

# Web Services Technologies

## SOAP vs. Jini

---

*Term Project*

Nadir Weibel (nad@nadnet.ch)  
Rudi Belotti (rbelotti@student.ethz.ch)

Prof. Dr. Moira C. Norrie

Supervisor: Andrea Lombardoni

Institute for Information Systems  
Group of Global Information Systems

Swiss Federal Institute of Technology - Zurich

Version 1.0  
*Last Update July 28, 2002*



## **Abstract**

Nowadays web services are becoming more and more important, especially in e-business. The underlying technology is relatively young. There are different emerging standards that address the problem of discovery and utilization of the resources available on a network. These standards tend to cover the same areas: almost all of them offer some kind of service discovery mechanism and a way to do a remote procedure call. The most important technologies seem to be Jini, developed by Sun as a Java technology, and SOAP, a W3C standard, with all the related components like WSDL and UDDI. The main goal of this thesis is to design a small system based on web services, and to implement it using different technologies. This project will highlight the relative strengths and weaknesses of the different standards used.



# Acknowledgments

Our thanks to the Institute of Global Information Systems, Prof. Moira Norrie and our supervisor, Andrea Lombardoni, for supporting us and providing a great working place and a wonderful and competent environment during the whole project.

— Nadir Weibel, Rudi Belotti



# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Web Services</b>	<b>5</b>
2.1	Web Service Architecture . . . . .	6
2.2	Web Service Technology Stack . . . . .	7
<b>3</b>	<b>SOAP</b>	<b>9</b>
3.1	XML . . . . .	9
3.2	SOAP . . . . .	13
3.3	WSDL . . . . .	14
3.4	UDDI . . . . .	18
3.5	Status of the Specification . . . . .	19
<b>4</b>	<b>Jini</b>	<b>21</b>
4.1	The Structure of Jini . . . . .	21
4.2	Remote Method Invocation . . . . .	22
4.3	Discovery and Join . . . . .	22
4.4	Finding and Using a Service . . . . .	24
4.5	Status of the Specification . . . . .	26
<b>5</b>	<b>Building a Real Web Services Environment</b>	<b>27</b>
5.1	The idea . . . . .	27
5.2	Setting up SOAP . . . . .	31
5.3	Setting up Jini . . . . .	32
5.4	The Environment . . . . .	33
5.5	Tests . . . . .	33

---

<b>6</b>	<b>SOAP Implementation</b>	<b>35</b>
6.1	Service Side . . . . .	35
6.2	Service Description . . . . .	35
6.3	Client Side . . . . .	38
6.4	UDDI Client . . . . .	40
<b>7</b>	<b>Jini Implementation</b>	<b>45</b>
7.1	Discovering the Lookup Service . . . . .	45
7.2	Publishing the Service (Service side) . . . . .	47
7.3	Looking up a service (Client side) . . . . .	52
7.4	Bind and use the service . . . . .	55
<b>8</b>	<b>Results</b>	<b>57</b>
8.1	Setup Time/Service Time . . . . .	57
8.2	Scalability . . . . .	59
8.3	Network Traffic . . . . .	61
8.4	Code Complexity . . . . .	63
<b>9</b>	<b>Conclusion and Future Works</b>	<b>65</b>
<b>A</b>	<b>Glossary</b>	<b>67</b>
A.1	General . . . . .	67
A.2	SOAP . . . . .	67
A.3	Jini . . . . .	68

# List of Figures

2.1	Web Service Environment . . . . .	5
2.2	Web Service Architecture . . . . .	6
2.3	Layering: web services vs. TCP/IP . . . . .	7
3.1	SOAP and its web service infrastructure . . . . .	10
3.2	XML Parser: DTD + Document . . . . .	11
3.3	Message path between a SOAP service and a SOAP client . . . . .	15
3.4	Hierarchy of built-in data types in XML Schema[1] . . . . .	16
3.5	How a simple Java class is described by means of WSDL . . . . .	17
3.6	UDDI distributed hierarchy . . . . .	19
4.1	The layers of the Jini technology . . . . .	22
4.2	The service provider searches the lookup service to register its services . . . . .	23
4.3	The service provider registers itself with the lookup service . . . . .	23
4.4	The client searches the lookup service to find the services it needs . . . . .	24
4.5	The client requests the service it needs . . . . .	25
4.6	The client communicates with the service . . . . .	25
5.1	A Pow Client getting the result of an exponentiation . . . . .	29
5.2	Flow diagram of the recursive operation distribution algorithm . . . . .	30
8.1	SOAP vs. Jini: setup and service time . . . . .	58
8.2	SOAP scalability: the multiplication service . . . . .	59
8.3	SOAP scalability: the exponentiation service . . . . .	60
8.4	Jini scalability: the multiplication service . . . . .	61
8.5	Jini scalability: the exponentiation service . . . . .	62



# Chapter 1

## Introduction

The world of web services is rapidly evolving. This involves a set of standards that promises great advantages for application integration and distributed computing.

The goal of this project is to analyze and compare two of these emerging technologies, i.e. W3C's SOAP[2] with its related complements WSDL[3] and UDDI[4] and Sun Microsystems' Jini[5][6]. The analysis is divided in two parts, the first part investigates the different concepts and philosophies of the different technologies, while the second part involves building a simple web services environment for each of the two technologies and measure their performance and their implementation complexity. With our performance measurements we intend to investigate:

- the *service discovery time*, i.e. the time required for a client to locate a service,
- the *service time*, i.e. the time required for a client to send a request to a service and get the answer,
- the *scalability*, i.e. how the service time varies depending from the number of available services,
- the *network traffic*, i.e. the number and the size of the network packets sent in order to perform a service request.

This report is structured as follows:

**Chapter 1** This introduction.

**Chapter 2** Presents the general web service concept and the components involved. Its goal is to familiarize with this interesting topic.

**Chapter 3** Focuses on SOAP and its related components WSDL and UDDI beginning with an introduction to XML, which is the basis of the whole technology.

**Chapter 4** Focuses on Jini explaining its concepts and the way it works, i.e. how we can publish, find or use a service.

**Chapter 5** Here we expose the details of the environment we have built: the basic idea, the software used and the hardware specification.

**Chapter 6** Contains an extract from our SOAP web services environment implementation, these are the base classes required to discover and invoke a service.

**Chapter 7** Contains an extract from our Jini web services environment implementation including the base classes to publish, discover and invoke a service.

**Chapter 8** Presentation of the measurements' results.

**Chapter 9** Conclusions and future works.

**Appendix A** A basic glossary which could help the reader to deal with the acronyms used.

For the implementation details we provide two files including all the classes we implemented for both SOAP and Jini environments, their names are *soap\_env.tar* and *jini\_env.tar* respectively.

## Chapter 2

# Web Services

Before beginning with the real prototype, it's useful to have a closer glance at the universe of *web services*, which are the basis of this project.

A *web service* (see figure 2.1) is a network accessible interface implementing the functionality of a remote application. To be reachable a web service is therefore built using standard Internet technologies [7].

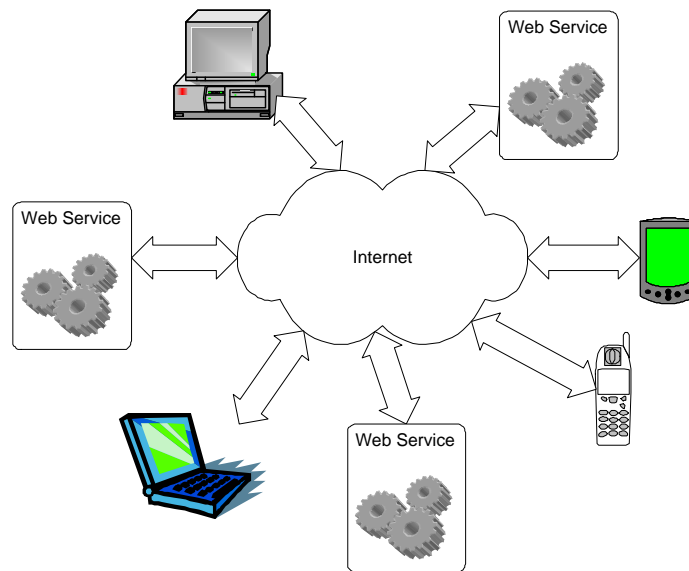


Figure 2.1: Web Service Environment

In other words every application which can be accessed through the internet using some communication protocol (HTTP [8], SMTP[9]) and some system of encoding (e.g. XML<sup>1</sup>) is a web service. Let's look at web services a little more in details.

Web services can be considered as *Middleware*, in other words, they are software interfaces placed between a client (who wants to use the service) and an application (which can serve the client and give it the desired service). A web service may take any form, can be used anywhere and may serve any purpose but it always acts as a *standardized abstraction layer*. This abstraction allows any client supporting the web service to get access to the service even if the application and the client are not

---

<sup>1</sup>see section 3.1

fully compatible (different languages, different environments, etc.).  
Now we know what is a web service, but how does it work?  
A web service consists of three fundamental components:

- a **Service Listener** to receive the message
- a **Service Proxy** to take that message and translate it into an action to be carried out (the invocation of a method or a function)
- an **Application Code** to implement the action

One of the most interesting features of web services is that they can be placed anywhere in the Internet universe, where it is possible to reach them (a web server, a PDA, a mobile phone,...) and anyone who has access to these services can use them; this introduces us to the *service paradigm* (“Everything is a service”). In practice, a client who wants to use a web service, has only to send its request to the *service listener* and wait for the answer. Requests between client and web service are usually invoked through TCP/IP[10][11], and transported over the Internet to the Web Application Server. The Application Server establishes a connection with the application and by means of a **Remote Procedure Call (RPC)**[12][13] collects all data and results necessary to send back an answer to the client.

## 2.1 Web Service Architecture

Once web services are available on the Internet we need a way to use them: in other words what we need is a kind of *Just-In-Time Integration*. We have to add an infrastructure in which it is possible to *provide*, to *publish*, to *find* and to *use* Services. This is called **Web Service Architecture** (see figure 2.2).

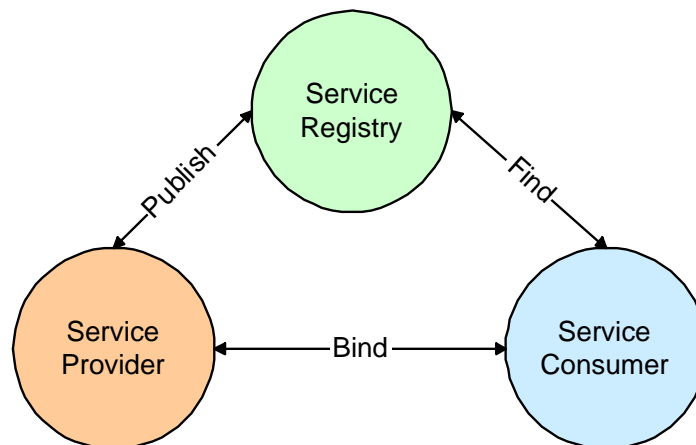


Figure 2.2: Web Service Architecture

The **Service Provider** publishes a description of the service that it offers via the **Service Registry**. The **Service Consumer** searches the Service Registry to find a Service that satisfies its needs. *Binding* refers to a Service Consumer actually using the service offered by a Service Provider.

## 2.2 Web Service Technology Stack

The web service architecture is based on a layered model very similar to the ISO/OSI[14] network model used to describe the architecture of Internet based applications. The components which play a role in this layered stack are essentially 5 (see figure 2.3).

Like all layered models (TCP/IP, ISO/OSI[15], etc.), the base idea is to have a different layer for each level of abstraction, so that each one performs a different and well defined function [15]. Every layer provides a set of *primitives* (operations) to the one above it and the above one can use them by knowing the interface that this layer implements, without knowing any detail of the real implementation. This allows the layers to be completely independent from each other and to be changed without influencing the whole system. The goal is the total modularization of the distributed computing environment, which is particularly important for web services, because of the rapidly evolving nature of the standards.

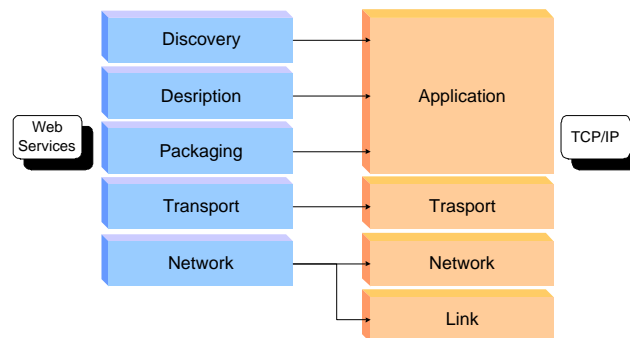


Figure 2.3: Layering: web services vs. TCP/IP

The key in the web service's architecture is to provide a *Just in Time Integration* and a platform-independent service; that is the reason why we need the *Packaging*, the *Description* and the *Discovery* layers.

- The **discovery** layer provides a mechanism to discover services offered by service providers, so that service consumer can effectively use them.
- The **description** layer of a service provides an interface where the consumer can learn about the network, packaging and transport protocols it will support.
- The **packaging** layer solves the problem of data integration and representation, i.e. it packages data in a format that can be moved around the network (by the transport layer) independently of the language or the platform using them (in Java we often say *serializing* or more generally *marshaling*). One of the most used serialization format is certainly *XML*.
- The **transport** layer enables direct application-to-application communication on top of the network layer and includes all those protocols such TCP, HTTP, SMTP or any other.
- The **network** layer provides the basic communication directives such as routing and addressing.



# Chapter 3

## SOAP

The *Simple Object Access Protocol*[2] (SOAP) is one of the most widely used technologies to implement web services. It's a lightweight protocol for exchanging information in a decentralized, distributed environment.

SOAP was originally developed at the World Wide Web Consortium (W3C)[16] on 1998 as a stand-alone module, subset of XML-RPC[17], just to codify how to send transient XML (eXtensible Markup Language<sup>1</sup>) documents and how to trigger operations or responses on remote hosts. With the development of XML by 1999/2000, it became clear that the serialization format needed by SOAP was XML. By the same time XMLSchema[18] was emerging as new interesting type definition language, the one needed by SOAP, so it also became part of the SOAP architecture.[19]

Today XMLSchema specifications are stable (and a proposed recommendation), W3C has founded a XML Protocol Working Group and we are close to have a *standardized metadata format* for SOAP. W3C members are currently working on SOAP Version 1.2 which is based on 3 working drafts (July 9, 2001; October 2, 2001; December 17 2001) produced by the *XML Protocol Working Group*.

As we said before SOAP is an *XML* based protocol and it basically consists of three parts:

- an **envelope**, that defines a framework and describes what a message is and how to process it
- a **set of encoding rules** for expressing instances of application-defined datatypes
- a **convention** to represent *Remote Procedure Calls* and responses

SOAP can potentially be used in conjunction with a variety of other protocols, but it is usually deployed in combination with HTTP and its extension framework [2].

It is useful to have a general idea of the infrastructure on which SOAP works in defining web services. There are different technologies that constitute that infrastructure: *UDDI* (section 3.4), *SOAP* (section 3.2), *WSDL* (section 3.3), *XML* (section 3.1), *HTTP* and *TCP/IP*.

The major goals of the SOAP environment are **simplicity** and **extensibility**.

