the underlying network. For example, a typical distributed system, such as DCOM or CORBA, attempts to recover from communication errors by automatically resending messages and retrying the requested operation a number of times. Usually this happens below the programming API, and the situation is reported back to the application only after the recovery has failed. Furthermore, in such situations it is not necessarily clear what the state of the communicating peer is nor whether it has detected the network disruption at all.

4.5 Java's Language Environment as Requirement

The Jini system relies heavily on the Java environment. The system is Java technology-centered, assuming and requiring that the communication between a client and services used by that client are accomplished through a Java interface. This is because Jini builds on the existing Java environment and because it requires features that are available only with the Java environment. The Jini architecture simplicity and functionality, is based on the assumption that the Java programming language is the implementation language for components. The portable source code, the ability to dynamically download and run code, and the object-based approach are central to a number of the features of the Jini architecture. This tight coupling with the Java environment is subject to most of the misunderstandings concerning Jini. The common understanding is, that Jini is tied to the Java programming language, and the communication has to be carried out using Java RMI. The following sections will show, that Jini provides a much more flexible system.

4.6 The Importance of the Interface Type

The type of a service determines the interfaces that make up that service and also defines the set of methods that can be used to access the service. The type represents the contract between the service and its environment. A single service may be implemented by using other services. A client locates an appropriate service by its type, that is, by its interface written in the Java programming language. Additionally, to describe a service and facilitate the lookup, attributes can be added which provide more information on the service. Programmatic interfaces are identified by the type system of the Java programming language, and services can be found in a lookup service by asking for those that support a particular interface. Finding a service this way ensures that the program looking for the

service will know how to use that service, because that use is defined by the set of methods that are defined by the type.

To ensure that downloaded code from other machines is not malicious, the environment has built-in security, to allow the confidence to run code downloaded from another machine. Strong typing in the Java application environment enables identifying the class of an object to be run on a virtual machine even when the object did not originate on that machine. The result is a system in which the network supports a flexible configuration of objects which can move from place to place as needed and can call any part of the network to perform operations. The Jini architecture exploits these characteristics of the Java application environment to simplify the construction of a distributed system. The Jini architecture adds mechanisms that allow for mobility and management of all components in a distributed system, extending the easy movement of objects around the entire networked system.

4.7 Mobile Code

The Jini system and its flexibility is based on Java's mobile code ability which allows it not only to Transmit remote references, but "real" objects, entailing behavior and state. In order to understand this important mechanism of the Java environment some technical background is presented.

The Java Virtual Machine's (JVM) job is to execute Java bytecodes. Bytecode is stored in Java class files, which are loaded into the JVM via a *class loader*. The class loader mechanism is the part of the JVM implementation which takes care of finding and loading types. Loading a class means locating a class file that contains the desired type, based on the type's name, and then creating the class from that file [SUNJVM]. Once a class is loaded into a VM, it is linked into the VM's execution state, which means that it becomes part of the program's execution. Finally, the VM initializes the class by calling a special initialization method, which essentially corresponds to static initialization of the class.

Another crucial part concerning Java's code mobility capabilities is the Java *object serialization* mechanism [SUNSERIAL]. The JVM is able not only to load classes, but also to load objects from the network by employing the Java object serialization facility together with the class load mechanism. Object serialization maps a Java runtime object into a stream in such a way that a Java program can reconstruct the object at a later time. The serialized form must contain enough information to be able to identify and verify the Java class from which the contents of the object were saved and to restore the contents to a new instance. The binary stream representing the serialized object can be made persistent by storing it in, for instance, in a file or in a database management system.

The combination of these two mechanisms, the class loader and object serialization, enables the capability to dynamically download object instances from JVMs across the network. The JVM's class loaders are able to find and load classes from their codebases, if the class name is known. And, in addition, using object serialization, objects can be written to a stream and

transported across VMs. Object streams contain instance data to recreate the object as well as a descriptor of the object's class.

To make the dynamic downloading of objects across distributed JVMs possible, there is a requirement to tell the VM where it can locate the object's class . The strategy used is to add the code location for the class onto the serialized object stream. This is done by annotating the serialized object with the codebase URL . This method facilitates dynamic code mobility, because the VM can decide at runtime where it should download the classes from. This is also the fundamental technique used by Java RMI [SUNRMI]. Jini uses exactly this method to down- and upload service proxies to the lookup service and to clients. The key is to specify the correct service codebase, and then to make sure that you can actually download the classfiles needed from these specified locations. Specifically, what is happening is that the runtime system is annotating the correct codebase to serialized objects inside so called *`Service Item`s* before a service registers with lookup services.

The conclusion is that object mobility on the network is the cornerstone of Jini technology. Jini exploits the JVM's ability to dynamically download and link code from anywhere on the network, and takes advantage of object serialization to transport objects between Java VMs in a distributed Jini federation. To allow both objects and classfiles to independently traverse the network, Jini builds on the codebase annotation technology employed by Java RMI. This is achieved by annotating the codebase URLs for an object's class to the object's serialized stream. It is these techniques together that requires a client to have only a well-known Java language interface type locally available. The Jini discovery protocols can find lookup services and retrieve any object that guarantees the contract specified by the service interface. Classes required to interact with the service are then dynamically downloaded to the client from the service's codebase URLs.

4.8 Architecture Independence

The idea of providing services accessible through a lookup server, being delivered on demand to a client, leads to a very flexible architecture. The ability to move objects and code from the service provider to the lookup service and from there to the client of the service gives the service provider great freedom in the communication patterns between the service and its clients. This flexibility is in contrast to other middleware approaches, where the client/server relationship represents a static collaboration and the client receives the stub from a directly accessed server.

Since the server provides the stub code to the client on the fly and on demand, that code can change. In particular, an RMI-based server can implement an extension of a previously supported interface without there needing to be any change on the part of the clients. The extension means there is new stub code for the client to use, but the client will receive that new stub code the next time it receives a reference to the service. This allows the service to change, and the clients to automatically update themselves on an as-needed basis. No central coordination is needed to update clients. This code movement also ensures that the service object held by the client and the service for which it is a proxy are always synchronized,

because the service object is supplied by the service itself. The client only knows that it is dealing with an implementation of an interface written in the Java programming language, so the code that implements the interface can do whatever is needed to provide the service. Because this code came originally from the service itself, the code can take advantage of implementation details of the service known only to the code.

Putting these features together, a Jini service provider has great freedom in implementing an interface. A service interface may be implemented, for instance, by providing to the client an RMI reference to the remote object that implements the service. Or, the proxy could act as a local computation that provide all of the service locally to the client. Further, a combination of these two approaches can be used, implementing some of the functions of a service locally and the remainder through remote calls to a centralized implementation of the service.

Following from that, one architectural choice could be to move all the service code to the client, hence the service is running entirely within the client. The server's job is to expose the service to clients. This entails the registration of the proxy with service locators. The server manages the service leases. This fat proxy approach does not involve communication back to the server at all. The opposite extreme to this is where all of the processing is done on the server side. The proxy redirects calls from the client, by invoking the respective method in the service on the server, and returns the result to the client. This is the approach Java RMI takes for doing distributed networking. Distributed programming is kept transparent to the programmer. The RMI stub delivered to the client is essentially invisible to the programmer. The server code is written to export the implementation, but the RMI runtime component of Java recognizes this and actually exports the stub instead. This structure is useful when the service needs to do no processing on the client side, but the application logic is located on the server. Further, a combination of these "fat" and "thin" client approaches can be used. Parts of the processing are done in the client side and the rest on the server. This approach does not specify particular communication protocols between client and server. Usually copies of the client have to be distributed out to all machines. If there is a problem with the client, they all have to be updated which is often impossible. Worse, if there is a change to the protocol, then the server must be rebuilt to handle old and new versions while attempts are made to update all the clients. When changes occur, the service and its proxy can be updated together, and there is no need to make changes to the all the clients out on all the various machines because they will download the changes proxy dynamically.

4.9 Protocol Independence

Following from the last section it becomes clear that the used protocol between the client and the service is not mandated by the Jini system. Because a proxy can be uploaded to a client, the proxy can then use whatever protocol it prefers. This is possible because the proxy code, encapsulates the communication protocol to the service, is a dynamically loaded and executed piece of code rather than something that has been associated with the client.

The only requirement that a Jini proxy object has to fulfill is to be an implementation of an interface, which will be used to identify the object in the Jini lookup service. This allows the

client of the service to know what operations a service is supporting. But how the proxy communicates with the service itself is completely up to the proxy and the service from which it comes. As mentioned in the section before, the proxy does not even have to speak to the server at all. The proxy can be an RMI reference, an object that communicates using some other common and well known protocol, such as CORBA's IIOP protocol. Further, the proxy can use a specialized protocol known only to the proxy and the service itself, or a full implementation of the service that runs locally in the client's address space. The point is, that all this is transparent to the client, who only sees the Java interface. The protocol used between a proxy and a service is a private matter between those two objects. They could open up a socket connection, for example and exchange messages using a message structure only they understand. Or they could communicate using a well-known protocol, such as HTTP for example. The proxy could make HTTP requests, and the service could act as an HTTP server handling these requests and returning documents. Client-server applications often communicate using a specialized protocol between the client and server. Jini can simulate this kind of protocol and make the client/server processes part of the Jini system.

The strategy used here is to associate the proxy with the service and dynamically load the proxy on demand. In effect, the proxy and the service can be seen as a single object, which is itself distributed. One part of the object is living in the address space of the client and part of the object living at the location of the service. This approach gives great flexibility to what protocol is actually used. Services can use their own specialized protocols that are optimized for that particular pair of proxy and service. Protocols can evolve over time as new needs emerge. The need for this tightly coupled relationship explains why the Jini system access to the network is Java environment-centric. The feasibility of this approach is dependent on the facility to dynamically download code from the service to the client-code that the client can safely load into its address space and call. Java technology provides this kind of environment and mechanisms to satisfy both RMI and Jini.