### 3.1 XML

All SOAP messages are encoded using the *eXtensible Markup Language* (XML) and rely heavily on XML standards like *XMLSchema* and *XML Namespaces*[20]

---

<sup>1</sup>see section 3.1

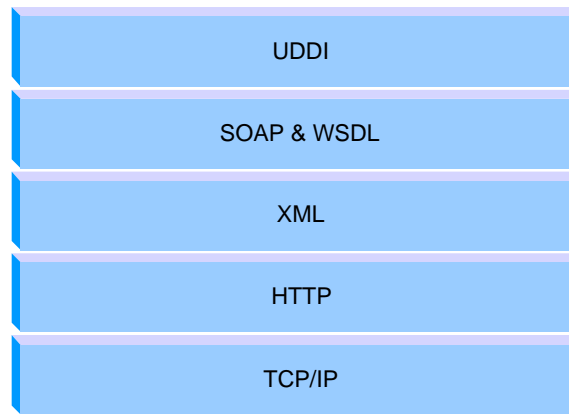


Figure 3.1: SOAP and its web service infrastructure

XML emerged in 1986 as a way to overcome the shortcomings of its two predecessors, SGML[21] and HTML[22] which were both very successful markup languages, but which were both flawed in certain ways. The discussion began on how to define a markup language with the power and extensibility of SGML but with the simplicity of HTML. W3C decided to sponsor a group of SGML gurus, including Jon Bosak from Sun, to develop a new standard implementing these needs: all of the non-essential, unused, cryptic parts of SGML were sliced away and the XML specifications was written. Over the next few years, XML evolved and a list of new features were added to this eXtensible Markup Language.

XML is fundamentally a markup language that allows the use of *application-specific document types* [23]. The main purpose of XML is to separate the presentation of a document from its content: what is really important in a document is the content, because it must communicate something. The presentation is client-dependent, it is bound to the specific software and the specific hardware that is trying to access the document. By means of XML structured documents and a styling language (such as the eXtensible Style Language[24]) it is possible to present the same content, to different clients (a web browser on a PC, a microbrowser on a PDA, a WAP[25] browser on a mobile phone, etc.) by means of a single document.

But what is the difference between XML and a standard markup language such as HTML?

Both XML and HTML are implementations of the *Standard Generalized Markup Language* (SGML). HTML defines a close class of document types, whereas XML defines a general data structuring language that can be used for structured information of any type. The main difference is that an HTML document defines both the content and its presentation (e.g. the tag <H1> implies the data is a title, and how it is presented on the screen), whereas XML defines only the content.

Structurally, an XML document is based on a *Document Type Definition* (DTD[26][23]), which defines the whole grammar of the markup language, by defining markup tags, the structure relationship between the tags, the sequence definition of the tags and all properties that can be associated with tags. In the case of an XML document, we have to give to the parser both the XML Document and the XML DTD (see figure 3.2)

Therefore XML can be considered as a kind of trade-off between the inflexibility of HTML and the complexity of SGML (for more information see [23]).

As we have seen before, document type definition (DTD) can contain all sort of user-defined tags. A simple DTD can for example be constructed to save entries in an addressbook:

```
<!DOCTYPE addressbook
```

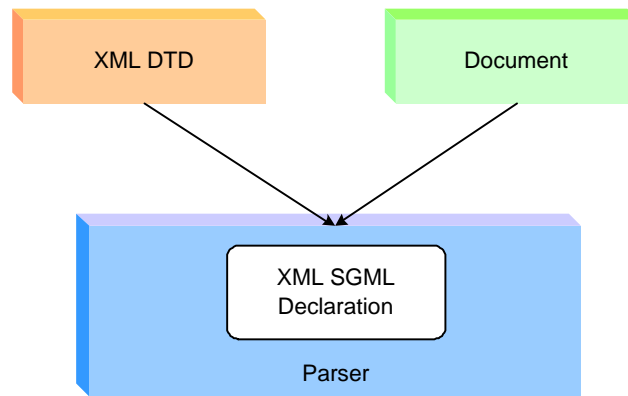


Figure 3.2: XML Parser: DTD + Document

```

SYSTEM "http://myhost.mydomain.com"
[
<!ELEMENT addressbook(item*)>
<!ELEMENT item(firstname+, lastname, address+, phone?, email?, www?)>
<!ELEMENT firstname #PCDATA>
<!ELEMENT lastname #PCDATA>

<!ELEMENT address(street?, no?, cap, city)>
<!ELEMENT street #PCDATA>
<!ELEMENT no #PCDATA>
<!ELEMENT cap #PCDATA>
<!ELEMENT city #PCDATA>

<!ELEMENT phone(fix+, mobile?, fax*)>

<!ELEMENT fix #PCDATA>
<!ATTLIST fix
  int CDATA #REQUIRED>

<!ELEMENT mobile #PCDATA>
<!ATTLIST mobile
  int CDATA #REQUIRED>

<!ELEMENT fax #PCDATA>
<!ATTLIST fax
  int CDATA #REQUIRED>

<!ELEMENT email #PCDATA>
<!ATTLIST email
  url CDATA REQUIRED>

<!ELEMENT www #PCDATA>
<!ATTLIST www
  url CDATA REQUIRED>
]>
  
```

In this example an addressbook item contains one or more firstnames, one lastname, one or more addresses (composed by optional street, optional number, cap and city), an optional phone record that must contain one or more fix numbers, an optional mobile number and zero or more fax numbers. Every phone number (fix, mobile, fax) must moreover specify an international prefix. The addressbook item completes itself with optional email and www address that also have to specify an attribute: the URL.

Given a DTD is now possible to create a conforming XML document, just by using the tags defined in the DTD:

```
<?xml version="1.0"?>
<!DOCTYPE addressbook SYSTEM "addressbook.dtd">
<addressbook>
  <item>
    <firstname>John</firstname>
    <lastname>McLaud</lastname>
    <address>
      <street>Main Street</street>
      <no>2</no>
      <cap>15254</cap>
      <city>San Francisco</no>
    </address>
    <address>
      <street>Broadway</street>
      <cap>11040</cap>
      <city>New York City</city>
    </address>
    <phone>
      <fix int="+001">012/457 66 55</fix>
      <mobile int="+001">045/654 22 11</mobile>
    </phone>
    <email url="mailto:john@mclaud.com">john@mclaud.com</email>
    <www url="http://www.mclaud.com">www.mclaud.com</www>
  </item>

  <item>
    <firstname>Frank</firstname>
    <firstname>Adam</firstname>
    <lastname>O'Connell</lastname>
    <address>
      <cap>45687</cap>
      <city>Edinburgh</cap>
    </address>
    <phone>
      <fix int="+52">042/468 99 54</fix>
    </phone>
    <email url="mailto:f.a.oconnell@ue.edu">f.a.oconnell@ue.edu</email>
  </item>
</addressbook>
```

This is only one of the possible applications of DTDs and XML; as its acronym indicates, XML is extensible.

At this point we focus on SOAP, one of the most important implementations of the XML philosophy.

In other words SOAP is XML, or better an application of it.

## 3.2 SOAP

SOAP works by exchanging XML-encoded messages. This is a very flexible way for applications to communicate because XML is not bound to a particular application, operating system or programming language and therefore it can be used in all environments. The fundamental idea is that two applications may openly share information using only a simple message encoded in a way that both applications understand. In order to have a common encoding method, we need to define some agreed conventions:

- the *type of information* being exchanged (encoded in the envelope)
- the *XML implementation* of that information (the encoding rules)
- the *sending paradigm* being used (the RPC representation)

The **envelope** consists of an optional *header*, containing information about the message processing (routing, delivery, authorization, etc.) and a mandatory *body*, containing the actual message. The XML syntax (encoding rules), necessary to encode a SOAP message is based on the W3C SOAP-defined namespace: <sup>2</sup>

The following example is a simple SOAP message consisting of Header and Body and representing a purchase order[7]:

```
<s:Envelope
  xmlns:s="http://www.w3.org/2001/06/soap-envelope">
  <s:Header>
    <m:transaction xmlns:m="soap-transaction" s:mustUnderstand="true">
      <transactionID>1234</transactionID>
    </m:transaction>
  </s:Header>
  <s:Body>
    <n:purchaseOrder xmlns:n="urn:OrderService"
      n:encodingStyle="http://www.w3.org/2001/06/soap-envelope">
      <from>
        <person>Christopher Robin</person>
        <dept>Accounting</dept>
      </from>
      <to>
        <person>Pooh Bear</person>
        <dept>Honey</dept>
      </to>
      <order>
        <quantity>1</quantity>
        <item>Pooh Stick</item>
      </order>
    </n:purchaseOrder>
  </s:Body>
</s:Envelope>
```

<sup>2</sup><http://schemas.xmlsoap.org/soap/envelope/>

The header entry is defined by an identifier, a name consisting of the namespace's URI and the local name (`s:Header`).

The SOAP `encodingStyle` attribute<sup>3</sup> can be used to indicate the encoding of the header. The `mustUnderstand` attribute requires that the recipient of the message understands how to process the message. Otherwise the recipient must discard the message.

The `encodingStyle` attribute defines a set of rules (defined at the given path) that applies to a particular set of XML elements.

An important class of SOAP messages are the *SOAP faults*. A SOAP fault is a message specifically used to communicate information about errors, that may have occurred during the processing of the SOAP message. SOAP faults communicate 4 kinds of information:

- **Fault code:** an ID for identifying the error (XML qualified name)
- **Fault string:** a human-readable explanation of the error
- **Fault actor:** the unique identifier of the message node at which the error occurred
- **Fault details:** they express application-specific details about the error

SOAP messages are essentially one-way, but the distributed architecture in which web services work, may introduce some intermediary between the sender and its recipient (see figure 3.3). A SOAP intermediary is a web service specially designed to sit between a service consumer and a service provider and add value or functionality to the transmission between the two.

The set of intermediaries that the message travels through is called *message path*. Every intermediary along that path is known as an *actor*[7].

The client is now ready to send the message and get its response through the defined web service. In order to do that, the client still needs an infrastructure for moving the message: the Internet. HTTP over TCP/IP is the most common transport protocol used to exchange SOAP messages because of its pervasiveness and its request/response behavior. The request will simply be posted to the server in the HTTP-request body and the response comes from the server in the HTTP-response body. The only difference is a newly defined `SOAPAction` HTTP header(see example) which is used to indicate the intent of the SOAP HTTP request.

```
POST /soap/servlet/rpcrouter HTTP/1.0
Host: myHost.myDomain:8080
Content-Type: text/xml
Content-Length: 655
SOAPAction: ""
```

```
.....
Body (Soap Request)
....
```

### 3.3 WSDL

The *Web Service Description Language* (WSDL[3]) is a support technology for the SOAP infrastructure, which describes the web services. In particular, WSDL allows automated code-generation tools to simplify building clients for existing web services. WSDL is the infrastructure that allows web

<sup>3</sup><http://schemas.xmlsoap.org/soap/envelope/>

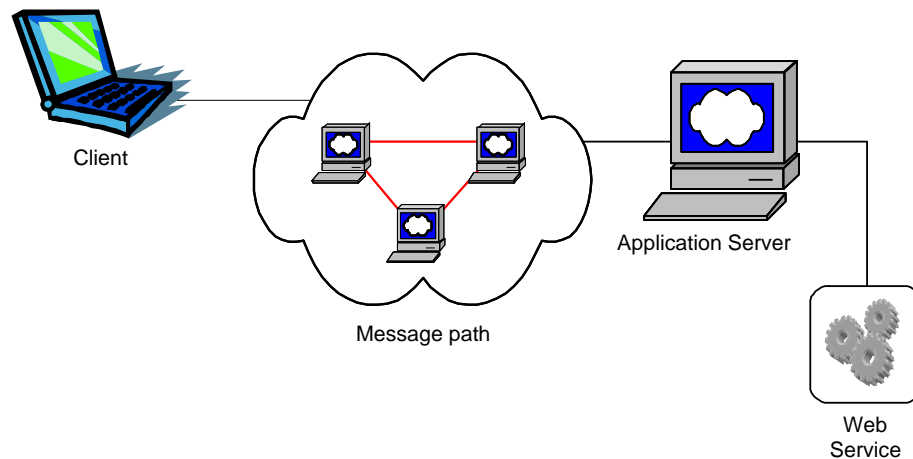


Figure 3.3: Message path between a SOAP service and a SOAP client

services to be *self-describing*. WSDL describes the *abstract interface* by which a service consumer communicates with a service provider, as well as the specific details of how a given web service implements the interface, by defining *data*, *messages interfaces* and *services*[7].

It is important to understand that WSDL is not directly a part of SOAP or of any web service infrastructure: it is an aid to implement this infrastructure and allowing it to be more integrated, more extensible and more useful. Although you can use SOAP without WSDL, a WSDL description of your services simplifies its use.

To allow seamless cross-platform interoperability, there must be a mechanism by which the service consumer and the service provider agree on a common set of types and data representation: XML, and in particular the W3C's *XMLSchema* specification (see figure 3.4), are the basis of this type and data specification sharing.

We can define a complex data type containing a collection of primitive data types: this example represents the type *telephoneNumber*, defined following the XMLSchema specifications:

```
<xsd:complexType name="telephoneNumber">
  <xsd:sequence>
    <xsd:element name="country">
      <xsd:simpleType>
        <xsd:restriction base=xsd:string">
          <xsd:pattern value="+\d{2}"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="number">
      <xsd:simpleType>
        <xsd:restriction base=xsd:string">
          <xsd:pattern value="\d{3}-\d{7}"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

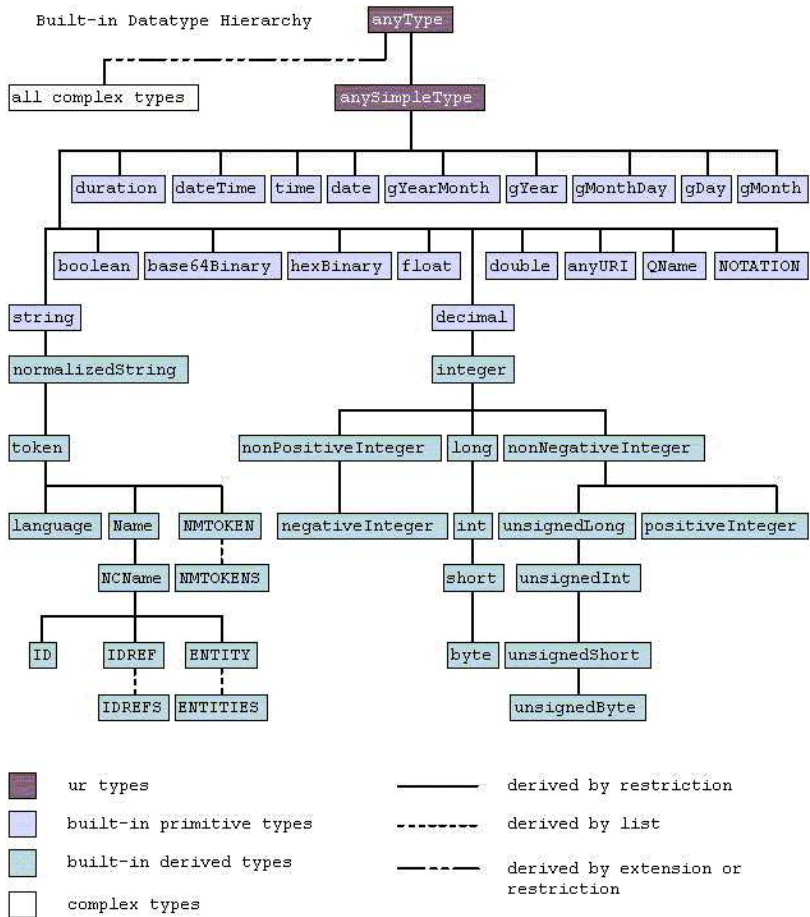


Figure 3.4: Hierarchy of built-in data types in XML Schema[1]

The data type looks in XML something like this:

```
<telephone xsi:type="abc:telephoneNumber">
  <country>+41</country>
  <number>001-5552244</number>
</telephone>
```

Once the data types are defined, they must be referenced within a WSDL description: it is either possible to embed the schema directly within the `<wsdl:types />` element or to import it using the `<wsdl:import />` element.

Web service interfaces are generally similar to object-oriented interfaces, in that they provide a sort of abstraction for implementing clients: there are input messages, output messages and fault messages. In WSDL, a web service interface is known as a *port type*. As in Java, COM or other OO languages, an interface must be implemented in order to be useful: in WSDL the keyword for implementation is *binding*.

There is still a detail that a WSDL service implementation must provide: the *network location* where the web service is implemented. This is obtained by linking a specific protocol binding to a specific network address in the WSDL `<wsdl:service />` and `<wsdl:port />` elements.

Figure 3.5 illustrates how the interface of a simple Java class providing a method to add two double numbers is described using WSDL (binding information and some other few details are omitted for simplicity).

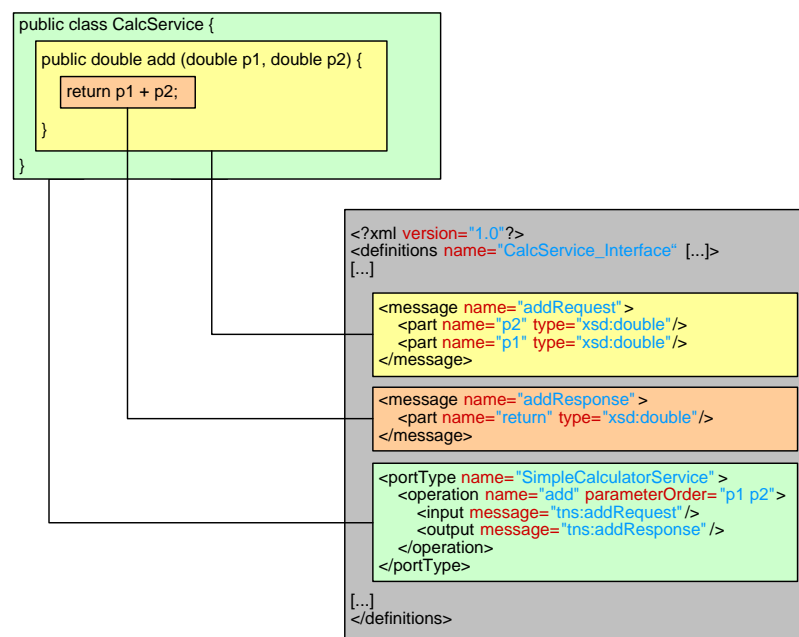


Figure 3.5: How a simple Java class is described by means of WSDL

## 3.4 UDDI

Once a WSDL description of a web service has been created, a service consumer must be able to use it. In other words, the client has to find the desired application; this process is known as *discovery*.

The *Universal Description, Discovery and Integration* (UDDI) specification[4] defines a way to publish and discover information about web services, by providing an interface that defines a simple framework for describing any kind of web service.

The specification consists of several related documents and an XML schema that defines SOAP-based programming protocol for registering and discovering web services[27]. The UDDI core is the *UDDI business registration*, an XML document used to describe a business entity and its web services. It consists of three parts:

- **“White pages”**, containing addresses, contacts and known web services identifiers
- **“Yellow pages”**, including industrial categorizations based on standards
- **“Green pages”**, the technical information about web services (references, URL, etc.) for manipulating and searching web services in the register

The UDDI *registry* allows a business to publish a description of its nature and the services it provides (see figure 2.2 and figure 3.6). A UDDI registry entry is composed of various parts:

- **Business Entity**: represents the provider which will publish the web service (“white and yellow pages”)
- **Business Services**: are defined in the business entity: they represent individuals web services provided by the Business
- **Binding Templates**: are within the service definition: they are the technical descriptions of the web services (“green pages”). A single business service may have multiple binding templates (e.g. for different protocols)
- **TModels**: represent a way for describing business, service and template structures stored in the UDDI registry. Any abstract concept (e.g. a new WSDL port type) can be registered within UDDI as a TModel

The UDDI environment is a global network in which there are many publishers and many consumers. To achieve the best results in registering, publishing and locating services, UDDI Registries have to be *distributed* and *federated* (see figure 3.6).

A UDDI registry is useless if we have no access to it. SOAP defines two interfaces for service consumers and service providers to interact with the registry. A service provider may publish its services by means of the `PublishSOAP` interface, and the clients can find a service using the `InquireSOAP` interface[7].

Once the WSDL-defined web service is published into a UDDI registry (i.e. by means of the `PublishSOAP` interface and a UDDI business definition), it is possible to build highly dynamic service proxies.

As an example let’s look at step-by-step definition and invocation of a service:

- The Publisher
  1. Writes the class (e.g. in Java)

2. Defines the WSDL interface
  3. Defines the UDDI business
  4. Publishes the service through UDDI and PublishSOAP
- The Consumer
    1. Locates the service in the UDDI registry (by means of the SOAP interface InquireSOAP)
    2. Accesses the WSDL description for service
    3. Invokes the service

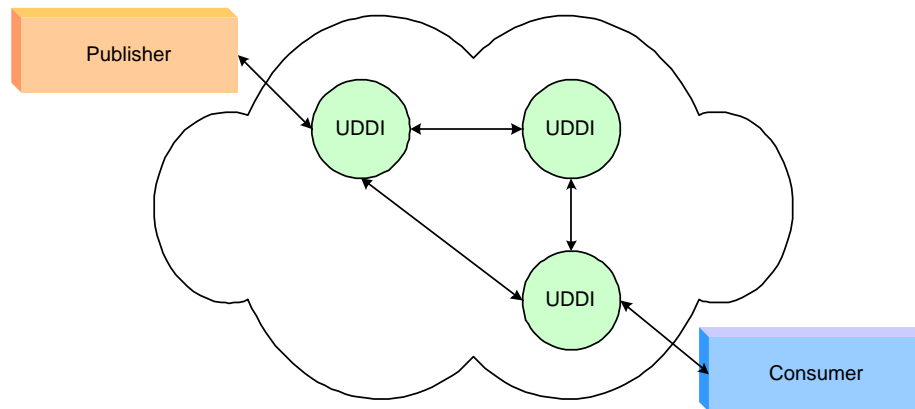


Figure 3.6: UDDI distributed hierarchy

### 3.5 Status of the Specification

SOAP is currently on version 1.2 (W3C Working Draft December 17, 2001) and W3C is working on the next draft. WSDL version 1.1 (W3C Note, March 15, 2001) is a stable release. For the UDDI Specification, the API is on version 2.0 (UDDI Open Draft Specification, June 8, 2001), but an errata (version 3) has been published on February 15, 2002.

An important change included in the errata is the possibility to search for services without specifying a businessKey. In the UDDI specification version 2.0 it is only possible to search for services within a given business entity, and finding all the services that implements a certain interface is more complicated because it requires more requests to the UDDI registry. With the new errata this should be simplified, alas at the moment we didn't find any implementation of this feature.



# Chapter 4

## Jini

The goal of the Jini[5] technology is the realization of a distributed computing environment that can support a rapid configuration with a “plug and play” model. This is a very important feature especially for spontaneous networks: in this kind of networks, the user does not want to spend a lot of time configuring it. By following the “plug and play” model, this activity is simplified and each component is automatically configured.

Jini is network based and follows the motto “the network is the computer”<sup>1</sup>. It is based on Java[28] and implemented in Java, that means it has an object oriented communication structure. The easiest way to speak to Jini’s remote services is using *RMI*[29]. Jini can be considered as an extension of the Java platform for distributed computing. The history of Java begun at the Sun Microsystems Labs in 1990 as a language called Oak. First in 1995 the Java language became public, whereas the Jini project remained hidden from public eyes until *New York Times* published an article on it in 1998. The Jini technology was introduced to the public on January 25, 1999 [30].

At first we will have an overview of the Jini structure and a short introduction to RMI; secondly we will have a closer look at the Jini technology, illustrating its main features and considering the classical example of a Jini enabled digital camera that wants to print some pictures on a Jini enabled printer. In this example the digital camera plays the role of the client and the printer that of the service provider.

### 4.1 The Structure of Jini

The Jini technology can be illustrated in a layered fashion as in fig. 4.1: the client (e.g. the digital camera) is on the left side, and wants to communicate with the right side where the service provider (e.g. the printing service) resides. On the top layer we find the application which communicates with the service using the service protocol and on the bottom layer there is obviously the physical transport on the network. Note that Jini does not require any particular operating system or network transport: infrared, radio or physical plugged network may be used without problems [31]. On the layers standing in between there is the Jini technology with its runtime environment giving the possibility to add, remove, find and use services and the Java RMI, which permits to invoke a remote method on the service side from the application side.

---

<sup>1</sup>Jim Waldo, chief architect for Jini technology (1986)

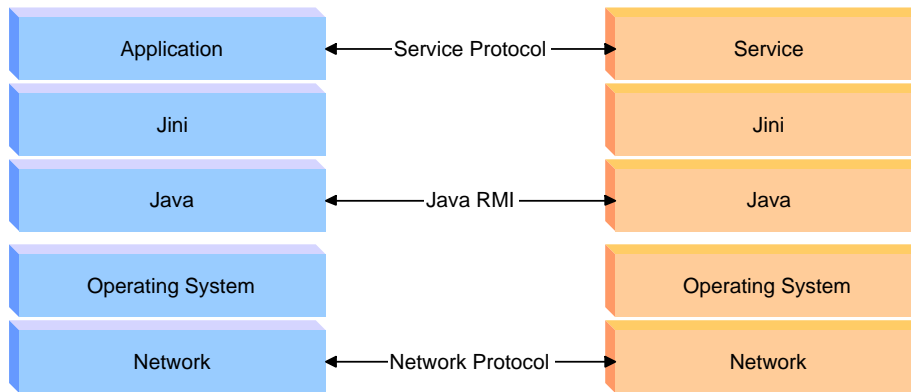


Figure 4.1: The layers of the Jini technology

## 4.2 Remote Method Invocation

The RMI allows the invocation of methods which reside on different Java Virtual Machines (JVM). This mechanism is similar to the remote procedure call (RPC) already used in SOAP. From the client perspective, the invocation of the remote method works as a standard method invocation. RMI calls are synchronous, i.e. the client which sends a request is blocked until the server gives an answer. From the server's point of view, a remote method invocation is seen as a callback method, i.e. the server registers the methods which are invoked by the client with the RMI infrastructure, and they are automatically called when the client's request arrives. By doing so, the programmer must only take care of the pure functionality of the methods and does not need to worry about the network connection and other details.

## 4.3 Discovery and Join

At this point, we will have a closer look at how Jini technology works. Before being used, all services have to be registered with the *lookup service* before being used, otherwise the client can't locate them. The lookup service acts as a repository, where services can be registered and can be found. As shown in fig. 4.2, the service provider uses the *discovery process* to find the lookup service in order to register its services. We notice that the lookup service is itself a service, thus there is a kind of "bootstrapping problem": how can we use this special service? For this purpose, Jini uses the discovery process, that simply sends out periodically messages on the network, i.e. multicast requests, so that it can find any lookup services in its environment.

After the service provider has located the lookup service, it issues a registration request (fig. 4.3) and sends its attributes to the lookup service, so that the clients can search for a more specific service, e.g. they can check if the printer is a color printer. The set of services on a particular network is referred to as a *Jini community*, or from the Jini specification *djinn*[32]. This process of registering the service is called the *join protocol*, i.e. joining the Jini community.

The process of adding a service to a lookup service is called *discovery and join*[33].

The registered service receives from the lookup service a *lease* and must constantly renew it to stay alive. For example, if the electric power fails, the printer can't renew its lease and gets automatically unregistered from the service repository. This makes the Jini environment *self-healing*<sup>2</sup>[32]. Once the service is registered, it is ready to be used.

<sup>2</sup>"Imagine if Internet search engines were able to do that and no longer pointed to dead links"[32]

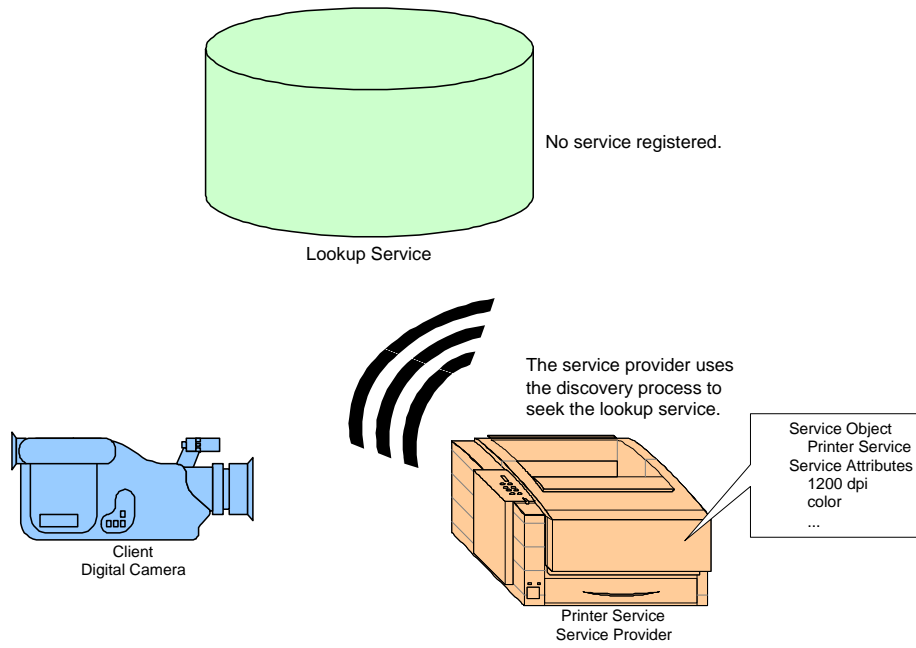


Figure 4.2: The service provider searches the lookup service to register its services

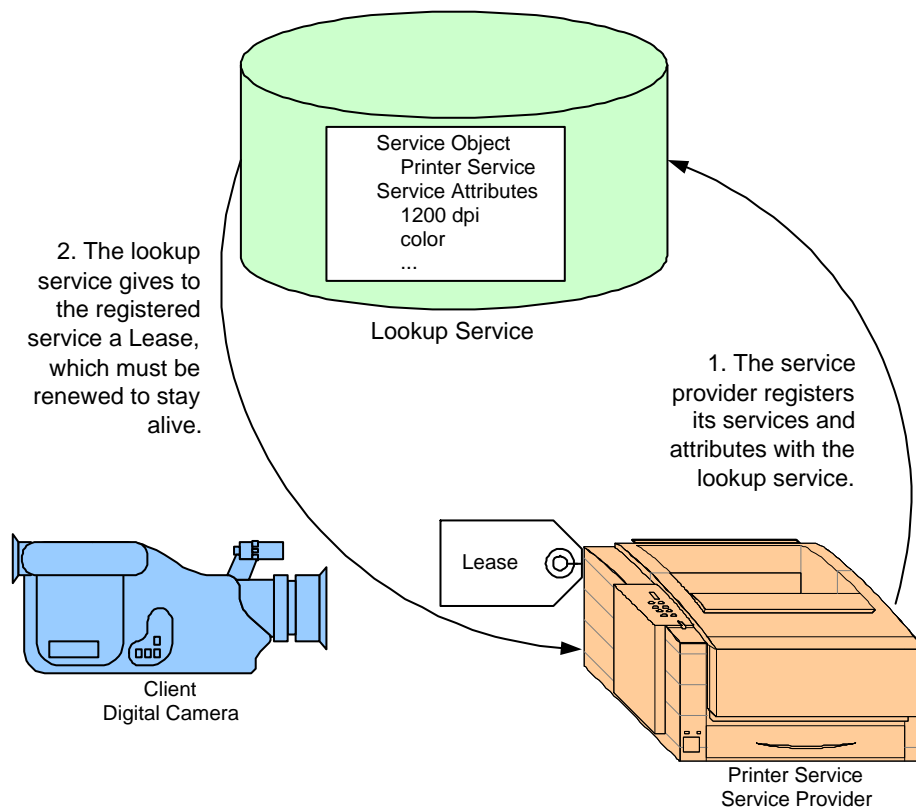


Figure 4.3: The service provider registers itself with the lookup service

## 4.4 Finding and Using a Service

The client, the digital camera in the example that we are considering, follows the same steps that the service provider did before in order to find the lookup service (fig. 4.4). If the camera itself offers some additional services, it would register them within the lookup service. In order to simplify, we assume that in this example the camera wants to use only some service.

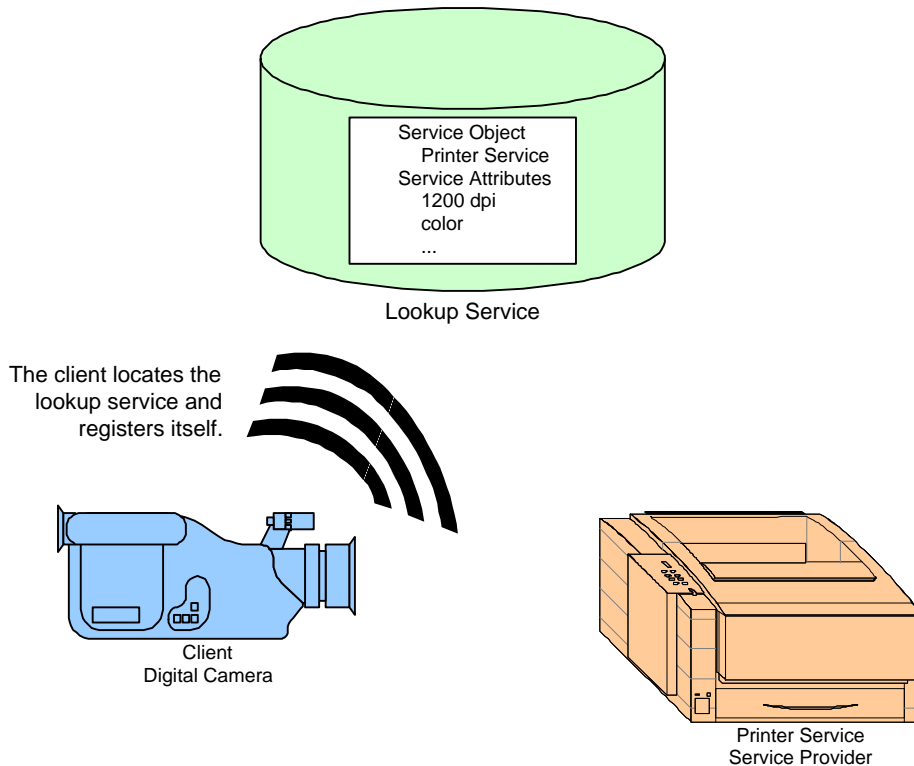


Figure 4.4: The client searches the lookup service to find the services it needs

After having located the lookup service, the client looks for the service it needs: in this case, it finds a printing service with a 1200 dpi color printer. At this point, the camera issues a request to use the service found and receives a proxy of the service object, as shown in fig. 4.5. The proxy object permits to hide all the details of the service's implementation, e.g. the client does not need to know how the printer drivers are implemented: it only needs the interface of the service it wants to use. The proxy allows the client to communicate directly with the service without knowing the details of its implementation. With the proxy, the client receives from the lookup service a lease, that defines how long the client can make use of the service. After the lease has expired the client must renew it to continue to use the service. We distinguish between *exclusive lease* and *non exclusive lease*: the first ensures that only one client can take a lease on the given resource, the second allows multiple clients to share the same resource[31].

Now the client is ready to use the service. Using the proxy of the service resource, it can communicate directly with the service object, and, in the example, the digital camera can send the pictures it wants to print to the printer (fig. 4.6).

This is how the Jini technology works. Note that our example assumes both the client and the service as being some piece of hardware, but this is not always the case. Jini makes no difference between hardware and software, both are considered as a service, i.e. you can have piece of software such as text editors or other utilities offered as service.

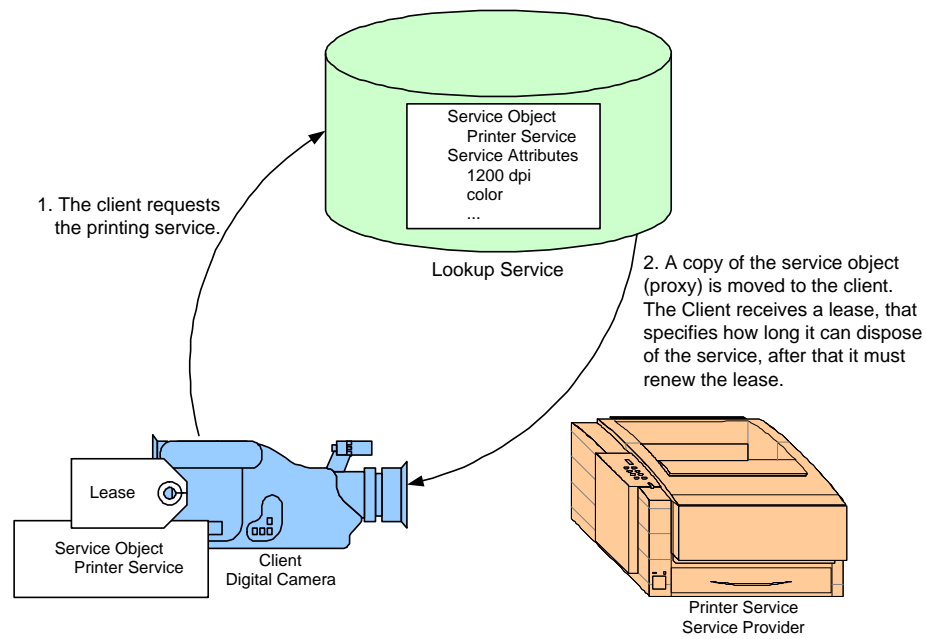


Figure 4.5: The client requests the service it needs

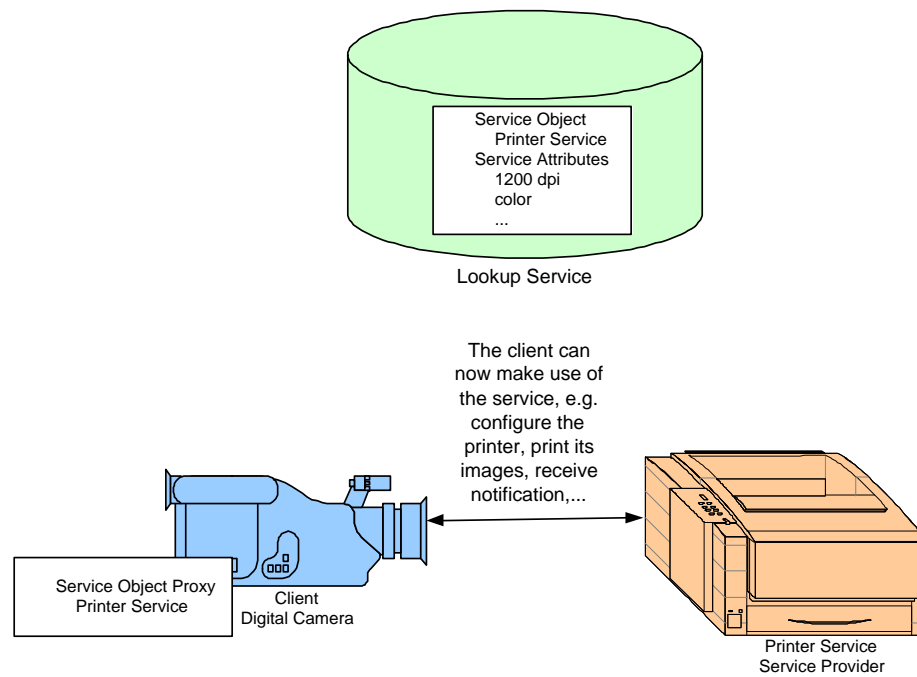


Figure 4.6: The client communicates with the service

## 4.5 Status of the Specification

Jini after the first stable releases 1.0 and 1.1 is now on version 1.2.1 (April 2002), both for the Architecture Specification and for the Core Platform Specification.