4.10 Language Independence

Because of the Java environment-centric access to the network, many have concluded that Jini requires all components in a Jini system must be written completely in Java. Moreover, it is common thinking that all the communication amongst those components is carried out by using the Java Remote Method Invocation system. While this is a common case with Jini services and clients, thinking that this is required misses a central point of the Jini approach to distributed computing.

For an Jini environment, all that is required is a Java environment running to allow the exportation of the Java classes that are the implementation of the services to the lookup service or to the client. That object can be implemented in any language that can be called by the Java Virtual Machine, using the Java Native Interface (JNI) mechanisms. All that is needed is a simple JNI wrapper, not a complete conversion. That is to say, the Java technology-centered nature of the Jini architecture depends on the Java application environment rather than on the Java programming language. Any programming language

can be supported by a Jini system if it has a compiler that produces compliant bytecodes for the Java programming language.

To summarize, there are at least to ways to avoid to use the Java language, that is through the JNI mechanism, or by providing a compiler that can translate a language to Java bytecodes.

4.11 Some Thoughts on Jini

The Jini system promises a new distributed network approach that fits better the needs of the new paradigm of a service-based model and peer-to-peer networking. The previous sections have shown that the Jini specification provides the means to build such systems in a reliable way, by focusing on partial failure and self-healing mechanisms. It is odd that the Jini technology is supported by very ambitious people or companies, who recognize the real power of Jini, but that there are quite a lot people who still have a wrong view on what the Jini technology really is. When Jini first appeared, it was promoted as a system, which enables to dynamically add devices, such as a printer or a digital camera, to a network. Resulting in the Jini system loading automatically the appropriate drivers. Usually, Jini is seen as a system, that enables the plug and play of hardware devices. In addition, compared to other middleware approaches, Jini has not gained much attention in enterprise related computing. For instance, the Enterprise Java Beans (EJB) [SUNEJB] technology has gained much more momentum in industry. The following sections try to elaborate the reasons for this lack of success.

4.12 Community-Based Evolvement

Sun originally released the Jini technology with a unique license, a "community source license" in which Sun makes both the source and binary code for the sample implementation available for use, inspection, and experimentation by developers who wish to join the Jini community. The rules imposed on the community work are few, and mostly entail keeping the type-space clean and publishing sharing profits from commercial applications of the technology with its originators. This model lets developers use the source and ideas behind the Jini technology for research without obligation or limit. The idea behind the license was to give developers the possibility to deal directly with the Jini source code, expanding the number of developers who would try the technology. Sun Microsystems expected a rapid feedback on the feasibility and usefulness of the Jini approach in different environment.

It is Sun's opinion that this kind of licensing model proved effective, resulting in interesting contributions from the users. Using the contributions of the community members a new release of the technology was issued. The Jini community can be seen as an ongoing experiment in trying to mix open source development techniques with industrial engineering development. The fact that Sun Microsystems is taking this approach may be a hindrance for companies to adopt Jini as their middleware backbone for business application development. Because there is no direct support from Sun's part. The Jini model is presented by Sun more like a work still in process, instead of promoting it as an architecture, which provide the capabilities to be used in a "mission-critical" environment. In contrast, in the case of Enterprise Java Beans a specification was issued with the support of many important vendors from different areas, such as BEA, Oracle, or IBM.

4.13 Lack of Marketing Support

Sun Microsystems itself put much less effort in promoting Jini than it puts in Enterprise Java Beans. The technology is introduced as a system intended for small, plugable devices and not suited for enterprise computing. By examining the specification, there is no reason why Jini should not be able to be used in mission-critical applications. Jini allows the implementation of distributed, self-healing systems. Jini is not restricted by its component model, as in the case with EJB, where for instance it is not allowed to use concurrency or threading mechanisms. Further, the Jini implementation provides an implementation of a transaction manager.

It seems, that Sun perceives the EJB model as the better approach to capitalize on. Because there were a lot of companies involved in submitting the specification, they are committed to this approach.

4.14 Security Issues

After having elaborated the more business related reasons for the lack of support for Jini, we will focus on technical related flaws. The most important issue emerging here is the absence of a security system which is appropriate for a distributed, decentralized system like Jini.

Jini is an infrastructure built on top of the mobile code facilities of the Java programming language enabling clients and services to spontaneously engage in arbitrary usage scenarios. This seems to work quite well in a closed infrastructure, but for mission-critical applications it lacks essential security properties. The dynamic nature of locating and using services is one of Jini's major strengths. It is the base for the creation of "plug and play" devices and services. It provides an infrastructure that allows clients to find services independent of both party's location. It is this major strength which makes Jini vulnerable in an untrusted environment, because it is not possible to know who is providing a certain service and what the service is doing. A problem may arise if one wants to use services via an open network like the Internet. Unfortunately, this area is currently untouched by Jini. There are no provisions for data encryption or authentication beyond the abilities of Java 2 and RMI.

4.14.1 Problems that Might Occur

Downloading foreign objects and classes must always be considered a dangerous activity. Amongst other things, the foreign code could export secrets, corrupt data, or prevent other code from operating correctly. Malicious code is a major problem in distributed systems, above when they are connected to the Internet.

Java was designed to cope with downloaded code, and has a fine-grained system of security management. Access to methods is granted on a class-by-class basis, and a call from an untrusted class will result in a security exception. For instance, Java applets were the first Java applications to become well known. One of the main issues with applets is to protect the

underlying operating system from attacks. This is achieved by using the "sandbox approach" which imposes restrictions on the execution of a Java applet inside a browser.

However, the existing implementation of security management is not enough to guarantee the security of Jini-enabled devices. The procedures for downloading objects and classes require extension and modification. Any system that admits downloaded code into its run-time environment is vulnerable to attack. The security mechanisms in Java are intended to minimize the possible effects, imposing restrictions upon the access available to downloaded objects and classes. Jini inherits these mechanisms. Different research efforts in this area, such as [HAKE], have investigated, that it turns out that the security mechanisms introduced with the new RMI extension framework [SUNSEC] are not sufficient for the special needs of a Jini environment.

Usually, the possibility of granting network access to untrusted code should be reduced to a minimum. But in a Jini environment there are ways in which this might occur and which are not obvious at first sight. In [CRICHT] some typical scenarios in a Jini environment in which clients are looking for services were examined. The result was that there are quite a lot of situations where problems might occur. To show how easy it is to compromise a Jini system the results are summarized in the following lines. A client who is searching for a particular service that is not available on the local network, might be tempted to look elsewhere. Because this is happening automatically, the client does not know were the service is coming from. Lookup services can advertise their presence widely, hence are accessible from outside a local network. An external lookup service might be registered as service and offered by a lookup service inside the firewall. To use a lookup service, a client must first obtain a proxy to the lookup service, and give the deserialized proxy access to the local network. The lookup proxy uploaded to the client might be the one from outside the firewall. This could result in using an untrusted service registrar and is one way of granting network access to a piece of untrusted code. Untrusted code with local network access can scan local ports. If the code is a proxy implementing a lookup service, or a stub of a different kind, it can send this information through the firewall. Another problem can appear if a stub for remote objects is checked against another stub using an *equals* methods. Such a check may give a piece of untrusted code a reference to another remote object. It can then make a serialized copy, and send this copy to an untrusted host.

Especially vulnerable remote objects that can be exported in this fashion include the lookup service proxy, lease objects, and event listener objects. In each case, the consequences could be disastrous. A stolen lookup service proxy can give an untrusted host access to an internal lookup service. The lookup service can then be used to provide untrusted services access to the local network. Its clients may be misdirected and misinformed. A stolen lease stub gives an untrusted host an opportunity to use a protected, internal resource. It may also be able to interfere with any other leases that the client holds and misusing them. A stolen event listener stub can be used to send a variety of messages to the recipient's event listener. The event mechanism is intended to allow one device to coordinate the actions of a number of others, so a significant degree of manipulation is possible.

The existing mechanism of security policies, if used properly, can ensure that untrusted code has no direct access to the network, hence avoiding the aforementioned problems. However, there are other ways in which untrusted code can subvert the behavior of the virtual machine.

An untrusted lookup service proxy, denied network access by a access control system, may still be asked to provide services. It may then create an instance of the service being requested, based on trusted code, but with its own implementation of the service. If the request was for a printer service, for example, then the lookup service could create a service item whose codebase points to trusted class file, hence making the requesting entity believe this is a safe service. But the original content, when deserialized, turns out to be replaced by a stub for an untrusted remote object. This stub could redirect a print job to a host outside the firewall, or subvert the printing service in other ways.

On a Jini-enabled device, the environment of the virtual machine will include an RMI security manager. Once this manager is present, any piece of code can attempt to use the RMI class loader by creating a marshaled object and attempting to deserialize it. This can be done by first creating an array of bytes that looks just like a serialized, marshaled object, and turning the buffer into a stream. Calling the *readObject* method, which is part of the serialization API, on the resulting stream will produce a marshaled object of our own design. The codebase may be set to point to an untrusted host, and encode any data that it is wished to send as the class name of the serialized contents. If a method is called on the marshaled object, the RMI class loader will send a request to the untrusted host containing the encoded data. In this way, a piece of untrusted code may export any information that it has access to. For example, a downloaded stub may serialize and send any stubs that it can reference.

A Jini service may need to call another service in order to satisfy its client. If it passes this second service a reference to itself, then it may be giving untrusted code access to its protected members. This is possible whenever the classes associated with the two services are stored in separate Jar files, containing the service classes in a packaged format, and the Jar file for the first service is accessible to an untrusted party. In [CRICHT] is explained how this might occur by considering the example of a database service that calls a graphing service in order to present data to its client. Only an own version of the classes has to be added, implementing the graphing service to the Jar file for the database service, together with an additional class that claims to be in the same package as the database classes. If the database service passes the graphic service a reference to an object that it has defined, the additional class can be loaded. It may then access the protected members of that object. Even if the database classes are package-sealed, this will not deter the class loader, because the additional class has been stored in the same Jar file. Eronen describes in [ERONEN] the solution of all the problems by outlining that they can be solved with a simple modification to the class loader mechanism. Only those files that have been signed by a trusted party should be loaded, anything else should be ignored. This should apply to all files, and not simply those recognized as classes.