## Chapter 5

# Building a Real Web Services Environment

After having seen what web services are, focusing in particular on two important technologies like SOAP and Jini, it is now time to have a closer look on how really implement web services.

### 5.1 The idea

What we are interested in is essentially to highlight the relative strengths and weaknesses of both Jini and SOAP. The important part of our work is therefore focused on the infrastructure and the environment where web services are to be used rather than on the services themselves. For this reason we tried to develop some simple services so that they should not influence our test environment.

The services used and the web service architecture (see figure 2.2) are the same for both technologies, what is different is the way to publish, access and use them (for details please refer to chapter 6 and chapter 7). For both technologies we used Java[28] as underlying programming language for implementing the services and the clients using them, and two Service Registry (one for SOAP, one for Jini) located on the same host.

To compare the 2 technologies and see how they work and how good they are for implementing web services, we need to do some benchmarking. We measure essentially 6 different aspects (for the benchmark details see the tests [section 5.5] and the results [chapter 8]):

1. Setup Time for the clients and the services
2. Discovery Time for a client to locate the service registry
3. Scalability of the system (by adding more services or more clients)
4. System Saturation
5. Network Saturation
6. Implementation difficulty

The four service we have developed are essentially mathematical functions which return the result to their clients:

- An addition of doubles
- A subtraction of doubles
- A multiplication of integers
- An exponentiation of integers

In order to use a service, a client has first to locate the service through the service registry, getting the *Service Access Points*, and then it can finally access the service to get the answer.

All these services use a distributed paradigm: the multiply service calls several times the addition service to perform its operation, and the exponentiation service calls several times the multiplication service.

We can therefore divide our services in 2 classes:

- Pure service behavior (the addition)
- Service-Client distributed behavior (the multiplication and the exponentiation)

An example will clarify this hierarchically distributed operation (see also figure 5.1):

1. An exponentiation client wants to perform an exponentiation: it looks for an exponentiation service by inquiring the service registry and gets the relative access point.
2. The exponentiation client invokes the power service at the found access point to get the result.
3. The power service gets the request, but it alone is unable to perform the operation. It becomes a multiplication client, looks for multiplication services and gets the access points.
4. The power service distributes the multiplication to be performed within the discovered access points and invokes the relative services, waiting for answers.
5. The multiplication services get the requests, but they also are unable to perform the requested operations themselves, so they become addition clients, look for add services within the service registry and get the access points.
6. This time is the multiplication service which distributes the addition requests between the discovered access points and waits for the relative answers.
7. Finally the addition services get the addition requests, perform the operations and give back the results.
8. When the result of the last addition is known, the multiplication service can send back its result to the exponentiation service.
9. The exponentiation service waits for the last result coming from the multiplications and sends the final result to its client.

To distribute the multiplications and the additions among the appropriate services we implemented a recursive algorithm which first builds a binary tree of the necessary operation and then starts the requests, beginning from the leaf level and going backwards through the tree by starting a new service thread for each operation until the last operation (the root of the tree) is done (see figure 5.2). Unfortunately this algorithm blocks the upper level threads which have to wait for the 2 child threads to complete, and this is not good for the purpose of our benchmark.

Even if the prototype is working, for the reason we mentioned before, we decided to test another prototype, based on the same architecture and the same environment, but with a different implementation of the service distribution within the distributed Service-Client paradigm. Practically we just change the algorithm for distributing the operation in this way:

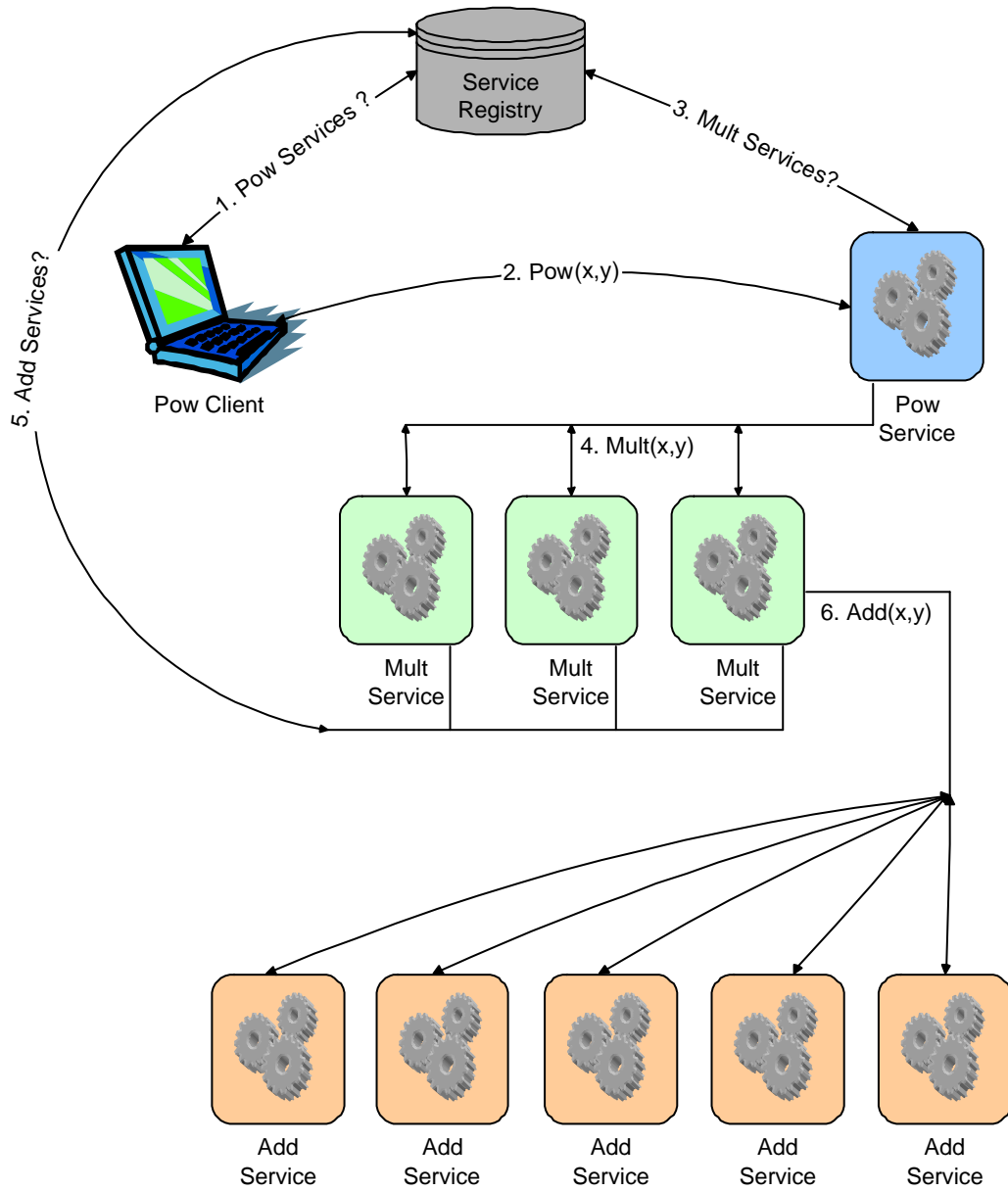


Figure 5.1: A Pow Client getting the result of an exponentiation

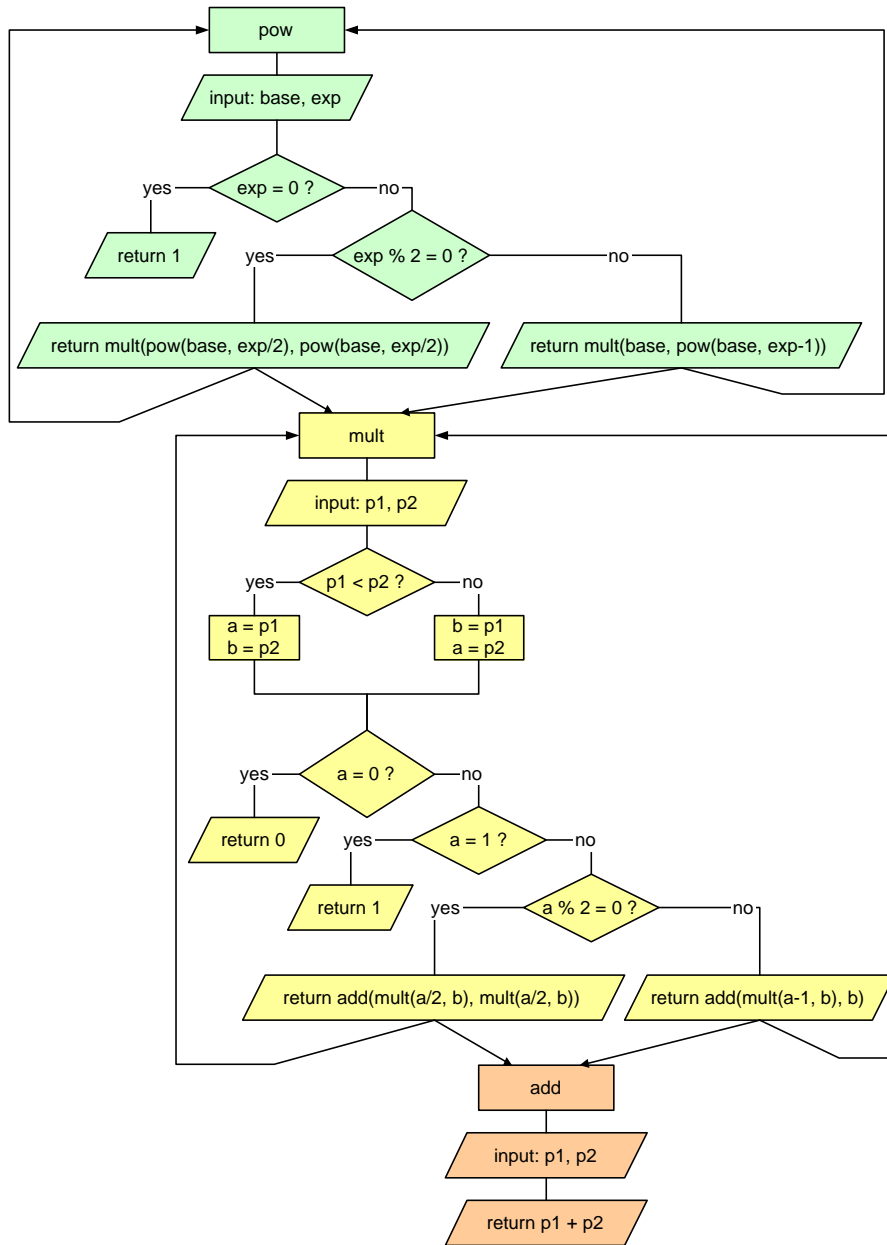


Figure 5.2: Flow diagram of the recursive operation distribution algorithm

- The multiplication service, receiving a request like  $\text{mult}(x,y)$ , just starts  $x$  parallel addition service threads each of them adding 2 random numbers.
- The exponentiation service, receiving a request  $\text{pow}(x,y)$ , just starts  $x$  parallel multiplication service threads which in turn start  $y$  parallel addition service threads.

The pseudo random numbers are derived for both technologies using the same algorithm and the same seed. This will ensure correctness in the choice of the data to be processed.

When the whole system is running is finally possible to start the test phase (see section 5.5) and get finally some result.

But let's now turn for a moment our attention on how to set up the two technologies for deploying and using webservices. As you will see this step is not so easy and straight-forward as one can think.

## 5.2 Setting up SOAP

To set up a SOAP web services infrastructure we need essentially three main components. The first component is a SOAP implementation, which allows to build and send SOAP calls, receive the answers and retrieve the results by parsing the SOAP answer message. The second component is a software which allows to access the deployed web services: in our case we used a servlet engine. The third component is the service registry, where the existing web services are registered to allows the clients to discover them. In general this registry uses a general database to store the service information.

The following software was used to build our SOAP infrastructure:

- SOAP
  - Apache SOAP 2.2<sup>1</sup>
- Servlet Engine
  - Apache Tomcat 4.0.3<sup>2</sup>
- UDDI
  - Systinet WASP UDDI 3.1 Standard<sup>3</sup>
  - Systinet WASP UDDI 4.0 Beta 1<sup>4</sup>
  - PostgreSQL 7.2.1<sup>5</sup> (needed by the UDDI to store its data)

We installed the Systinet WASP UDDI Beta implementation because the support service stated that with this release was possible to search for all services that implement a certain interface and not only for the services within a business entity as specified in the UDDI Version 2.0 API Specification Errata. Actually this is possible, but Systinet has not implemented this following the Specification Errata, so we didn't use this feature. However this implementation offers better access control, and seems sufficient stable, so we used this version for our tests because it should be the base for the future implementations. At the moment both implementations don't support the Java Virtual Machine version 1.4.0, therefore we used the version 1.3.1 (available at <http://java.sun.com>).

---

<sup>1</sup><http://xml.apache.org>

<sup>2</sup><http://jakarta.apache.org>

<sup>3</sup><http://www.systinet.com>

<sup>4</sup><http://www.systinet.com>

<sup>5</sup><http://www.pgsql.com>

## 5.3 Setting up Jini

Jini is a Java extension developed at Sun Microsystems for spontaneous networking. For this reason, developing Jini web services implies on one side the use of a Java environment and on the other side the Jini framework.

Prerequisites:

- Java 2 Standard Development Kit 1.4.0 (released February 2002)<sup>6</sup>
- Java 2 Platform API Specification, v1.4.0<sup>7</sup>
- Jini(TM) Technology Starter Kit v1.2.1 (released April 2002)<sup>8</sup>
- Jini(TM) Network Technology API Specification, v1.2.1<sup>9</sup>

Developing Jini services and clients involves a lot of Jini source classes. These are defined in the following jar libraries (included in the Jini Starter Kit):

- jini-core.jar
- jini-ext.jar
- sun-util.jar
- tools.jar

For each service we need an access point, which is essentially a little web server listening on a predefined port for incoming class requests.

We implemented it as a library modifying the class defined in the Jini tools.jar archive (so this file is not really needed if you use the library we developed).

The Service Registry, called Lookup Service in Jini, is already implemented and included in the Jini Starter Kit, it is called Reggie:

- reggie.jar
- reggie-dl.jar

In addition we defined an archive containing utility classes, needed for the benchmark, for initializing the web service, the operation distribution, etc. and a Jini archive containing the main service/client architecture classes:

- myUtil.jar
- myJini.jar

To set up the environment you just have to start the Jini Lookup Service, by calling the lookup script, and start some services of different type on different hosts but on the same subnet (by calling the relative scripts) so that they will be registered within the Service Registry. At this time you can start a client which will look up the Service Registry and give you access to the relative service.

<sup>6</sup>Downloadable at <http://java.sun.com/j2se/1.4/download.html>

<sup>7</sup><http://java.sun.com/j2se/1.4/docs/api/index.html>

<sup>8</sup>Downloadable at <http://www.sun.com/software/communitysource/jini/download.html>

<sup>9</sup><http://java.sun.com/products/jini/1.2.1/docs/api/>

If you want to implement new Jini services or clients you just have to extend the service and client class (in the myLib.jar library), compile the new defined classes and place them in the right directories, which will be exported by the web servers of the services.

You can find all the mentioned files in the jini\_env.tar archive. To use it just unzip the archive and then you can start the scripts or use the jar libraries.

Good Luck!

## 5.4 The Environment

The testing environment for our prototype is a Local Area Network (LAN) with the following specifications:

Network:

- 100 Mbits Ethernet (IEEE 802.3) connection
- Full switched LAN architecture

Hosts (Sun Ultra Sparc 60 Workstations<sup>10</sup>):

**CPU:** 450-MHz superscalar SPARC Version 9, UltraSPARC-II

**Cache:** 16-KB data and 16-KB instruction on chip, Secondary: 4-MB external

**Main Memory:** 512 MB (with 32-MB DIMMs, in pairs)

**Network adapter:** Ethernet/Fast Ethernet, twisted pair standard (10-BaseT and 100-BaseT)

**Operating System:** Unix Solaris 7 Operating Environment

## 5.5 Tests

The goal of the test phase is to have an idea of the differences in time and space between the two technologies (setup time, discovery time, network traffic, etc.). This inspection, should show how our prototype scales, how much it can be loaded (especially the Java Virtual Machine on which run the services) and what is the network load for the two technologies.

We tried to replicate the same tests on both SOAP and Jini, at the begin only for the one-step service of the multiplication and later also for the two-step service of the exponentiation.

The setup time and the discovery time are practically constant, because of the static position of the service registry and the implementation of the clients (for each client is the same). However is interesting to see the difference between SOAP and Jini in this environment.

For the scalability test, we have 4 variables which play an important role:

- The number of multiplication services currently running
- The number of addition services currently running
- The number of multiply service threads started by the exponentiation service
- The number of add service threads started by each multiply service

---

<sup>10</sup><http://www.sun.com/desktop/products/ultra60/>

For studying the scalability, we essentially changed these variables to observe the behavior of the prototype under different condition of load.

For the network traffic test, we measured the number of packets sent and received by each service and client of our prototype, and we generalized the result for each service request on the network (it works linearly).

The implementation complexity can be seen as a general observation on the length of the code and the real implementation differences between the two technologies.

The results of each of these observation can be found in the results chapter (chapter 8).

## Chapter 6

# SOAP Implementation

In this chapter we show the implementation details of our SOAP environment. We present an example of a simple service which performs addition and subtraction operations, its description by means of WSDL, a simple client accessing the service and an UDDI client (implemented using the Systinet's UDDI library) providing methods to find the services implementing a given interface.

### 6.1 Service Side

The service implementation is very simple and must only take care of the pure service functionality. The following example illustrates a method to add and one to subtract two double numbers.

```
public class ServiceExample {
    public double add(double p1, double p2) {
        return p1 + p2;
    }

    public double subtract(double p1, double p2) {
        return p1 - p2;
    }
}
```

The above class `ServiceExample` contains two methods `add` and `subtract`, that respectively add and subtract two numbers `p1` and `p2` of type `double`.

### 6.2 Service Description

SOAP web services are described using WSDL. There are two ways to write service descriptions: the simplest is to write both interface description and service location information in one single WSDL file, the other is to separate service interface description from service location in two different files. We choose the second way to illustrate how to describe the service of section 6.1. The following WSDL code describes the web service interface.

```
<?xml version="1.0"?>
<definitions name="ServiceExample_Interface"
```

```

targetNamespace="http://www.acme.org/wsd/ServiceExample_Interface"
xmlns:tns="http://www.acme.org/wsd/ServiceExample_Interface"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsd/soap"
xmlns="http://schemas.xmlsoap.org/wsd/">

<documentation>
  Standard WSDL service interface definition
  for the a simple calculator service
</documentation>

<message name="addRequest">
  <part name="p2" type="xsd:double"/>
  <part name="p1" type="xsd:double"/>
</message>

<message name="subtractRequest">
  <part name="p2" type="xsd:double"/>
  <part name="p1" type="xsd:double"/>
</message>

<message name="addResponse">
  <part name="return" type="xsd:double"/>
</message>

<message name="subtractResponse">
  <part name="return" type="xsd:double"/>
</message>

<portType name="SimpleCalculatorService">
  <operation name="add" parameterOrder="p2 p1">
    <input message="tns:addRequest"/>
    <output message="tns:addResponse"/>
  </operation>

  <operation name="subtract" parameterOrder="p2 p1">
    <input message="tns:subtractRequest"/>
    <output message="tns:subtractResponse"/>
  </operation>
</portType>

<binding name="SimpleCalculatorServiceBinding"
  type="tns:SimpleCalculatorService">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>

  <operation name="add">
    <soap:operation soapAction="http://www.acme.org/add"/>
    <input>
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:SimpleCalc" use="encoded"/>
    </input>

    <output>

```

```

        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="urn:SimpleCalc" use="encoded"/>
    </output>
</operation>

<operation name="subtract">
    <soap:operation soapAction="http://www.acme.org/subtract"/>
    <input>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="urn:SimpleCalc" use="encoded"/>
    </input>

    <output>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="urn:SimpleCalc" use="encoded"/>
    </output>
</operation>
</binding>
</definitions>

```

The definition begins by declaring the XML version, then in the `definitions` element we declare the `Namespaces` we are using and the `Namespace` of the current definition. The `documentation` element allows to include a natural language description of the interface. After that the technical part of the declaration begins.

With the `message` elements we describe all the input (request) and output (response) messages. The `name` attribute can be chosen freely (in general the method name plus message direction Request/Response is a good choice) while the `type` attribute correspond to a type definition, in this case we use the type `double` as defined in the XMLSchema specification. The `portType` element defines the whole interface, i.e. what operations (methods) are available within this interface by means of the above `message` declarations. At last the `binding` element specifies the encoding style and the `Namespace` of all operation.

The following description gives the service location information like the access point address.

```

<?xml version="1.0"?>
<definitions name="CalcService"
    targetNamespace="http://www.acme.org/wsdl/CalcService"
    xmlns:interface="http://www.acme.org/wsdl/CalcService_Interface"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

    <documentation>
        This service provides an implementation of a simple calculator service.
        It offers methods to add and subtract two numbers.
    </documentation>

    <import namespace="http://www.acme.org/wsdl/CalcService_Interface"
        location="http://www.acme.org/wsdl/CalcService_Interface.wsdl"/>

    <service name="CalcService">
        <documentation>Simple Calculator Service</documentation>

        <port name="CalcService"

```

```

        binding="interface:SimpleCalculatorServiceBinding">
        <documentation>Simple Calculator Service</documentation>
        <soap:address location="http://www.acme.org:8080/soap/servlet/rpcrouter"/>
    </port>
</service>
</definitions>

```

If we use two separate descriptions for the interface and the service location as in this example, for the latter we must import the interface definition: this is done using the `import` element. The service is described by the `service` element, the name in the `name` attribute and the access point is declared in the subelement `port` using the `location` attribute of the `address` element.

## 6.3 Client Side

For the client side implementation we used the Apache SOAP library, which provides classes allowing to build SOAP requests, and process SOAP responses. The following example illustrates a client that accesses a calculator service on the local machine (localhost) at the 8080 port (as is the case for Apache Tomcat with standard parameters) by calling its `add` method.