If untrusted code is denied access to the local machine, then all of the problems described above are eliminated. However, it is still possible to affect the behavior of a client by

tampering with the marshaled objects that it downloads. If an untrusted host has access to a lookup service on the internal network, then it may substitute its own service for one of those offered. It has only to register a new service with the same service id. The service id is stored in a lookup service to uniquely identify a service. The lookup service will then assume that the original service has moved. The new service item can have the same codebase as the original, that means every class file downloaded will be properly signed and trusted. However, the contents of the service item may be different. When deserialized, they might produce a stub for a remote object on an untrusted host. Note that it is not necessary to substitute for an existing service, simply adding a new service will make it available to clients on the internal network. If more than one service matches a request, then a lookup service may provide any of those available, the choice is randomized to provide a degree of load-balancing. Thus, the lookup service can decide to which service the request is sent. The client is absolutely unaware of this.

4.14.2 Requirements for a Secure Jini Environment

Taking the above mentioned scenarios into consideration some requirements arise. This requirements are absolutely necessary to support a secure Jini environment.

One major difference in the Jini architecture compared to traditional client/server systems like CORBA or the world-wide Web is the relationship between the client and the server. In the traditional approach the client permanently contains the code for communicating with a server. The protocol code is part of the client and therefore part of the client's trusted computing base. Hence, the client/server relationship is very tight. If a client needs some kind of security, such as authentication or encryption, it can choose to use any protocol that provides the required security properties, for instance the Secure Socket Layer (SSL). The Jini approach is fundamentally different. Jini clients do not implement any network protocol at all. It the service's proxy object's task to perform the communication with the server and to decide which protocol to use. As mentioned before, proxy objects originate from some source on the network which is usually untrusted. This includes the download and execution of code from that source. Clients know the interface of these objects but they can not know what these objects are effectively doing. The usual approach to secure the platform which is suggested in the Jini specification, is achieved by the Java sandbox model and appropriate security policies provided by Java. The problem here is that a client does not and cannot know what a proxy object is doing with supplied data. These means would be sufficient for a traditional client/server system. Jini requires a security approach that is different from those of traditional client/server systems. Because the supplied proxy is not in the scope of the client but rather associated with its service, the service should know which kind of security is appropriate for its application domain. The client has therefore to trust the proxy to enforce the correct security constraints. This is not solving the problem of mobile proxies. The problem is not how to establish trust in proxy objects, but rather how to ensure trust in the service provider that supplied the proxy. The problem is shifted from the proxy to the service provider.

[ERONEN] and [HAKE] are expressing ideas and designs for a secure Jini environment. Even if their approach is different they agree on several steps that are necessary to guarantee a secure Jini environment. One step is to ensure proxy integrity. To establish trust the object should not be changed on its way from the service to the client. As said before, an object consists of the two parts state and code. Both parts' integrity must be ensured. It is therefore necessary to digitally sign the code as well as the state. As we do not want anybody to observe the in-traffic service descriptions, the connections between the lookup service and its clients should be encrypted.

As shown above, trusting the proxy is not enough. Even if the communication is encrypted and authentic objects are provided, there is still the need to trust the lookup service. Even if a lookup service provides the Jini client with untampered objects, it might decide to deliver not the most appropriate service, but rather the services preferred by the service provider. From a service provider's view even the knowledge of a service's existence might be considered a valuable asset that must be protected. In a trusted infrastructure it is still possible to have malicious services registered with secure lookup services. The requirement is therefore is that services have to authenticate themselves to the lookup service. Likewise, clients too are required to authenticate themselves to the lookup service. This is an obvious requirement as it is important to make sure that only authorized people access somebody's bank account.

Sun Microsystems is aware of the problems and is currently working on an extension to RMI that is supposed to allow secure interaction with RMI-based servers including the establishment of trust in downloaded proxies. The specification is currently in draft status. It allows fine-grained control of different security properties. While the extension is currently only aimed at RMI it is supposed to be possible to use the same methods and interfaces for other middleware architectures as well. The most interesting part of the specification deals with the establishment of trust in downloaded proxies. The basic method used here is to allow only trusted code to be run. Further security properties, for instance authentication and encryption are then guaranteed by the trusted code. Trusted code includes dynamically generated RMI stubs. If a proxy is not an instance of a trusted class, it is asked to present another object which is trusted. The associated server is then asked if it trusts the original object.

4.14.3 Drawbacks of the Proposed Security Approaches

Some groups are working to find a solution for the security problem, such as [NIKAN]. The right solution has not been found yet. Their methods do, on the one hand, ensure security, but on the other hand restrict spontaneous networking which is a major strength of Jini. The Java approach expressed through the RMI security extension framework is only partially appropriate for the Jini system. A few problems that can be regarded as essential are not addressed. First, objects are instantiated before establishing trust. Malicious code could therefore be executed in the constructor of the proxy. Secondly, the specification is aimed at RMI in general and does not address Jini in particular. Services are therefore still visible to everybody. Different security levels can only be enforced after downloading the service's proxies and depend on their enforcement by every client and server. Missing parts are a

class loader which loads only those classes that are signed and certificated by a trusted party. Uncertificated classes are refused. To solve the problem of malicious objects, a class loader is necessary in which serialized contents of a marshaled object can be signed and certificated.

The result is that even in the examined architecture, a few questions are still open. Most of them assume that there is a central certification instance. In a dynamic environment, a distributed architecture would probably be a more favorable solution. Despite the obvious advantages of a secure service infrastructure, they do not come for free. The drawback is the partial loss of "spontaneity" of client/service interactions, which was said to be one of the main advantages of Jini. Plugging devices and services into the network, spontaneously finding these devices via the lookup service, and using them are easily done. Establishing trust relationships in such spontaneous environments seems to be a task that results in a decrease of spontaneity, since prior to actual use administrative processes, for instance distributing keys, must take place first. Open is the question whether the trade-off between trust and spontaneity can be avoided by additional means that take the mobility of users and devices into account. The gist is expressed by Eronen "without a suitable set of extensions, the Jini system is not ready for adoption in a supposedly-secure environment. The Jini version of plug and play is based around the free exchange of code; without some system of certification, a Jini-enabled device will be forced to trust in the good intentions of others." [ERONEN]

Part C The Web Service-to-Jini bridge

5 The Bridge

5.1 Overview

This section describes a set of facilities enabling a Web Service be translated into a Jini service. The goal of the prototype is to evaluate the promises made by the Web Service paradigm. Above all, the promise of SOAP interoperability, which is the major reason for SOAP's existence, and the ability of SOAP and WSDL in conjunction with XML Schema, to provide a type-safe description of classes and objects in a text-based form. The purpose of the Web Service-to-Jini bridge is to enable the automatic generation of a Jini service, which can access a Web Service created in Microsoft .NET. The bridge does not only provide a SOAP proxy but also generates a Jini interface, to support Jini's way to describe a service. The construction of the bridge entails the usage of Web Service technologies and current-day development tools. Hence, the bridge implementation is used as an approach to evaluate the promises made by the Web Service model, and the maturity of existing tools. The promises and the Web Service technologies were described in the previous parts of this document. The evaluation will be carried out by making heavy use of the points made before.

5.2 The Purpose of the Bridge

The Web Service model claims to be platform and language independent. Moreover, being based on open standards, the model promotes interoperability and the capability to integrate heterogeneous systems. Hence, the purpose of the bridge implementation is to use Web Service technologies from two different platforms and evaluate if they really provide interoperability. The investigation in section on SOAP (14) has shown that interoperability is one of the main issues which will influence the success of the Web Service model.

Further, the combination of SOAP and WSDL supports the means to dynamically access exposed Web Services. The bridge makes use of the service description, to generate automatically Java classes and SOAP calls. This step involves the translation from a text-based representation in XML to an object-based approach, hence it will be examined, if the XML representation can provide enough information to be reconstructed into classes and objects. The intention is to show if the approach is feasible to preserve type information. This ability is necessary to prove the language independence of the approach. Important here is, above all, if the type information can be preserved if the translation takes place among two different languages.

5.3 Requirement Analysis

Now that the purpose has been described, the requirements of the bridge can be defined. In order to provide the functionality needed to carry out a type-safe generation from an XML document to an object-oriented presentation some specific features are required. The Web Service-to-Jini bridge enables further the automatic generation of WSDL files, which can be located everywhere on a network, to Jini services. To provide additional features, the Jini service can be published automatically into a Jini federation. Hence, the bridge requires following features:

- Facilities to locate and download an WSDL document from the network. That can be from the Internet, or from a local network. A local WSDL can be used, too.
- Facilities to analyze service component descriptions. This is carried out by parsing the WSDL document exposed by a Web Service, using an XML parser which is able to deal with namespaces and schemas. The necessary information's have to be extracted.
- Facilities to harness the extracted information to generate automatically a Java interface, which provides the same functionality as the Web Service. In the case of Jini, the functionality is represented as Java interface. The facility provided here must be able to create Java interfaces from an XML schemas representation.
- Facilities to generate automatically the Jini service implementation. The implementation code functions as a proxy to the Web Service operations.
- Facilities to call Web Services from a Jini service. A mechanism is needed that can construct SOAP request messages and handle responses. This SOAP calls are used to use the functionality provided by the Web Service.
- Facilities to publish automatically the Jini service to a Jini federation. As an additional feature, it is possible to publish the generated Jini service to a Jini federation.

5.4 Design and Implementation Criteria

The following will describe the criteria we considered as important. Some of them reflect the above mentioned requirements, others are resulting from the technologies used.

- *Jini service description.* One possible solution for a Jini-Web Service bridge would have been to generate a Jini service, that provides only a proxy to a Web Service. The bridge goes further. To harness the features provided by the Jini environment, a Web Service is wrapped into a self-describing Jini service. Jini services gain their flexibility through their self-describing interface. The interface is published to a lookup server, so every interesting client can search the service through its type definition. Hence, the bridge generates a Java interface using the method names and parameter name provided in the WSDL file, and other service information.