```

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

// These classes are included in the soap.jar file
// of the Apache SOAP implementation
import org.apache.soap.*;
import org.apache.soap.rpc.*;

public class Client {
    public static void main(String[] args) {
        try {
            // Call the service on localhost at port 8080
            URL url = new URL ("http://localhost:8080/soap/servlet/rpcrouter");

            // Get the parameters
            Double p1 = new Double(args[0]);
            Double p2 = new Double(args[1]);

            // Build the call using the Apache SOAP implementation
            Call call = new Call();

            // Set the target object name, i.e. the name from the service
            // deployment information
            call.setTargetObjectURI("urn:SimpleCalculatorService");

            // This sets which service method to call
            call.setMethodName("add");
            call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

            // Vector which contains the parameter to pass to the service
            Vector params = new Vector();

```

```

// Add parameters to the vector and specify their type
params.addElement(new Parameter("p1", Double.class, p1, null));
params.addElement(new Parameter("p2", Double.class, p2, null));

// Set the parameters for the call
call.setParams(params);

// make the call: note that the action URI is empty because the
// XML-SOAP rpc router does not need this. This may change in the
// future
Response resp = call.invoke(url, "" );

// Check the response
if ( resp.generatedFault() ) {
    // Get the fault informations
    Fault fault = resp.getFault ();
    System.out.println("The call failed: ");
    System.out.println("Fault Code   = " + fault.getFaultCode());
    System.out.println("Fault String = " + fault.getFaultString());
}
else {
    // Get the return value and print it to the screen
    Parameter result = resp.getReturnValue();
    System.out.println(result.getValue());
}
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

For the client side implementation we first need to import the required packages, i.e. `org.apache.soap` and `org.apache.soap.rpc`, that are included in the `soap.jar` file coming with the Apache SOAP implementation. The `java.net` package is also required for the `URL` class and the `java.util` package for the `Vector` class.

At fist we declare the access point address in the variable `url`, then using the `soap` package's classes we build the SOAP call. The functionality to do this is provided by the `Call` class:

- `setTargetObjectURI` sets the name of the service deployed. The name corresponds to the one of the SOAP deployment description.
- `setMethodName` sets the method's name of the service object we request.
- `setEncodingStyleURI` sets the used encoding style.
- `setParams` sets the parameters for the service object method. This takes as argument a `Vector` containing the parameters.

At this point the SOAP call is ready to be sent to the service: this is done by using the method `invoke` of the `Call` class, which returns the answer message. The class `Response` provides methods to check the answer message and get the service results. The method `generatedFault` checks if the service returned some kind of faults and if this is the case we can get the fault code and description by means of `getFaultCode` and `getFaultString` respectively. Otherwise the service results are

retrieved using the method `getReturnValue`: this method returns an object of type `Parameter` which provides methods to easily get the results in a standard format (e.g. `String`).

The above example illustrates how to access the web service if the access point is known: this was the starting point for our infrastructure. We implemented an advanced version which uses an UDDI client to dynamically find the access points for a certain service and then call the service at the access points found.

## 6.4 UDDI Client

The following code provides the basic methods to find all the services implementing a given interface. As stated in the section 5.2 with the current UDDI specification it is only possible to search for services within a business entity. To search for all the services implementing a certain interface we must search for all businesses which implement it and then search for the services within these business entities.

```
import java.util.Vector;

// these classes are included in the uddiclient.jar that
// comes with the Systinet's WASP UDDI implementation
import org.idoox.uddi.client.api.v2.request.inquiry.*;
import org.idoox.uddi.client.api.v2.request.publishing.*;
import org.idoox.uddi.client.structure.v2.tmodel.*;
import org.idoox.uddi.client.api.v2.response.*;
import org.idoox.uddi.client.structure.v2.base.*;
import org.idoox.uddi.client.structure.v2.business.*;
import org.idoox.uddi.client.structure.v2.binding.*;
import org.idoox.uddi.client.structure.v2.service.*;
import org.idoox.uddi.client.api.v2.*;
import org.idoox.uddi.client.*;

public class UDDIClient {

    private static String DEFAULT_HOST_URL =
        "http://munro.inf.ethz.ch:8082/uddi/inquiry";

    private String host_url = DEFAULT_HOST_URL;
    private UDDIApiInquiry inquiry = null;

    /**
     * Get a single instance of the UDDIApiInquiry
     * @return the UDDIApiInquiry
     * @throws Exception
     */
    public UDDIApiInquiry getInquiry() throws Exception {
        if (inquiry == null) {
            // This will retrieve a stub to the UDDI inquiry port.
            inquiry = UDDILookup.getInquiry(inquiryHost);
        }
        return inquiry;
    }

    /**
```

```
* Find binding details for a given service (access point)
* @param serviceKey
* @param tModelKey
* @return BindingDetail
* @throws Exception
*/
public BindingDetail findBinding (String serviceKey, String tModelKey)
throws Exception {
    // Create a FindBinding instance.
    FindBinding findBinding = new FindBinding();

    // Set the service UUID to use in the query.
    findBinding.setServiceKey(new ServiceKey(serviceKey));

    // Set the tModel UUID to use in the query.
    findBinding.addTModelKey(new TModelKey(tModelKey));

    // Send the message and retrieve the response.
    BindingDetail bindingDetail = getInquiry().find_binding(findBinding);

    // Show the results.
    return bindingDetail;
}

/**
 * Find the list of all the businesses that implements a given interface
 * (specified in the tModel)
 * @param tModelKey
 * @return BusinessList
 * @throws Exception
 */
public BusinessList findBusinessByTModel (String tModelKey) throws Exception {
    // Create a FindBusiness instance.
    FindBusiness findBusiness = new FindBusiness();

    // Set the binding UUID to use in the query.
    findBusiness.addTModel(new TModelKey(tModelKey));

    // Send the message and retrieve the response.
    BusinessList businessList = getInquiry().find_business(findBusiness);

    // Show the results.
    return businessList;
}

/**
 * Find the list of tModels with a given name
 * @param name the name of the tModel
 * @return TModelList
 * @throws Exception
 */
public TModelList findTModel (String name) throws Exception {
    // Create a FindTModel instance.
    FindTModel findTModel = new FindTModel();
```

```

// Set the binding UUID to use in the query.
findTModel.setName(new Name(name));

// Send the message and retrieve the response.
TModelList tModelList = getInquiry().find_tModel(findTModel);

// Show the results.
return tModelList;
}

/**
 * Get all the access points for a given service that implements
 * a certain interface defined by the tModel
 * @param tModelName
 * @return Vector of all access points
 * @throws Exception
 */
public Vector getAccessPointVectorForService(String tModelName)
throws Exception {
    String tModelKey;
    String serviceKey;
    TModelInfos tModelInfos;
    BusinessInfos businessInfos;
    BusinessInfo businessInfo;
    ServiceInfos serviceInfos;
    ServiceInfo serviceInfo;
    BindingTemplates bindingTemplates;
    BindingTemplate bindingTemplate;

    Vector accessPointVector = new Vector();

    // 1st call: find tModel uuid from its name
    tModelInfos = findTModel(tModelName).getTModelInfos();
    tModelKey = tModelInfos.getFirst().getTModelKey().getValue();

    // 2nd call: find all the businesses that implement the tModel
    businessInfos = findBusinessByTModel(tModelKey).getBusinessInfos();
    businessInfo = businessInfos.getFirst();

    while (businessInfo != null) {
        serviceInfos = businessInfo.getServiceInfos();
        serviceInfo = serviceInfos.getFirst();

        while (serviceInfo != null) {
            serviceKey = serviceInfo.getServiceKey().getValue();

            // 3rd call: find the binding informations
            bindingTemplates = findBinding(serviceKey, tModelKey).getBindingTemplates();
            if (bindingTemplates != null) {
                bindingTemplate = bindingTemplates.getFirst();

                while (bindingTemplate != null) {
                    accessPointVector.addElement(bindingTemplate.getAccessPoint().getValue());
                }
            }
        }
    }
}

```

```
        bindingTemplate = bindingTemplates.getNext();
    }
}
    serviceInfo = serviceInfos.getNext();
}
    businessInfo = businessInfos.getNext();
}

    // return the service access point vector
    return accessPointVector;
}
}
```

All the imported packages `org.idoox.uddi.client` are included in the file `uddiclient.jar` contained in the Systinet's WASP UDDI implementation. For our example we used the UDDI specification version 2. We need to know the UDDI repository's address, this is defined in the `String` variable `DEFAULT_HOST_URL`. There are two basic APIs related to UDDI: the *inquiry API* and the *publish API*. In this example we focus only on the first, the second can be used in analogous way.

The method `getInquiry` returns a single instance of the inquiry API, which is then used by all the methods to perform requests to the UDDI registry. All methods we implement work in the same way: they build the UDDI call and then they inquire the registry, which in general returns a list of matching objects. As we have already seen, UDDI messages are XML messages. Essentially for each message element there is a corresponding Java class defining all its properties. This makes the work quite easy.

At this point we analyze the `findBinding` method in details, the others all work in a analogous way. To find a service binding we need to pass two parameters to the UDDI registry, these parameters are the service key and the TModel key and are defined in `String` format. The class `FindBinding` is what we need to inquire the registry, this requires the service key and TModel key information to be set. For this purpose we use the methods `setServiceKey` and `addTModelKey`. At last we call the `UDDIApiInquiry`'s method `find_binding` to perform the request. It returns a `BindingDetail` object which contains the information we need and the methods to get these information in a useful format.



# Chapter 7

## Jini Implementation

Implementing Jini services and clients requires several fundamental steps: first of all both service and client, have to discover a lookup service. As a second step they have to publish services, or search for services. Finally they should be able to respond to service requests, respectively use the service.

The whole process involves several classes. In the next sections we will look at these classes more in detail.

### 7.1 Discovering the Lookup Service

In this phase both client and service need to discover a lookup service within the same subnet. This is done by implementing the jini `DiscoveryListener` class. This class defines two methods: `discovered` (invoked if a new service registry is found) and `discarded` (called to explicitly discard a lookup service from the list of the available registries):

```
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceRegistration;
import java.util.HashMap;

// Instantiated by the classes which need to locate a lookup service
public class MyDiscoveryListener implements DiscoveryListener{

    protected ServiceInterface Service; // The Service instance
    protected ClientInterface Client; // The Client instance
    protected String type; // client/service

    // Constructs a new object for the service
    public MyDiscoveryListener(ServiceInterface service, String type){
        this.service=service;
        this.type=type;
    }

    // Constructs a new object for the client
    public MyDiscoveryListener(ClientInterface client, String type, int nofService){
```

```

    this.client=client;
    this.type=type;
    this.nofService=nofService;
}

// Container for the registered Lookup Services
protected HashMap registrations = new HashMap();

// Called if a Discovery Event occurs by the Discovery EventListener
public void discovered(DiscoveryEvent ev) {

    // Saves the found Lookup Services in the array
    ServiceRegistrar[] newregs = ev.getRegistrars();

    // Gets the lookup service and invokes the relative methods on clients and services
    for (int i=0 ; i<newregs.length ; i++) {

        if (!registrations.containsKey(newregs[i])){

            try{

                // Gets the Host where the lookup service is running
                String lookupName=newregs[i].getLocator().getHost();
                System.out.println("discovered a lookup service [" + lookupName + "!");
                registrations.put(newregs[i],null);

            }catch(Exception e){
                System.out.println("Problem Contacting the lookup: " + e.getMessage());
            }

            if(type.equals("service"))

                // Calls the method to publish the services on the service class
                Service.registerWithLookup(newregs[i]);
            else

                // Calls the method to look up a service on the client class
                Client.lookupForService(newregs[i], nofService);
        }
    }
}

// Called when we explicitly discard a lookup service, not "automatically" when a
// lookup service goes down. Once discovered, there is NO ongoing communication
// with a lookup service
public void discarded(DiscoveryEvent ev) {
    ServiceRegistrar[] deadregs = ev.getRegistrars();
    for (int i=0 ; i<deadregs.length ; i++) {
        registrations.remove(deadregs[i]);
    }
}
}

```

As you can see, once the lookup service is discovered, the discovery listener calls the method to publish the service (`registerWithLookup`, defined in the service class) or the method to look for the desired service (`lookForService` defined in the client class).

## 7.2 Publishing the Service (Service side)

The first step in publishing a service is defining the interface for the service we are publishing. This interface will be used (and implemented) on the service side to give access to the service and on the client side to access the service. The interface for a specific service extends the more general `ServiceInterface` with the methods to access the service:

```
// A general interface for defining Services
public interface ServiceInterface {

}

// The specific service interface of one service
public interface oneServiceInterface extends ServiceInterface {
    public Object doSomething(params);
}
```

After having defined the interfaces, we have to write the proxy to be registered within the lookup service and to be sent to the client, with the functionality to access the service. The proxy implements the `ServiceInterface` interface so that the client can use this interface directly by using the `doSomething` method, and the `Serializable` interface, because it is an object to be sent over the network (marshaling problem):

```
import java.io.Serializable;
import java.rmi.Remote;
import java.rmi.RemoteException;

public class ServiceProxy implements ServiceInterface, Serializable{
    protected BackendProtocol backend;

    public ServiceProxy() {
    }

    public ServiceProxy(BackendProtocol backend){
        this.backend = backend;
    }

    // Note: doSomething() calls backend.useTheService()
    public Object doSomething(params){
        Object result;
        try {
            result=backend.useTheService(params);
        } catch(RemoteException ex) {
            System.out.println( "Couldn't contact backend: " + ex.getMessage());
        }
        return result;
    }
}
```

```
    }
}
```

The Proxy uses a back-end protocol to communicate with the service: this protocol is defined in the `BackendProtocol` interface and will be implemented by the service to execute the client requests. The method `useTheService` is called by the proxy to get access to the real method on the server side. The interface extends `Remote`, because it works over RMI and will generate RMI stubs:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface BackendProtocol extends Remote {
    public double useTheService(params) throws RemoteException;
}
```

We are now ready to publish our service proxy into the lookup service. It is time to define the `registerWithLookup` method to register a service within a lookup which is called by `MyDiscoveryListener` and the whole service.

The `Service` class defines the real functionality of all services. This class extends the Java class `Thread` so that an instance of it is always active. The proxy definition and the particular requirements of the specific services will be defined in the classes extending it.

```
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.core.lookup.ServiceItem;
import net.jini.lookup.entry.ServiceInfo;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lease.Lease;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Iterator;
import java.util.ArrayList;
import java.util.HashMap;
import java.net.InetAddress;
import java.rmi.RMISecurityManager;
import java.io.*;

public class Service extends Thread{

    //Lease Time set to 5 minutes
    protected final int LEASE_TIME = 5 * 60 * 1000;
    //The object containing the proxy to be sent to the lookup service
    protected ServiceItem item;
    //The container storing the discovered Lookup Services
    protected HashMap registrations = new HashMap();
    //The class managing the discovery process
    protected LookupDiscoveryManager discoverymanager;
    //The address of the web server exporting the code for the service
    protected String codeBase;

    // Constructs a new Service object by defining the codebase
```

```
// It sets the security policy.

public Service(String port, String policy) throws IOException {
    String localhost=getLocalHost();
    this.codeBase="http://" + localhost + ":" + port + "/";

    // Defines the System property (you can also use the -D option in the shell)
    System.setProperty("java.security.policy" , policy);
    System.setProperty("java.rmi.server.codebase", codeBase);

    // Set a security manager.
    if (System.getSecurityManager() == null)
        System.setSecurityManager(new RMISecurityManager());
    }
}

// Sets the serviceItem to be sent to the Lookup Service for this service
protected void setServiceItem(ServiceItem item){
    this.item=item;
}

// Registers the service within the lookup service
public synchronized void registerWithLookup(ServiceRegistrar registrar) {
    ServiceRegistration registration = null;
    Lease L = null;
    try {
        // The service item is registered within the lookup service.
        // It will be defined in the classes extending this general class
        registration = registrar.register(item, LEASE_TIME);
    } catch (RemoteException ex) {
        timestamp.println("Couldn't register: " + ex.getMessage());
        return;
    }
    // If this is the first registration, use the service ID returned to us.
    // Ideally, we should save this ID so that it can be used after
    // restarts of the service.
    if (item.serviceID == null) {
        item.serviceID = registration.getServiceID();
        timestamp.println("Set serviceID to " + item.serviceID);
        timestamp.println("Set Lease Time to " + LEASE_TIME/60/1000 + " min");
        timestamp.println("Waiting for client requests...");
    }
    // Store the registered service into the container
    registrations.put(registrar, registration);
}

// Gets the name of the local host as a String object
protected String getLocalHost(){
    InetAddress host = null;
    try{
        host = InetAddress.getLocalHost();
    }
    catch (Exception e){
        timestamp.println("Exception: " + e.getMessage());
    }
}
```

```

    }
    String name=host.getCanonicalHostName();
    return (name);
}

// Starts the thread and sleeps to maintain the service alive
public void run() {
    while (running) {
        try {
            long sleepTime = LEASE_TIME - (20*1000);
            Thread.sleep(sleepTime);
            this.renewLeases();
        }catch (InterruptedException ex) {}
    }
}
}
}

```

The next step is defining a specific service class extending the general service class we have just seen. This class sets the proxy and the serviceItem (by calling the `createProxy` and the `setServiceItem` classes), starts the discovery process (by creating a `DiscoveryListener` object) and initializes the real service (the `calc` class). The inner class `Backend` implements the backend protocol and it is the class listening for incoming client requests. For this reason this class extends `UnicastRemoteObject` and will generate the RMI Skeleton. This class returns a value to the client and is the class really invoking the external service.