- *Jini architecture approach.* As described in the section on Jini, it is possible to construct Jini services adhering to different architectural patterns. This provides the possibility to

implement Jini services in different shapes. One possibility would be to have a thin client using RMI. Jini's proxy has to be uploaded to the requesting client. The proxies task with this configuration is merely to use remote method invocations to call methods on the server which is providing the Jini service implementing classes. That entails a call to the lookup server which uploads the necessary stub to the requesting client. Having the RMI proxy on the client side the calls can be directed directly to the implementing server. The Jini service has to create the SOAP calls. That means, that the SOAP generating mechanisms are stored on the server, where the Jini service is located. This configuration requires always two calls, from the client to service, and then the service generates the SOAP call and sends the message to the Web Service. The other possibility would be to upload the whole proxy into the client. This results in avoiding the Jini proxy-to-Jini service call across the network, which seems to be a faster solution. The problem here is that the client needs to upload the Apache SOAP classes, which are providing the SOAP API and are generating the call, additionally to the service classes. The jar file size is 216 kilobytes. To let the user decide, the bridge allows the choice between one of the two configuration.

- *Creating reusable service description.* Through parsing and extracting the necessary information from a WSDL file, a Java interface is generated. But first, a XML schema representation is generated, which contains all the necessary information to create an interface and its implementation. It would have been possible to generate automatically the interfaces and its implementation. The additional step was chosen, to provide a reusable representation of the Web Service. The XML schema can be stored and reused later. This file can be reused, for instance, to generate a object-oriented representation in another language than Java, without having to download the WSDL again. The generated XML schema contains only the necessary information, and is therefore smaller and faster to parse, compared to the WSDL file.
- *Type mapping extensibility.* The bridge supports in its current version, only the built-in types defined in the XML schema specification. To provide a flexible design and the possibility to extend the supported types, a TypeMapper class is defined, which contains all the supported type mappings from XML schema types to Java types.

5.5 Selection of Technologies

The used technologies reflect the bridge purpose expressed above. To evaluate SOAP's interoperability, we have chosen to use Microsoft's SOAP and IBM's SOAP to interact. The bridge uses Apache SOAP's API to generate the calls to the Web Service.

Moreover, to show that Web Services are platform independent, the bridge is able to use Web Services implemented, deployed and exposed by Microsoft .NET to generate Jini services, which are implemented and deployed in a Java environment. Because Java is inherently platform independent, the bridge works on Linux, or other platforms supporting Java, too. Hence, we have two platforms using Web Services, that the Java environment using SOAP to access Web Services, in a Microsoft .NET environment.

The bridge is implemented in Java. The generated Jini code is also Java. The Web Service carried out with the .NET development environment shown in the next section, is implemented with C#, a language similar to Java. This results in a Jini service using functionality implemented by an C# application. The Web Service is not directly part of the bridge, but it is mentioned here, to show the cross-language functionality of SOAP.

Moreover, the bridge uses WSDL to extract the needed information to generate automatically a Jini interface and an implementation, consisting of SOAP proxy calls to the Web Service. The bridge uses the Java API for XML parsing (JAXP) in combination with Xalan XML parser provided by Java API to parse the documents. The parser is able to handle namespaces and schemas, which is a requirement for the processing of the WSDL file.

5.6 Operational and Functional Description

5.6.1 The Core Classes

- The `SchemaMapper` class loads WSDL documents. The file can be loaded locally from the harddrive, or from a network resource, such as an URL. Notice, that WSDL documents can be directly accessed by adding #service_name to the URL where the Web Service is provided. (see Appendix A for source code)
- The `JavaSchema` class parses the XML document and extracts the specific parts of the file. The idea is to use the metadata provided and to serialize it into a XML schema specified. This extra step is used so the metadata necessary to generate a language specific interface can be stored and reused. In this case it is used to generate a Java interface. This class takes the aforementioned schema to generate a Java interface from it. It uses the meta information to create a self describing, typed interface. The schema has to provide enough information to make it possible to extract the names and the type information. The result is a Java interface which is going to be used as Jini service description. (see Appendix B for source code)
- The `ImplCode` class enables the next step, that is the implementation of the interface. The interface does not implement the service itself but delegates the requests to the Web Service. Hence, the interface's job is to generate SOAP calls to the appropriate service. Apache SOAP's API is used to generate automatically the required code. If SOAP interoperability is working the Web Service will return a SOAP response and the result will be processed by the Jini service. ImplCode class generates automatically the method implementations. These implementations consists mainly in generating the appropriate SOAP calls and handling SOAP responses. (see Appendix C for source code)
- The bridge provides different additional features. The user can configure the bridge. A feature is, for instance, the possibility to generate the Jini service and automatically publish the service to a lookup service. these part are not part of the core.

In the appendix of this thesis the core classes can be found.

5.6.2 A Walkthrough Scenario

The following description is used to walk through a typical application for generating a Jini service from a Web Service description, and publishing the Jini service automatically to Jini lookup services. This will describe the system's functionality and operational characteristics in more detail. The Jini service is built using the abstract definition of operations and the concrete bindings as defined in the WSDL document. Further, the type information provided in the description document is used to deserialize the XML document into Java entities. Hence, Jini clients or other services can look up the service using its type.

This example will use a Web Service generated using Microsoft .NET. The Web Service is implemented with C#. The implementation of the service with .NET is very straightforward. The application logic is implemented without having to consider that the service is going to be exposed as a Web Service. After having implemented the application, the developer can use tools to define the methods which should be exposed to the Internet. The tools generate automatically the necessary WSDL document and the Web Service can easily be deployed in a Web Server. The Web Service used provides a simple banking interface. A client can add or withdraw money from its account. To make use of this service in a Jini environment the user has to carry out following steps.

- The bridge provides a batch file. The user has only to use this batch file, and configure it with the functionality the he wants to use. The batch file can be applied as follows:

  ```
  WStoJini –dir directory_name –wsdl wsdl_location –proxy thin|fat –
  jini pub|unub
  ```

  The WStoJini batch file provides an easy mechanism for the user, to use the bridge functionality. The user can define with the `–dir` option in which directory the generated files should be saved. This option is not mandatory, if no value is provided the files will be saved in the current directory. The `–wsdl` option is mandatory and requires the URL where the WSDL file can be located. The location can be an Internet address, or a local file. The `–proxy` option allows the user to chose if the Jini proxy should be generated as a thin client using RMI calls to the server, as explained in section 4.8 "Architecture Independence" (60), or a fat proxy, entailing to download the SOAP API. The default value is thin. The last option provides the possibility to automatically publish the generated Jini service in a lookup server. If the pub option is chosen, the Jini service will automatically be published. The default value is unpub. Notice, it is not the purpose of the bridge, to provide all the necessary Jini configuration. If the user is going to publish the service, it is his task to provide a configured Jini environment on the system. The next points will describe what s going behind the scenes

- The provided WSDL location is passed to the `SchemaMapper` class. Then the class tries to load the WSDL into memory. If the document is available at the provided URL, it is parsed and the XML parser creates a Document Object Model (DOM) representation of the document. The DOM is a standard tree-like representation stored in memory. A user can traverse this tree, extracting information, or changing the structure of the tree by, for instance, adding or removing nodes. The mapper class uses this representation of the WSDL document to extract the needed properties. The mapper class uses the `<types>`

element in the WSDL file, to get all the information about the types used in the service. These type information are necessary to generate the right method parameters and the method return values. The following code shows a part of the <types> from the banking WSDL file. Here, the types passed to an NewAccount operation are specified. Additionally the target namespace of the document is provided.

```
<types>
    <s:schema attributeFormDefault="qualified"
elementFormDefault="qualified"
targetNamespace="http://eratos.csse.monash.edu.au/banking">

      <s:element name="NewAccount">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="l_name"
nillable="true" type="s:string" />
            <s:element minOccurs="1" maxOccurs="1" name="l_email"
nillable="true" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
            …
   </types>
```

Further, the mapper extracts all the information needed to locate and access the Web Service through an SOAP call from the <service> element. That entails the unique identifier, and the URL, where the service is exposed. The SOAP structure needs this information, for instance for the SOAPAction option in the header. As shown in the next code snippet the <operation> element provides the SOAPAction value which is necessary to uniquely identify a method. The <service> element embraces the <port> element, which contains the <soap:address> identifier and the location of the specific Web Service.

```
   …
   <operation name="Withdraw">
     <soap:operation
          soapAction=http://eratos.csse.monash.edu.au/banking/Withdraw
          style="document" />

   …
   <service name="BankingService">
       <port name="BankingServiceSoap" binding="s0:BankingServiceSoap">
         <soap:address
   location="http://eratos.csse.monash.edu.au/banking/banking.asmx" />
       </port>
          …
       </port>
     </service>
```

Additionally, the mapper extracts the service name, the names of the operations, and all the names of the operation's parameters. These names can be used to generate a Jini

service with aquivalent names. More information on this WSDL file and the generated code can be found in the Appendix D.

All this information is used to generate an XML schema which then reflects the Web Service. The defined schema has following structure:

Having the XML representation of the Web Service makes it now possible to generate actually the Jini interfaces and implementation. The `SchemaJava` class does this. The class uses an XML parser to extract the method and type information to generate a Java interface. Further, the URL address and the URN identifier are added to the interface as typesafe constants, because this values are true for all the operations, they are inserted in the interface as global attributes. The `SchemaJava` class uses the `TypeMapper` class to map XML schema types to Java types. See Appendix D for the generated code.

- The missing part is the implementation of the Java interface, which provides the functionality for the Jini service. The Jini service does not implement the service functionality, rather it creates a SOAP call to the Web Service implementing the functionality. That means, the task of the Jini service is to redirect call, by translating the method invocation on the Jini interface to a SOAP call on the Web Service. The Apache SOAP API provides a `Call` class, to define the call parameters and automatically generates the appropriate call. Because each operation which can be called on the Web Service needs its own call structure, all interface methods have to be implemented by providing the code necessary to construct the call message. The following code extract shows a part of the generated code. See Appendix D for the generated code.

5.7 Results

The walkthrough scenario described in the previous section, was carried out in an environment, which was defined to evaluate the Web Service model and its technologies. This testbed consists of a Microsoft Internet Information Server (IIS) running on a Windows 2000 machine. On this machine resides the banking Web Service, which is implemented in C#. The Web Service was deployed and configured using Microsoft .NET and its Web Service tools. On another machine runs the Web Service-to-Jini bridge. This machine has the Apache SOAP API and IBM Web Service Toolkit installed and configured. Moreover, the Jini API is provided. This environment was chosen because it provides all the necessary technologies to evaluate the Web Service model. It is examined if the promise of language independence holds, by using Java and C#, both using SOAP calls to communicate. Further, the call crosses technology boundaries by accessing functionality implemented in an Microsoft environment from a Java-centric environment.

The WSDL file downloaded can be found in the Appendix D. Further, the generated Jini interfaces, and the implementation are located there, too. The previous section has already shown some parts of it. The important part is that the Web Service is responding with the right SOAP response message. A .NET tool (MSoapT) is used to observe the all the requests that are sent to the server and all the responses that the server generates. For instance, in the

case that the generated Jini service creates a SOAP call to invoke the withdraw operation on the Web Service the following SOAP response is returned by the server.

```
<?xml version=”1.0” encodoing=”UTF-8” standalone=”no”?>
<SOAP-ENV:Envelope SOAP-
ENV:encodingStyle=”http://schemas.xmlsoap.org/soap/encoding/”
 xmlns:SOAP-ENV=”http://schemas.xmlsoap.org/soap/envelope/”>
<SOAP-ENV:Body>
  <m:balanceResponse xmlns:m= >
    <Result>120</Result>
  </m:balanceResponse>
</SOAP-ENV:Body>
<SOAP-ENV:Envelope>
```

Observing this response message, it is possible to see, that the result element is not typed, and hence causing the interoperability problem with Apache SOAP. This situation is handled by instructing the generated Apache SOAP client not to expect the type information in the response received. Rather, the mapping registry, managed by the SOAP implementation, must be passed explicitly the necessary type information. Apache SOAP uses a mapping registry to inform its SOAP server which types are supported, and which kind a deserialization mechanism has to be used for each type. This is done by following lines:

```
    SOAPMappingRegistry smr = new SOAPMappingRegistry();
    StringDeserializer sd = new String();
    smr.mapTypes(Constants.NS_URI_SOAP_ENC, new QName("", "Result"), null,
null, sd);
```

The deserializer type has to be changed, if a different type is needed. This is handled by the bridge. The deserializer is a mechanism provided by Apache SOAP to automatically deserialize a XML schema type and its value in a Java representation. A serializer is also provided to map a Java object in a XML schema representation.

The result that can be found here is that SOAP interoperability works, though some minor changes have to be made. But this can be seen as a problem of the early stage of the development tools provided. This tool incompabilities will be changed in the close future. The language independence and platform independence is proved, and further the integration of heterogeneous systems is possible. This is achieved, without loosing functionality across system boundaries. The test was not run behind a firewall, hence it is not possible to make a statement in that case. But, because the SOAP calls are strictly built on HTTP, the possibility that the calls will pass a firewall barrier is very high.
The step from one object-oriented language, such as C#, to an another language, such as Java, is possible, without loosing type information. This is achieved by providing a standard encoding standard based on XML.

5.8 SOAP Interoperability with Apache SOAP and Microsoft

The results carried out are true for Apache SOAP release 2.1 or above, and Microsoft SOAP Toolkit Beta 2 or higher. The investigation shows that these implementations are not naturally interoperable. The occurring problem is caused by the `xsi:type` attribute which is based on different views on SOAP. The SOAP specification provides the possibility to define the types of elements which are part of the SOAP envelope. This information is an optional part. The SOAP implementer can provide other means to communicate the type information, they do not have to be included directly in the SOAP message. It is recommended to use the `xsi:type` attribute only if no other means are supported. As shown in section 2.9 "Critical look" (35), the size of a SOAP message call is smaller without providing the type information inside each call, and hence the better solution.

Apache SOAP demands the type attribute to be always added in a SOAP message. Whereas, Miscrosoft's SOAP does not require this information. Microsoft's SOAP implementation has a built in dependency on an external service description document. This document describes the data type and can be accessed by requester and provider of the service. Apache's SOAP does not understand a description language, such as WSDL. Both implementations adhere to the specification, but are not compatible. Microsoft's SOAP was first dependent on the Microsoft proprietary Service Description Language (SDL), but changed then to support the open standard WSDL. Hence, Microsoft SOAP requires the use of a WSDL file that describes the interface and location of the service.

The ILAB mentioned in the SOAP section above revealed the incompatibilities and both companies are providing a workaround in their newest implementations, proving the usefulness of the SOAP community and their commitment to SOAP. The bridge makes use of these enhancements, as shown above, and has evaluated its feasibility in a generic environment.

Because the Jini service succeeds in calling the Web Service we can deduce from that that this to SOAP implementations are interoperable. Of course, further investigations are necessary, for instance the bridge uses only built-in types. But the successful scenario carried out by the bridge shows, that the SOAP implementers are committed to the idea of platform integration through SOAP. Further, they have established the right means to track interoperability issues, in the form of the ILAB. The ILAB has already proven its usability in the case of the problems emerged in our example.

5.9 Known Limitations and Evolution Path

The bridge is built in that way that it is possible to carry out the necessary evaluation. The bridge can easily be extended to support further functionality. For instance, the bridge supports only simple, built-in types in its current status. Other types are rejected. The bridge can be extended with support for more types by adding the necessary mapping information into the `TypeMapper` class, or a mechanism to automatically generate Java types from complex types, as they are defined in XML schema, can be provided inside the `TypeMapper`.

The first approach is relatively inflexible because the `TypeMapper` class can only add mappings about types already known before. Hence, the type mapping will fail, for not previously known types. In an Internet environment it is quite likely that new types are appearing. The second approach provides a more dynamic mechanism, and as long as the types adhere to the type specification in the XML schema, the conversion should not be a problem. But note, that the SOAP mechanism used, must also be able to understand the types used. Apache SOAP provides, in its current state, serializer and deserializer for mapping issues, but does not support a generic mechanism which is able to handle all the types.

The bridge does not generate additional information about a Jini services. To discover a Jini service dynamically in a lookup service, the client can use the type to do so, or additional provided metadata. This metadata is encapsulated in, so called, *Entry* objects. For instance, a service can provide additional information about some specific functionality it provides. A client can then search using a pattern matching mechanism to find an appropriate service, instead of using the type. This information is usually provided by the service implementer. The bridge can not predict what metadata a service may want to expose. A solution for this would be to provide a graphical user interface, instead of only the batch file. Here, the user can add the information he wants to be exposed with the Jini service. Moreover, the user would have the possibility to specify its own service and message names.

The `SchemaMapper` can handle only WSDL documents, which are generated with Microsoft .NET tools. This is not a limitation imposed by the bridge, but rather introduced by the WSDL tools used. Though, WSDL specifies an open standard, there are still some incompabilities amongst different tools. In the SOAP section the problems are mentioned. There are two possibilities to solve this problem. The first is to wait until all the tools are supporting the same WSDL standard, or to provide an own `SchemaMapper` class for each distinct WSDL definition. The same is true for the SOAP call generation. The bridge uses an extension to provide Microsoft and IBM SOAP interoperability. This extension might cause problems when trying to access another implementation.

6 Conclusion

With the approach we have taken to evaluate the Web Service model we believe that the model, even if it is still in an early stage, will have a great impact in the future way of building Web-based applications. Because of performance problems, SOAP is not the best solution for server-to-server communication, where high-speed communication is necessary. But the rapid evolution of software and hardware will solve this problem. The advantages of SOAP can be interesting in every context. Above all the capability to serve as an intermediary protocol to integrate two different systems, makes SOAP a very powerful mechanism. The investigation has shown that one major issue for the success of the Web Service model, above all of SOAP is, that interoperability is guaranteed. The approach that the model has taken, namely using only open standards as underlying technologies, is an important step to a standardised component model.

The implementation of the Web Service-Jini bridge has show, that WSDL and SOAP are able to provide platform- and language-independence. Further, the use of SOAP, as communication protocol, enables integration of heterogeneous systems. It is possible with the use of WSDL, SOAP and other XML standards to translate from one object-oriented language to another, with XML schema as an intermediary language without loosing type information. The bridge has proved, that calls across technology boundaries are possible. Though, in the current state there are still some problems to solve, as examined with Microsoft SOAP and Apache SOAP. But, there is evidence that the involved companies are committed to the Web Service approach, and tools vendors are working together to enable a seamless integration of different tools and development environments.

References

[AGRAW] Agrawal R., Bayardo R., "Vinci: A Service-Oriented Architecture for Rapid Development of Web Applications", IBM Almaden Research Center, http://www.