```

import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.core.lookup.ServiceItem;
import net.jini.lookup.entry.ServiceInfo;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RMISecurityManager;
import java.io.*;

public class oneService extends Service{

    private Object calc; //The object which will perform the service

    public oneService(String port, String policy) throws IOException {

        //Call the common Super class Service
        super(port, policy);

        // Creates an instance of the service proxy
        oneServiceInterface proxyobj = createProxy();

        // Creates a service item to be added to the lookup service
        ServiceInfo [] SInfo = new ServiceInfo[1];
        ServiceInfo S = new ServiceInfo(
            "One Service", "nadnet.ch", codeBase, "v1.0", "backend", "002");
        SInfo[0]=S;

        //Set the service Item with the created proxy

```

```

    setServiceItem(new ServiceItem(null, proxyobj, SInfo));

    // Creates a discovery Listener
    DiscoveryListener myListener = new MyDiscoveryListener(this, "service", timestamp);

    // Searches for the "public" group, which by convention is named by the empty string.
    discoverymanager = new LookupDiscoveryManager(new String[] { "" }, null, myListener);

    //Instantiates the service
    calc = new Object();
}

class Backend extends UnicastRemoteObject implements BackendProtocol {

    // Nothing for the constructor to do but let the
    // superclass constructor run.
    Backend() throws RemoteException {
    }

    // This method is executed locally and only the result is passed to RMI
    public Object useTheService(params) throws RemoteException{
        Object result=calc.doSomething(params);
        return (result);
    }
}

// The Proxy: creates a remote server object that will receive method
// invocations from the proxy, and creates a new Proxy object that refers to it.
protected oneServiceInterface createProxy() {
    try{
        Backend backend = new Backend();
        ServiceProxy proxyobj=new ServiceProxy(backend);
        return (oneServiceInterface)proxyobj;
    }catch (RemoteException ex) {
        ex.printStackTrace();
        System.exit(1);
        return null;// NOT REACHED
    }
}

// Main creates the wrapper
public static void main(String args[]) {

    String port="1234";
    String policy="policy.all";
    String dir="dir";
    try{
        //starts the web server exporting the required classes
        Thread WSthread =
            new Thread(new WebBrowser(Integer.parseInt(port), dir, false, true));
        WSthread.start();
        new oneService(port, policy).start();
    }catch (IOException ex) {
        System.out.println("Couldn't create service: " + ex.getMessage());
    }
}

```

```

    }
  }
}

```

### 7.3 Looking up a service (Client side)

Once a service is registered and ready to be used, a client can use it. The first step for the client is locating the lookup service (see section 7.1); once discovered, the client must look up the service registry for the desired service (the method `lookForService` called by `MyDiscoveryListener`) and then use it. The general client class implements the `clientInterface` interface and defines the common methods: `setTemplate` for defining the template to be sent to the lookup service, the mentioned `lookForService` class, the `registerForEvents` class for listening to service events from the lookup and the `getResult` class, which will be overridden and implemented in the extending classes, to get and print out the result once the proxy has been received.

```

import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lease.Lease;
import net.jini.core.lease.UnknownLeaseException;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceMatches;
import net.jini.core.lookup.ServiceEvent;
import net.jini.core.lookup.ServiceItem;
import net.jini.lookup.entry.Name;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.UnknownEventException;
import net.jini.core.event.EventRegistration;
import java.io.*;
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;
import java.net.InetAddress;

public class Client extends Thread implements ClientInterface{

    // The template defining the service we are searching for
    protected ServiceTemplate template;
    // The Array to save the discovered Lookup Services
    protected ArrayList eventRegs = new ArrayList();
    // The lookup discoverer: listen for discovery events
    protected LookupDiscovery disco;
    // The Lease time to be registerd within the lookup service
    protected final int LEASE_TIME = 1*60*1000;

    // Constructs a Client object and sets the codebase
    // and the security policy for the client
    public Client (String port, String policy) throws IOException, RemoteException{

        String localhost=getLocalHost();
        String codeBase="http://" + localhost + ":" + port + "/";

```

```

System.setProperty("java.security.policy" , policy);
System.setProperty("java.rmi.server.codebase" , codeBase);

// Set a security manager
if (System.getSecurityManager() == null)
    System.setSecurityManager(new RMISecurityManager());
else System.setSecurityManager(new RMISecurityManager());
}

// Sets the template for the client
protected void setTemplate(ServiceTemplate template){
    this.template=template;
}

// Looks for services matching the ServiceTemplate
public ServiceMatches lookForService(ServiceRegistrar lusvc, int nofService) {
    ServiceMatches Services = null;
    try{
        // Looks for the desired service into the lookup service by sending it
        // the service template (defined in the class extending this class)
        Services = lusvc.lookup(template, nofService);
    }catch (RemoteException ex) {
        System.out.println("Error doing lookup: " + ex.getMessage());
        return null;
    }
    if(Services.totalMatches == 0){
        System.out.println("No matching services");
        System.out.println("Waiting for services to be registered");
        try{
            // registers itself within the lookup service to get notifications
            // if the desired service will be registered
            registerForEvents(lusvc);
        }catch(RemoteException ex){
            System.out.println("Can't solicit events: " + ex.getMessage());
            // discards the lookup service. It can't be used anymore
            disco.discard(lusvc);
        }
        finally{
            return null;
        }
    }
    else{
        System.out.println("Got " + Services.totalMatches + " matching services!");
        // invokes the method calling the service
        getResult(Services);
        return Services;
    }
}

// Registers itself within the Lookup Server to get service events from it
protected void registerForEvents(ServiceRegistrar lusvc) throws RemoteException{
    EventRegistration evreg;
    int transition = ServiceRegistrar.TRANSITION_NOMATCH_MATCH;

```

```

    evreg = lusvc.notify(template, transition, eventCatcher, null, LEASE_TIME);
    eventRegs.add(evreg);
}

// Invokes the service and print out the results
// It will be implemented in the subclasses.
protected void getResult(Object obj){
    // To be overridden by the specific methods
}

// Gets the name of the local host as a String object
public String getLocalHost(){
    InetAddress host = null;
    try{
        host = InetAddress.getLocalHost();
    }
    catch(Exception e){
        System.out.println("Exception:" + e.getMessage());
    }
    String name=host.getCanonicalHostName();
    return (name);
}

// Starts the thread and sleep to maintain the service alive
public void run(){
    while(true){
        try{
            Thread.sleep(LEASE_TIME);
        }catch(InterruptedException ex){}
    }
}
}
}

```

The extending class defines the service template (to be sent to the lookup service) by means of the known `ServiceInterface` and implements the `getResults` method to get the result from the service by means of the received proxy:

```

import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceMatches;
import net.jini.lookup.entry.Name;
import java.io.*;
import java.util.*;

public class oneClient extends Client{

    protected Object params;
    protected Object result;

    public oneClient(params, String port, String policy) throws IOException{

        super(port, policy);
    }
}

```

```

    this.params=params;
    // sets the service template type
    Class[] types = {oneServiceInterface.class};
    Name Service[]=new Name[10];
    Service[0] = new Name("oneService");

    // sets the service template
    setTemplate(new ServiceTemplate(null, types, null));

    // only searches the public group
    disco = new LookupDiscovery(new String[] { "" });

    // Install a listener
    DiscoveryListener myListener = new MyDiscoveryListener(this, "client", 1);
    disco.addDiscoveryListener(myListener);
}

protected void getResult(ServiceMatches Services){
    System.out.println("Got a matching service, issuing the request");
    // Calls the service on the proxy using the known service interface
    result=((oneServiceInterface)(Services.items[0].service)).doSomething(params);
    // prints out the result
    System.out.println("Result: " + result);
}

// creates a Client and starts its thread.
public static void main(String args[]) {
    String port="12345";
    String policy="policy.all";
    String dir="dir";
    String params;