[BOX] Box, D., "A Young Person's Guide to The Simple Object Access Protocol: SOAP Increases Interoperability Across Platforms and Languages", MSDN Magazine, March 2000, http://msdn.microsoft.com

[CRICHT] Crichton C., Davies J., Woodcock J., „When to trust mobile objects: access control in the Jini Software System", Oxford University Computing Laboratory

[ERONEN] Eronen, P., Nikander, P., „Decentralized Jini Security", Helsinki University of Technology

[FISCO] Fisco D., "IBM's Web Services arcitecture debuts", IBM Corp., September 2000, http://alphaworks.ibm.com

[GISOLFI] Gisolfi, D., "Web services architect, Part 3: Is Web services their reincarnation of CORBA?", IBM Emerging Technologies, July 2001,
http://www.ibm.com/developerworks/webservices

[GOVIND] Govindaraju M., Slominski A., "Requirements for and Evaluation of RMI Protocols for Scientific Computing", Indiana University, August 2000, http://www.extreme.indiana.edu/soap/

[HAKE] Hasselmeyer, P., Kehr, R., Voß, M., „Tade-offs in a Secure Jini Service Architecture", Departement of Computer Science, Darmstadt, September 2000

[JAVASERIAL] Sun Microsystems, "Java Object Serialization Specification", Sun Microsystems, November 1998

[JINI1] Sun Microsystems, "Jini Network Technology", Sun Microsystems, 2001, http://www.sun.com/jini/

[JINI2] Sun Microsystems, "Jini Architectural Overview", Sun Microsystems, 2001, http://www.sun.com/jini

[KAO] Kao J., "Developer's Guide to Building XML-based Web Services with the Java 2 Platform, Enterprise Edition (J2EE)", SUN Microsystems, June 2001

[KIRT] Kirtland,K., "A Platform for Web Services", Microsoft Developer Network, January 2001, http://msdn.microsoft.com

[KREGER] Kreger, H., " Web Services Conceptual Architecture (WSCA 1.0)", IBM Software Group, May 2001, http://www.ibm.com/developerworks/webservices

[MEYER] Meyer, B., "What to Compose", Software Development Online, March 2000, http://www.sdmagazine.com

[NIKAN] Nikander P., „Fault Tolerance in Decentralized and Loosely Coupled Systems", Ericsson Research Nomadic Lab

[ORACLE] Oracle Corp., “Oracle Developing, Deploying, Managing Web Services with Oracle9*i*”, An Oracle White Paper, June 2001, http.//technet.oracle.com

[SZYP] Szyperski, C., “Components and Architecture”, Software Development Online”, October 2000, http://www.sdmagazine.com

[STEIN] Steinberg D., „Jini aims at JavaOne“, IBM Corp, June 2001, http://www.ibm.com/developerworks/java/

[SUNEJB] Sun Microsystems, “Enterprise Java Beans specification 1.1”, Sun Microsystems, 2000, http://www.javasoft.com

[SUNRMI] Sun Microsystems, “Java Remote Method Invocation Specifcation”, December 1998, http://www.javasoft.com/products/jdk/rmi/

[SUNSEC] Sun Microsystems, “RMI Security Extensions”, Sun Microsystems, 2001, http://www.javasoft.com

[SUNJVM] Sun Microsystems, “The Java Virtual Machine Specification”, Sun Microsystems, 1999, http://www.javasoft.com/

[WALDO] Waldo, J., Wyant G., “A Note on Distributed Computing”, Sun Micorsystems, November 1994

[WALDO2] Waldo J., “The Jini Architecure for Network-centric Computing”, Sun Microsystems, July 1999.

[WSDL] Microsoft Corp., “Web Service Description Language (WSDL) 1.1”, Microsoft Corp., January 2001

[XMLSCHEMA0] W3C, “XML Schema Part 0: Primer”, W3C Recommendation, May 2001, http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/

[XMLSCHEMA1] W3C, “XML Schema Part 1: Structures”, W3C Proposed Recommendation, March 2001, http://www.w3.org/TR/2001/PR-xmlschema-1-20010316/

[XMLSCHEMA2] W3C, “XML Schema Part 2: Datatypes”, W3C Recommendation, May 2001, http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

[W3CSOAP] W3C, “Simple Access Protocol (SOAP) 1.1”, W3C Note, May 2000, http://www.w3.org/TR/SOAP

Appendix A – The `SchemaMapper` class

```
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import org.w3c.dom.*;

import java.io.*;
import java.net.URL;
import java.util.*;

public class SchemaMapper {
    /** file where the schema is stored */
    private File xmlFile = null;

    /** Storage for methods */
    private Map methods    = new HashMap();

    /** Storage for method parameters and its type definitions */
    private Map parameters = new HashMap();

    /* to identify specific methods */
    private String key = null;

    /** stores the service name */
    private String serviceName = null;

    /*stores the service location */
    private String location = null;

    /** stores the unique identifier of a service */
    private String uri = null;



    /**
     * <p>
     *  <code>SchemaMapper</code> handles the generation of an XML schema file
     *    from an WSDL document.
     * The constructor allows to the wsdl document to be supplied as an url.
     *
     * @param wsdlFileUrl <code>URL</code> where wsdl document is located
     * @throws <code>IOException</code> when problems in locating occurs
     *
     * @author David Wuerth
     */
    public SchemaMapper(URL wsdlFileUrl) throws IOException {

        /** Creates builder to generate DOM presentation of the WSDL file */
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        DocumentBuilder db = null;


        try {
            db = dbf.newDocumentBuilder();
        }catch(ParserConfigurationException e) { System.err.println(e); }

        Document doc = null;

        // try to get the document root from the wsdl file
        try {
            doc = db.parse(wsdlFileUrl.toFile());
```

```
    }catch(SAXException ex) {
        System.err.println("ex" + ex);

    }catch(IOException e) {
        System.err.println("e" + e);
    }

    Element elem = doc.getDocumentElement();

    // process steps necessary to generate schema
    getURI(elem);
    getLocation(elem);
    getAction(elem.getElementsByTagName("binding").item(0));
    NodeList nl = elem.getElementsByTagName("types");

    processNode(nl.item(0));

    // calls the method to generated the whole schema
    generateSchema();

    //writes generated schema in file
    writeSchema(generateSchema(), new File(args[0]));
 }


 /**
  * <p>
  *  This will handle the extraction of an uri from the
  *    WSDL document
  * </p>
  *
  * @param node <code>Element</code> to extract from.
  *
  */
 public void getURI(Node node) {
    Element element = (Element)node;
    NamedNodeMap atts = node.getAttributes();

    // gets the uri from the target namespace attribute
    uri = atts.getNamedItem("targetNamespace").getNodeValue();
 }


/**
  * <p>
  *  This will handle the extraction of the service location from
  *    the WSDL document
  * </p>
  *
  * @param node <code>Element</code> to extract from.
  *
  */
 private void getLocation(Node node) {
    Element element = (Element)node;
    NodeList nl = element.getElementsByTagName("service");
    Node service = nl.item(0);
    NodeList list = element.getElementsByTagName("soap:address");

    // extract the service name in the <service> element
    serviceName = service.getAttributes().getNamedItem("name").getNodeValue();

    //extract the location from inside the <location> element
    location = list.item(0).getAttributes().getNamedItem("location").getNodeValue();
```

```
    return;
}



/**
 * <p>
 *  This will handle the extraction of the SOAPAction field
 *    from the WSDL document
 * </p>
 *
 * @param node <code>Element</code> to extract from.
 *
 */
public String getAction(Node node) {
    Element element = (Element)node;

    // extracts the child elements from the <soap:operation> element
    NodeList list = element.getElementsByTagName("soap:operation");

    // extracts all the SOAPAction fields provided for the appropriate operation
    for(int i=0; i<list.getLength(); i++) {
        NamedNodeMap atts = list.item(i).getAttributes();
        String soapAction = atts.getNamedItem("soapAction").getNodeValue();
    }

    return;
}


/**
 * <p>
 *  Helper method to traverse recursively through a tree
 * </p>
 *
 * @param node <code>node</code> which is going to be traversed.
 *
 */
public void processNode(Node n) {
    NodeList nl = n.getChildNodes();

    for(int i=0; i<nl.getLength(); i++) {
        Node node = nl.item(i);

        echo(node);
        processNode(node);
    }
}


/**
 * <p>
 *  Exmamines the type of an specific element.
 *  Elements which are not necessary for the purpose are discared.
 * </p>
 *
 * @param node <code>node</code> which is going to be traversed.
 *
 */
public void echo(Node node) {
    int type = node.getNodeType();
```

```
    switch(type) {
        //if the node type is element, extract the neessary information
    case Node.ELEMENT_NODE:
        if(node.hasAttributes()) {

            NamedNodeMap atts = node.getAttributes();


            if(atts.getLength() == 1) {
                // add method key and name to storage
                key = atts.getNamedItem("name").getNodeValue();
                methods.put(key, new HashMap());
            } else {
                String paramName = null;
                String paramType = null;

                for(int i=0; i<atts.getLength(); i++) {

                    // all the parameter names and values are extracted
                    if(atts.item(i).getNodeName().equals("name")) {
                        paramName = atts.item(i).getNodeValue();
                    }

                    //all the type information is extracted
                    if(atts.item(i).getNodeName().equals("type"))
                        paramType= atts.item(i).getNodeValue();

                    //the data is stored inside a storage
                    if(paramName != null && paramType != null) {
                        // add all method parameter names and types
                        // to appropriate method
                        parameters = (HashMap)methods.get(key);
                        parameters.put(paramName, paramType);
                    }
                }
            }
        }

    default:
        break;
    }
}

/**
 * <p>
 *  Method which uses the extracted information from the WSDL document
 * to generate an XML schema
 * </p>
 *
 * @return <code>String</code> containig the whole generated XML schema
 *
 */
public String generateSchema() {
    String interName = null;
    String methodEnd = null;

    // temorary buffer containing the XML schema
    StringBuffer fileBuffer = null;

    // adds schema name
    String schema = new StringBuffer().append("<?xml version=\"1.0\"?>").toString();

    // adds service description name
```

```
        String serviceDescr = new StringBuffer().append("<ServiceDescription name=\""
+serviceName + "\">").toString();

        // adds service location
        String location = new StringBuffer().append("<location>" +  location  +
"</location>").toString();

        // adds unique identifier
        String uri = new StringBuffer().append("<uri>" + uri + "</uri>").toString();

        // adds the end tag of the document
        String serviceEnd = new StringBuffer().append("</ServiceDescription>").toString();

        // extract interface name and add it
        int index = serviceName.indexOf("Service");
        interName = new StringBuffer().append("<InterfaceDescription name=\"" +
serviceName.substring(0,index) +"\">").toString();

        // create schema
        fileBuffer.append(schema + "\n")
            .append(serviceDescr + "\n")
            .append(location + "\n")
            .append(uri + "\n")
            .append(serviceEnd + "\n")
            .append(interName + "\n");

        // iterate trough the storage containing all the method and parameter information
        Iterator iter = methods.keySet().iterator();
        while(iter.hasNext()) {
            String key = (String)iter.next();

            // add method name to schema
            fileBuffer.append("<method name=\"" + key + ">");

            HashMap tmp = (HashMap)methods.get(key);

            // iterates through the parameter list to get all the method arguments
            for(Iterator i = tmp.keySet().iterator();i.hasNext(); ) {
                String txt = (String)i.next();
                String type = (String)tmp.get(txt);
                int index2 = type.indexOf(":");

                // add parameter names and types to schema
                fileBuffer.append("<param name=\"" + txt + " type=\""
+type.substring(index2+1, type.length())+  "\"/>");
            }

            // indicates the end of an method definition
            methodEnd = new StringBuffer().append("</method>").toString();
            fileBuffer.append(methodEnd);
        }

        // indicates the end of a service description
        String interNameEnd = new StringBuffer().append("</InterfaceDescription>").toString();
        fileBuffer.append(interNameEnd);

        return fileBuffer.toString();
    }

    /**
     * <p>
     *  This will write the generated schema into a file.
     * </p>
```

```
     *
     * @param schema <code>String</code> containing the schema
     * @param file <code>File</code> to write to.
     * @throws <code>IOException</code> - when output errors occur.
     */
    public void writeSchema(String schema, File file) throws IOException {
        FileWriter writer = new FileWriter(file);

        writer.write(schema);
        writer.flush();
        writer.close();
    }

    public  static  void main(String[] args) {
        SchemaMapper test = new SchemaMapper(args[0]);
    }
}
```

Appendix B - The `JavaMapper` class

```
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import org.w3c.dom.*;

import java.io.*;
import java.net.URL;
import java.util.*;

public class SchemaJava {
    static final String outputEncoding = "UTF-8";
    private File xmlFile = null;

    // storage for the methods to be mapped
    private Map methods    = new HashMap();

    // storage for the methods parameters
    private Map parameters = new HashMap();

    // stores all the necessary information for the description
    private String key = null;
    private String serviceName = null;
    private String locTmp = null;
    private String uriTmp = null;
    private String uri = null;
    private String interfaceName = null;


    /** stores the Interface code to generated */
    private Set interfaceCode = new HashSet();

    /** the ImplCode class generates the implementation code */
    private ImplCode icode = new ImplCode();


    /**
     * <p>
     *  <code>SchemaJava</code> handles generation of Java interfaces and classes
     *    from a specific XML Schema, containing a service description.
     *
     * </p>
     *
     * @author David Wuerth
     */
    public SchemaJava(URL schemaUrl) {

        // creates builder to generate DOM docuemnt root from the wsdl file
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        DocumentBuilder db = null;

        try {
            db = dbf.newDocumentBuilder();
        }catch(ParserConfigurationException e) { System.err.println(e); }

        Document doc = null;

        // try to get the document root from the xml schema file
        try {
            doc = db.parse(xmlFile);
        }catch(SAXException ex) {
```

```
            System.err.println("ex" + ex);
        }catch(IOException e) {
            System.err.println("e" + e);
        }

        Element elem = doc.getDocumentElement();


        /** process steps necessary to generate the Java interfaces and implementations */
        getInterfaceAttributes(elem);
        getMethods(elem.getElementsByTagName("InterfaceDescription").item(0));
        writeCode();

        // call methods in ImplCode class to generate implementation code
        icode.generateImportCode(interfaceName, url);
        icode.generateMethodCode("string", uri, "Login", param));
    }

    /**
     * <p>
     *  Extracts the interface attributes from the xml schema.
     * </p>
     *
     * @param node <code>Element</code> to extract from.
     *
     */
    public void getInterfaceAttributes(Node node) {
        Element element = (Element)node;
        NamedNodeMap atts = node.getAttributes();

        uri = atts.getNamedItem("name").getNodeValue();
        System.out.println(uri);

        NodeList loc = element.getElementsByTagName("location");
        NodeList uri = element.getElementsByTagName("uri");

        // gets the service location
        String location = loc.item(0).getAttributes().getNamedItem("url").getNodeValue();

        // gets the service unique identifier
        String uriName = uri.item(0).getAttributes().getNamedItem("name").getNodeValue();


        locTmp = new StringBuffer().append("final static String url = \"" + location +
"\"").toString();
        uriTmp = new StringBuffer().append("final static String uri = \"" + uriName +
"\"").toString();
    }



    /**
     * <p>
     *  Extracts the methods from the xml schema.
     * Adds the method parameters and types
     * </p>
     *
     * @param node <code>Element</code> to extract from.
     *
     */
    public void getMethods(Node node) {
        Element element = (Element)node;
        NamedNodeMap atts = node.getAttributes();
```

```
        String returnParam = null;
        String method = null;
        StringBuffer buffer = null;

        // gets interface name
        interfaceName = atts.getNamedItem("name").getNodeValue();

        // adds interface name to the generated interface
        String interfaceStart = new StringBuffer().append("public interface
").append(interfaceName + "{").toString();

        interfaceCode.add(interfaceStart);

        NodeList nl = element.getElementsByTagName("method");

        for(int i=0; i<nl.getLength(); i++) {
            String name = getNamedValue(nl.item(i), "name");
            String[] part = new String[nl.getLength()];
            buffer = new StringBuffer();

            NodeList list = getList(nl.item(i), "param");

            // iterates through all methods and parameter information provided in the xml
schema document
            for(int j=0; j<list.getLength(); j++) {
                String paramName = getNamedValue(list.item(j), "name");
                String paramType = getNamedValue(list.item(j), "type");

                // checks if parameter information is a method argument or a return value
                if(name.endsWith("Response")) {
                    returnParam = new String(paramType);

                    if(returnParam==null)
                        returnParam="void";
                    break;
                }
                //adds parameters to method
                else {
                    part[j] = new StringBuffer().append(paramType + " " +
paramName).toString();

                    if(list.getLength()==1 || j==list.getLength()-1) {
                        buffer = buffer.append(part[j]);
                    }
                    else {
                        buffer = buffer.append(part[j]).append(", ");
                    }
                }
            }

            // adds method signatures to an method
            method = new StringBuffer().append("  public " + returnParam + "
").append(name).append(" (" + buffer.toString() + ");").toString();

            // adds all the extracted methods to the generated interface code
            interfaceCode.add(method + "}");
        }

        NodeList list = getList(nl.item(0), "param");
    }

    /**
     * <p>
```

```
     *  Helper method to get named values from a node.
     *
     * </p>
     *
     * @param node <code>Element</code> to get value from.
     * @param name <code>String</code> containing node name.
     *
     * @return String containing a named value.
     */
    public String getNamedValue(Node node, String name) {
        return node.getAttributes().getNamedItem(name).getNodeValue();
    }
    /**
     * <p>
     *  Helper method to get a node list.
     * </p>
     *
     * @param node <code>Element</code> get list from.
     * @param name <code>String</code> contains name of the node.
     *
     * @param NodeList containig nodes from a specific element
     */
    public NodeList getList(Node node, String name) {
        Element element = (Element)node;

        return element.getElementsByTagName(name);
    }

    /**
     * <p>
     *  Extracts the return value of a methods from the xml schema.
     * </p>
     *
     * @param node <code>Element</code> to extract from.
     *
     */
    public void getReturnValue(Node node) {
        Element element = (Element)node;

        NodeList list = element.getElementsByTagName("param");
        System.out.println(list.getLength());
    }

   /**
     * <p>
     *  Writes the generated code on the standard output
     * </p>
     *
     *
     *
     */
    public void writeCode() {
        System.out.println("\n\n\n");

        for(Iterator i=interfaceCode.iterator();i.hasNext(); )
            System.out.println((String)i.next());

    }


    /**
     * <p>
     *  This will write the generated code into a file.
```

```
     * </p>
     *
     * @param schema <code>String</code> containing the schema
     * @param file <code>File</code> to write to.
     * @throws <code>IOException</code> - when output errors occur.
     */
    public void writeSchema(String code, File file) throws IOException {
        FileWriter writer = new FileWriter(file);

        writer.write(schema);
        writer.flush();
        writer.close();
    }


    public String generateImplCode() {
        StringBuffer code = new StringBuffer();

        //calls the ImplCode method to generate code
        code.append(icode.generateImportCode(interfaceName, location));
        code.append(icode.generateMethodCode("string", uri, "Login", param));

        return code.toString();
    }


    public  static  void main(String[] args) {
        try {
            SchemaJava Test = new SchemaJava(new File(args[0]).toURL());
        }catch(Exception e) { System.err.println(e); }
    }
}
```

Appendix C – The `ImplCode` class

```
import java.util.*;

public class ImplCode {


  /**
     * <p>
     *  This class generates the implementation code for
     *  a provides interface.
     * </p>
     *
     * @param directory <code>File</code> to write to (should be a directory).
     * @throws <code>IOException</code> - when output errors occur.
     */
    public ImplCode() {

    }

  /**
     * <p>
     *  Generates implementation code for a specific interface.
     *  Adds necessary import statements and implements keyword.
     * </p>
     *
     * @param name <code>String</code> contains name of the interface to be implemented.
     * @param url <code>String</code> contains the location of the service.
     *
     * @return String contains the generated code
     */
    public String generateImportCode(String name, String url) {
        String code = null;

        // generates the implemenation code for the interface
        code = new StringBuffer().append("import java.io.*;\n")
            .append("import java.util.*;\n")
            .append("import java.net.*;\n")
            .append("import org.w3c.dom.*;\n")
            .append("import org.apache.soap.util.xml.*;\n")
            .append("import org.apache.soap.*;\n")
            .append("import org.apache.soap.encoding.*;\n")
            .append("import org.apache.soap.encoding.soapenc.*;\n")
            .append("import org.apache.soap.rpc.*;\n")
            .append("import org.apache.soap.transport.http.SOAPHTTPConnection;\n")
            .append("\npublic class SOAP" + name + "Client implements " + name + " {\n")
            .append("  private URL url = new URL(" + url + ");\n")
            .append("\n  public SOAP" + name + "Client() throws Exception {\n")
            .toString();

        return code;
    }


    /**
     * <p>
     *  Generates implementation code for a specific method.
     *  Main task is to add the SOAP call construction.
     * </p>
     *
     * @param returnType <code>String</code> contains the return type.
     * @param uri <code>String</code> contains the unique identifier for the service.
```

```
     *
     * @return String contains the generated code.
     */
    public String generateMethodCode(String returnType, String uri, String methodName, Vector
param) {
        String method = null;
        StringBuffer paramsTmp = null;
        String params = null;
        String fault = null;
        int counter = 0;

        // generates the SOAP call using the extracted information from the xml schema
        method = new StringBuffer().append("  SOAPMappingRegistry smr = new
SOAPMappingRegistry();\n\n")
            .append("  " + returnType + "Deserializer sd = new " + returnType + "();\n")
            .append("  smr.mapTypes(Constants.NS_URI_SOAP_ENC, new QName(\"\", \"Result\"),
null, null, sd);\n")
            .append("\n  SOAPHTTPConnection st = new SOAPHTTPConnection();\n")
            .append("\n  Call call = new Call();\n")
            .append("  call.setSOAPTransport(st);\n")
            .append("  call.setSOAPMappingRegistry(smr);\n")
            .append("\n  call.setTargetObjectURI(\"" + uri + "\");\n")
            .append("  call.setMethodName(\"" + methodName + "\");\n")
            .append("
call.setEncodingStyleURI(\"http://schemas.xmlsoap.org/soap/encoding/\");\n")
            .toString();

        paramsTmp = new StringBuffer().append("\n  Vector params = new Vector();\n");

        // generates SOAP call parameters
        Enumeration enum = param.elements();
        while(enum.hasMoreElements()) {
            Vector tmp = (Vector)enum.nextElement();
            //System.out.println("Size " + param.size());
            String type = (String)tmp.get(0);
            String value = (String)tmp.get(1);

            paramsTmp.append("  params.addElement(new Parameter(\"Value" + (counter++))
                            .append("\", " + type + ".class, \"" + value + "\", null));\n");
        }
        paramsTmp.append("  call.setParams(params)\n");

        params = paramsTmp.toString();

        // generates fault reponse
        fault = new StringBuffer().append("\n  if(resp != null && !resp.generatedFault())
{\n")
            .append("    Parameter ret = resp.getReturnValue();\n")
            .append("    Object value = ret.getValue();\n")
            .append("  } else {\n")
            .append("    Fault fault = resp.getFault();\n")
            .append("    System.err.println(fault.getFaultCode());\n")
            .append("  }\n}\n}")
            .toString();

        return method + params + fault;
    }
}
```

Appendix D - Test Walkthrough and Generated Code

Parts of the WSDL file of the Web Service used in the test scenario. Missing parts are denoted with "…".

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:s0="http://eratos.csse.monash.edu.au/banking"
targetNamespace="http://eratos.csse.monash.edu.au/banking"
xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <s:schema attributeFormDefault="qualified" elementFormDefault="qualified"
targetNamespace="http://eratos.csse.monash.edu.au/banking">

      <s:element name="NewAccount">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="l_name" nillable="true"
type="s:string" />
            <s:element minOccurs="1" maxOccurs="1" name="l_email" nillable="true"
type="s:string" />
            <s:element minOccurs="1" maxOccurs="1" name="l_country" nillable="true"
type="s:string" />
            <s:element minOccurs="1" maxOccurs="1" name="l_pwd" nillable="true"
type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>

      <s:element name="NewAccountResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="NewAccountResult" nillable="true"
type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="Logout">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="l_id" nillable="true"
type="s:string" />
            <s:element minOccurs="1" maxOccurs="1" name="session_pwd" nillable="true"
type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>

        …

    </s:schema>
  </types>

  <message name="NewAccountSoapIn">
```

```
    <part name="parameters" element="s0:NewAccount" />
  </message>
  <message name="NewAccountSoapOut">
    <part name="parameters" element="s0:NewAccountResponse" />
  </message>
  <message name="LogoutSoapIn">
    <part name="parameters" element="s0:Logout" />
  </message>
  <message name="LogoutSoapOut">
    <part name="parameters" element="s0:LogoutResponse" />
  </message>
        …

  <portType name="BankingServiceSoap">
    <operation name="NewAccount">
      <input message="s0:NewAccountSoapIn" />
      <output message="s0:NewAccountSoapOut" />
    </operation>
    <operation name="Logout">
      <input message="s0:LogoutSoapIn" />
      <output message="s0:LogoutSoapOut" />
    </operation>
    <operation name="Login">
      <input message="s0:LoginSoapIn" />
      <output message="s0:LoginSoapOut" />
    </operation>
    <operation name="Withdraw">
      <input message="s0:WithdrawSoapIn" />
      <output message="s0:WithdrawSoapOut" />
    </operation>
    <operation name="Deposit">
      <input message="s0:DepositSoapIn" />
      <output message="s0:DepositSoapOut" />
    </operation>
    <operation name="Balance">
      <input message="s0:BalanceSoapIn" />
      <output message="s0:BalanceSoapOut" />
    </operation>
  </portType>


  <binding name="BankingServiceSoap" type="s0:BankingServiceSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
    <operation name="NewAccount">
      <soap:operation soapAction="http://eratos.csse.monash.edu.au/banking/NewAccount"
style="document" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
    </operation>
    <operation name="Logout">
      <soap:operation soapAction="http://eratos.csse.monash.edu.au/banking/Logout"
style="document" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
    </operation>
```

```
    <operation name="Login">
        …
  </binding>

  <service name="BankingService">
    <port name="BankingServiceSoap" binding="s0:BankingServiceSoap">
      <soap:address location="http://eratos.csse.monash.edu.au/banking/banking.asmx" />
    </port>
  </service>
</definitions>
```

Generated XML instance document by the `SchemaMapper` class, see Appendix D for the schema class

```
<?xml version="1.0"?>

<ServiceDescription name="BankingService">
  <location>http://eratos.csse.monash.edu.au/banking/banking.asmx</location>
  <uri>http://eratos.csse.monash.edu.au/banking</uri>

<InterfaceDescription name="Banking">
  <method name="Withdraw>
    <param name="session_pwd type="string"/>
    <param name="l_amount type="decimal"/>
    <param name="userid type="string"/>
  </method>
  <method name="WithdrawResponse>
    <param name="WithdrawResult type="string"/>
  </method>
  <method name="NewAccount>
    <param name="l_email type="string"/>
    <param name="l_name type="string"/>
    <param name="l_pwd type="string"/>
    <param name="l_country type="string"/>
  </method>
  <method name="NewAccountResponse>
    <param name="NewAccountResult type="string"/>
  </method>
  <method name="Logout>
    <param name="session_pwd type="string"/>
    <param name="l_id type="string"/>
  </method>
  <method name="LogoutResponse>
    <param name="LogoutResult type="boolean"/>
  </method>
  <method name="Deposit>
    <param name="session_pwd type="string"/>
    <param name="l_amount type="decimal"/>
    <param name="userid type="string"/>
  </method>
  <method name="DepositResponse>
    <param name="DepositResult type="string"/>
    </method>
  <method name="Login>
    <param name="l_password type="string"/>
    <param name="l_id type="string"/>
  </method>
    <method name="LoginResponse>
    <param name="LoginResult type="string"/>
  </method>
  <method name="Balance>
    <param name="session_pwd type="string"/>
    <param name="userid type="string"/>
```

```
  </method>
  <method name="BalanceResponse>
    <param name="BalanceResult type="string"/>
    </method>
 </InterfaceDescription>
</ServiceDescription>
```

Generated Java interface by the `SchemaJava` class.

```
public interface Banking {
  public static final Sring url = "http://eratos.csse.monash.edu.au/banking/banking.asmx";
  public static final uri = "http://eratos.csse.monash.edu.au/banking";

  public String withdraw(String session_pwd, int l_amount);
  public String newAccount(String l_email, String l_name, String l_pwd, String l_country);
  public boolean logout(String session_pwd, String l_id);
  public String deposit(String session_pwd, int l_amount, String userid);
  public String login(String l_password, String l_id);
  public String balance(String session_pwd, String userid);
}
```

Generated interface implementation by the `SchemaJava` class. Only the implementation for the `balance` method is shown.

```
import java.io.*;
import java.util.*;
import java.net.*;
import org.w3c.dom.*;
import org.apache.soap.util.xml.*;
import org.apache.soap.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.rpc.*;
import org.apache.soap.transport.http.SOAPHTTPConnection;

public class SOAPBankingClient implements Banking {
  private URL url = new URL("http://eratos.csse.monash.edu.au/banking/banking.asmx");

  public SOAPBankingClient() throws Exception {
  }

  public String balance(String session_pwd, String userid) {

    SOAPMappingRegistry smr = new SOAPMappingRegistry();
    StringDeserializer sd = new String();
    smr.mapTypes(Constants.NS_URI_SOAP_ENC, new QName("", "Result"), null, null, sd);

    SOAPHTTPConnection st = new SOAPHTTPConnection();

    Call call = new Call();
    call.setSOAPTransport(st);
    call.setSOAPMappingRegistry(smr);

    call.setTargetObjectURI("http://eratos.csse.monash.edu.au/banking");
    call.setMethodName("balance");
    call.setEncodingStyleURI("http://schemas.xmlsoap.org/soap/encoding/");

    Vector params = new Vector();
```

```
    params.addElement(new Parameter("Value0", String.class, "session_pwd", null));
    params.addElement(new Parameter("Value1", String.class, "userid", null));
    call.setParams(params)

    Response resp = null;
    try {
      resp = call.invoke(url, "http://eratos.csse.monash.edu.au/banking");
    } catch(SOAPException e) {
        System.err.println(e);
    }

    if(resp != null && !resp.generatedFault()) {
      Parameter ret = resp.getReturnValue();
      Object value = ret.getValue();
    } else {
      Fault fault = resp.getFault();
      System.err.println(fault.getFaultCode());
    }

    return (String)resp;
  }
  …
}
```