    try{
        // starts the web server exporting the required classes
        Thread WSthread =
            new Thread(new WebBrowser(Integer.parseInt(port), dir, true, true));
        WSthread.start();
        oneClient client = new oneClient(params, port, policy);
    }catch (IOException ex) {
        System.out.println("Couldn't create client: " + ex.getMessage());
    }
}
}

```

Of course it is possible to add some extra functionality to both service and client (such as a lease renewal mechanism), and actually we implemented it, but it was not presented here because this would complicate a lot the explanation of the core functionality of Jini services and clients.

## 7.4 Bind and use the service

At this time we can finally use the Jini clients and services. To bind a service, a client just calls the known method (defined in the `ServiceInterface`) on the proxy received by the lookup service and

gets the result.

The service will be bound to the client as soon as it receives a Remote Method Invocation on its **Backend** class which will invoke the external service and send back the result to the client.

Our prototype is just a bit more complex of what we explained here, because of the Client/Service behavior of some classes (see section 5.1). In these cases the method **useTheService** in the service class **Backend**, implements the **ClientInterface** and defines the whole client functionality to access the needed service (this is very similar to the client class).

# Chapter 8

## Results

In the following sections we will finally have a look on the qualitative behavior of SOAP and Jini with respect to time and space.

We will first highlight the main differences between the two technologies in term of time and we will later deepen on several important and sometimes surprising considerations about both technologies such as service access and network load.

### 8.1 Setup Time/Service Time

In order to use a service, a client must perform some steps, which can be divided in two main phases:

1. Setup
  - Start the Client
  - Discover the service registry
  - Inquire the service registry
  - Get the service access point
2. Service Use
  - Connect to the service
  - Issue the request
  - Wait for the answer
  - Get the result from the service and print it out

The graph of Figure 8.1 shows of a multiplication client performing different requests to a multiplication service starting some add thread request (see section 5.1 for an explanation of these adapted services). In this example we measured the setup and service time by invoking the multiplication client and varying the number of add thread (10, 100, 1000) and the number of active services on the subnet (1, 2, 4, 8, 16).

From this graph we can immediately see the big difference between a SOAP client and a Jini client: the setup time for SOAP is almost 4 times bigger as for Jini and (what is more important) the service time is really much higher for SOAP. This fact could be due to the problem we stated in section 5.2 that with the current implementation of the UDDI is only possible to search for services

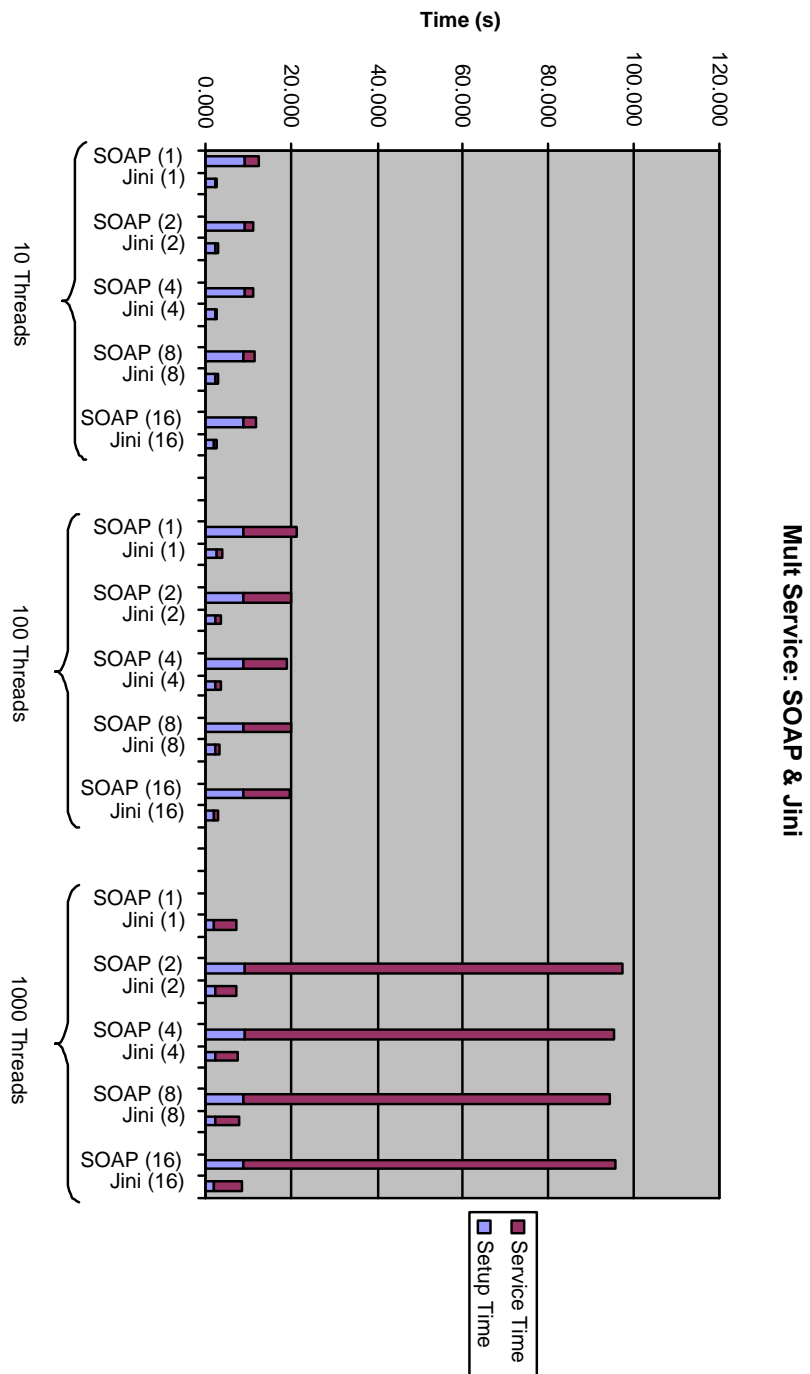


Figure 8.1: SOAP vs. Jini: setup and service time

within a business entity. The workaround for this is to make three requests to the UDDI registry to find all the web services implementing a certain interface. Not surprisingly, the setup time remains constant when increasing the number of available services or the number of started threads.

## 8.2 Scalability

In the distributed scenario, scalability is an important feature: it is generally not possible to forecast how much clients will use one service. For this reason a reliable system should work also with a big load (a high number of client requests).

In the next paragraphs we will study scalability for both Jini and SOAP, by increasing the load for the services (number of service thread) or increasing the number of available services, for both the one-step multiplication service and the two-step exponentiation service.

We begin our investigation with SOAP scalability for both the *multiplication* (Fig. 8.2) and the *exponentiation* (Fig. 8.3) services. From the SOAP measurements graphs it is possible to observe

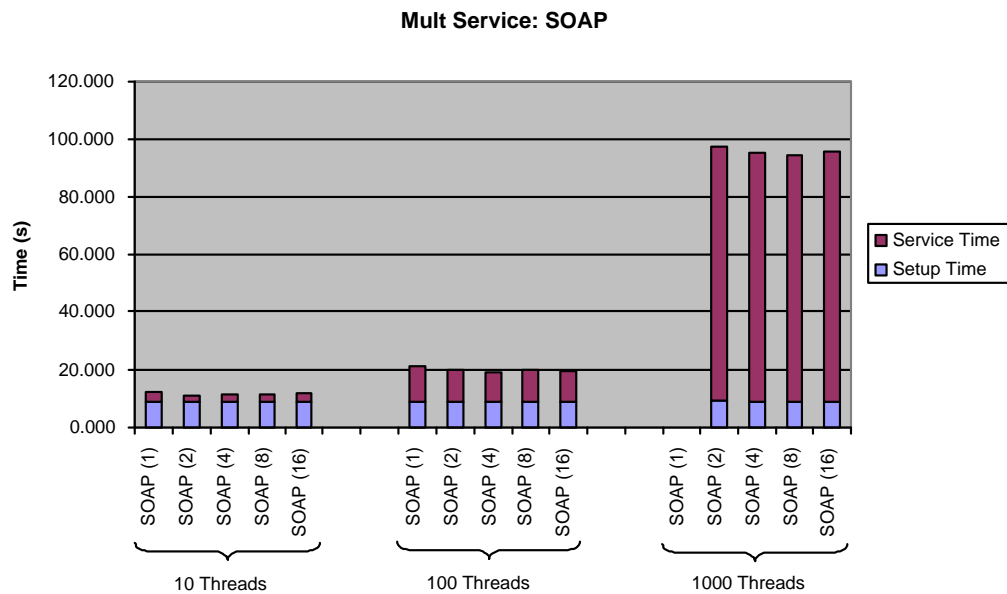


Figure 8.2: SOAP scalability: the multiplication service

that in the case of the multiplication, the service it is not so scalable. The first graph (Fig. 8.2) in particular shows practically an unscalable environment. The exponentiation graph (Fig. 8.3), instead shows a more scalable behavior. This can be maybe explained by the fact that the time spent in distributing requests over the network is high and with less requests it is preferable to work with less services.

As a note it is important to see that not all requests can be executed by SOAP: e.g. the distribution of 1000 addition threads cannot be executed using just one add service (see Fig. 8.2); this can be due to several reasons: one of these could be a sort of timeout in the soap architecture (maybe in the HTTP request to the Apache Tomcat server); another example are the exponentiation requests with 1000 add threads on an environment with less than 4 multiplication and 8 addition threads (see Fig. 8.3): these requests crashed the Java Virtual Machine, which apparently is not able to tolerate a so high load.

In parallel we investigate also the Jini scalability, using the same environment and the same requests:

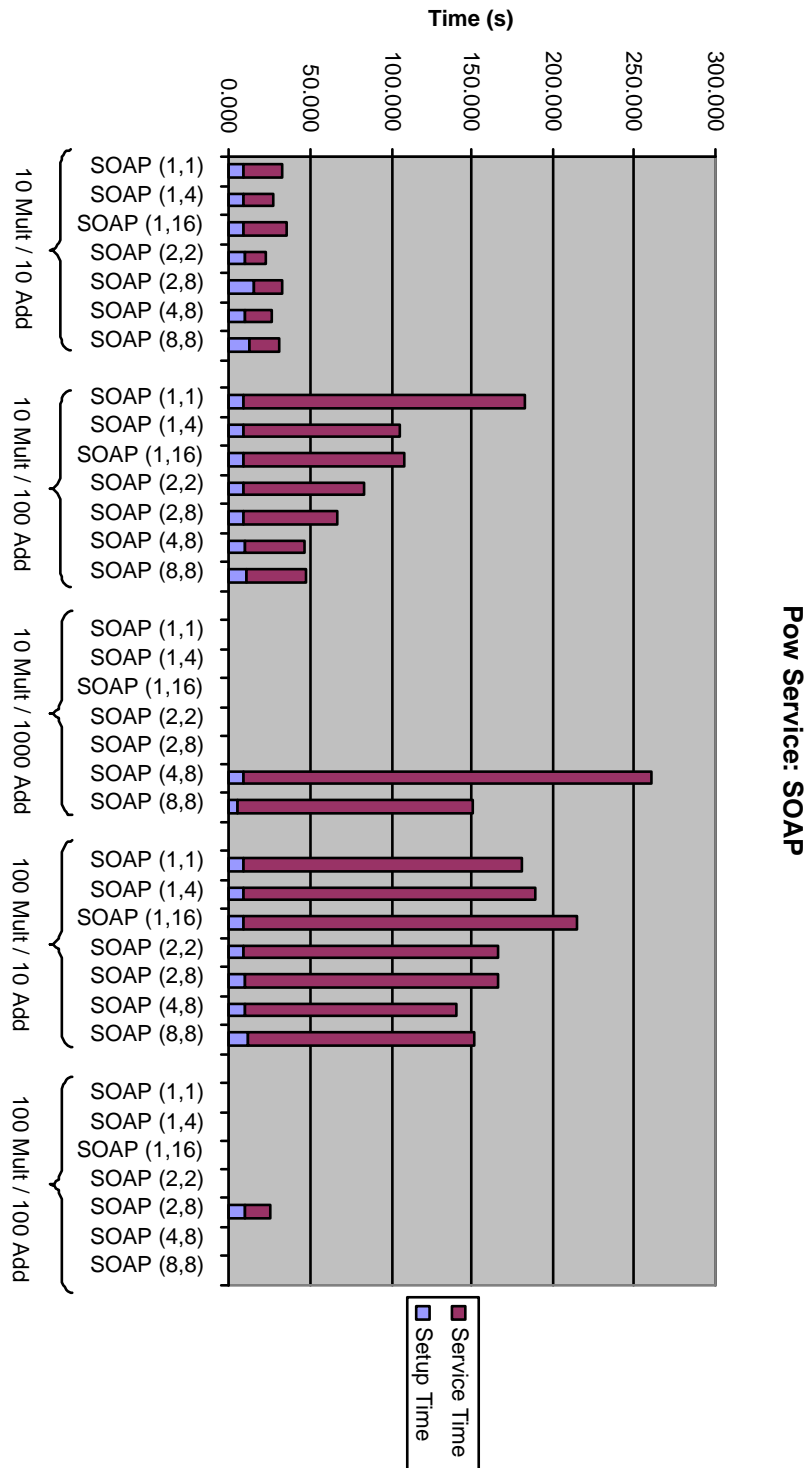


Figure 8.3: SOAP scalability: the exponentiation service

the results are somehow strange. You can see the results in the two Jini graphs: figure 8.4 and figure 8.5. By analyzing the two graphs we can see that Jini seems to be in general more scalable than

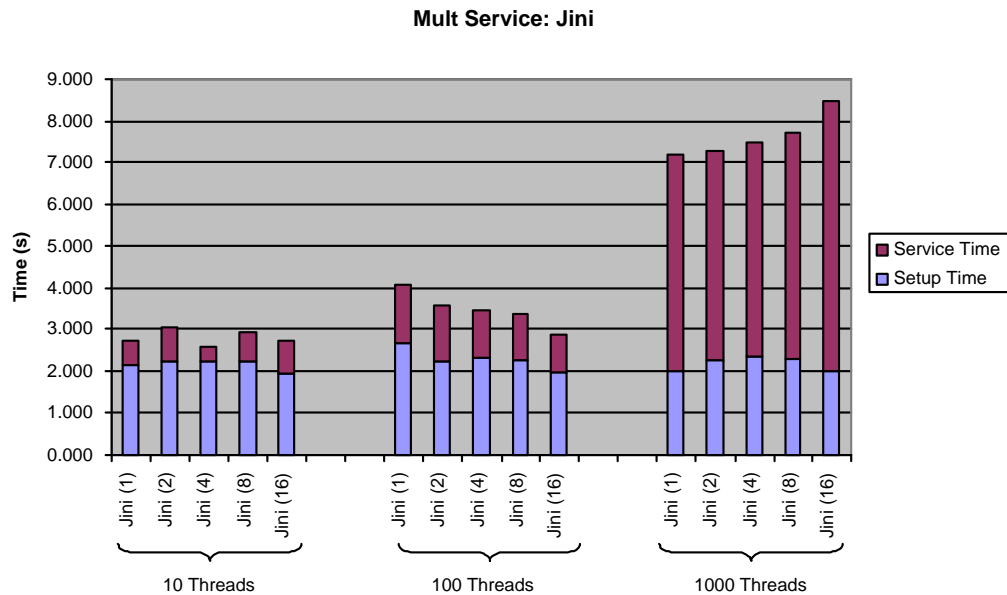


Figure 8.4: Jini scalability: the multiplication service

SOAP. However there are some strange results, i.e. the 1000 threads effort of the multiplication service (Fig. 8.4), which instead of scaling by increasing the number of service, misbehaves by increasing the total time.

Also Jini presents the same problems than SOAP with respect to the Java Virtual Machine. The JVM seems not to tolerate too much load and crashes when invoking more than 100 add threads, or in the tests using less services (see figure 8.5).

## 8.3 Network Traffic

The network plays an important role in such a distributed environment. Most of the test time is spent sending information over the network.

The network traffic test can be divided in two parts:

- Communication with the service registry to locate the service to be used
- Communication with the service to really use its functionality

For both SOAP and Jini we measured the number of packet exchanged and their total size:

	Registry		Service	
	Packets	Size (Bytes)	Packets	Size (Bytes)
<b>SOAP</b>	18	3855	13	1279
<b>Jini</b>	130	60866	75	6678

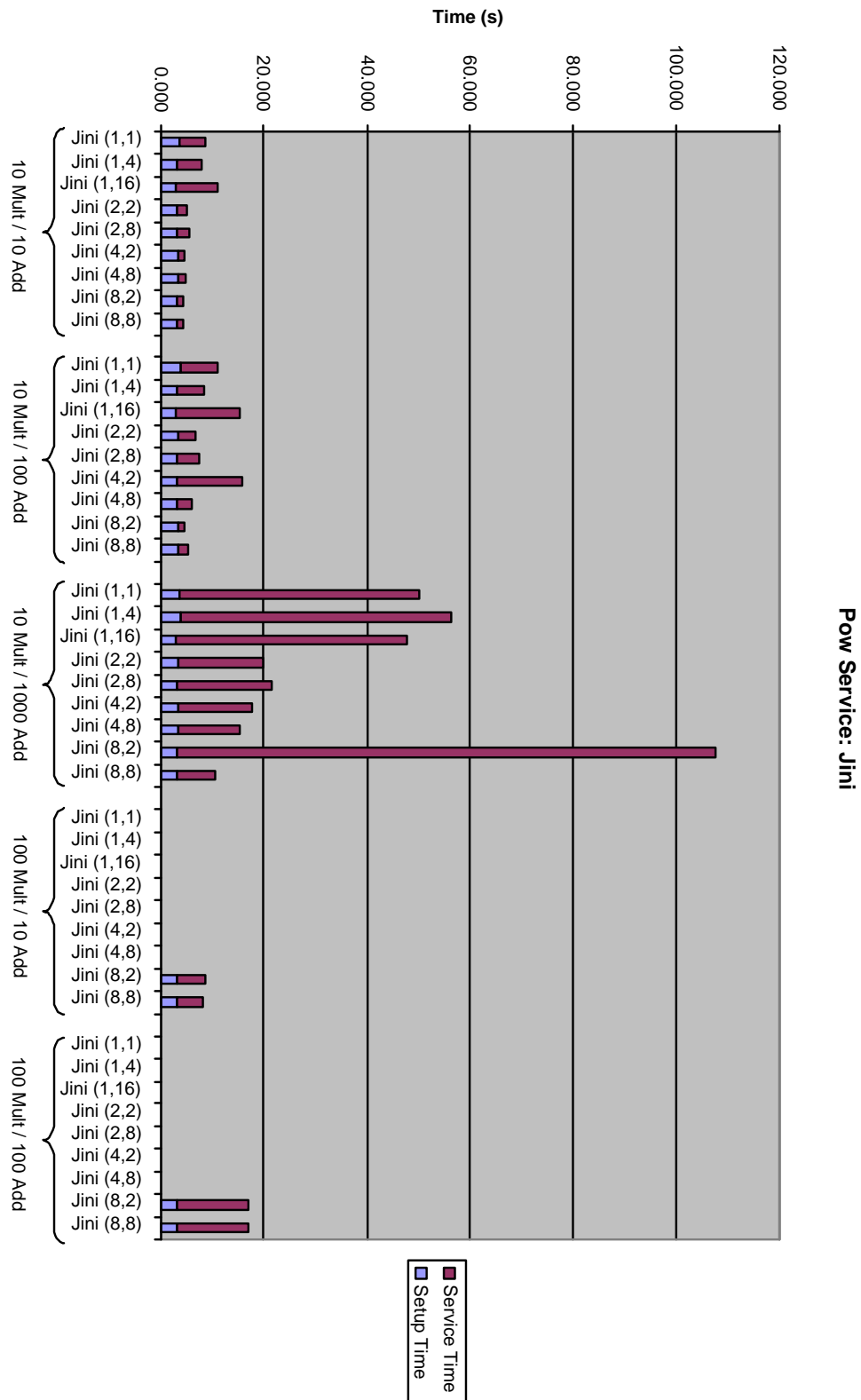


Figure 8.5: Jini scalability: the exponentiation service

Surprisingly the number of packet exchanged by Jini and the relative size is much higher than for SOAP. Even if SOAP includes a lot of structural overhead, because of the text-based XML encoding, Jini requires much more packets in order to use a webservice.

This characteristic of Jini, due to the exchange of whole Java objects, makes it not at all suited for slow networks (e.g. mobile phones). However, the XML encoding of SOAP is one of the reason of the big difference in terms of time, between Jini and SOAP: SOAP spends a lot of time in encoding and decoding information in XML.

## 8.4 Code Complexity

From the point of view of the service, implementing the core functionality of the service is practically identical for both technologies.

The SOAP technology requires in general simpler code to access and discover the services. We do not need to extend or implement any classes or interface, essentially we need only to include and use some library allowing to generate the required XML documents.

Jini involves more classes working together: practically the whole process is a real implementation of the Object Oriented philosophy (it is Java), in the sense that all classes are bound together and exchange object with each other.

However the SOAP's service description by means of WSDL seems to be more complex than the Jini's simple Java interface definition.



## Chapter 9

# Conclusion and Future Works

The goal of this work was to show the concepts and the differences between SOAP (and related complements) and Jini, two technologies allowing to build web service infrastructures. Since these technologies are relatively young, the specifications are still changing, making you feel like shooting at a moving target. From this we learned to look carefully at the version of the software stated in the documentation or in the literature, because the specification changes and consequent implementations can make the web service infrastructure completely unusable.

From our results, Jini seems to offer more functionality and flexibility, i.e. it adapts dynamically its environments (spontaneous networking). SOAP instead does not offer such features of spontaneous networking, but tends to be easier to use. However from the installation point of view Jini, due to the fact that it comes from only one software producer, is much more installation-friendly. For the SOAP environment installation, being a combination of different specifications from different organizations and different producers' implementation, we had to pay a great attention to the software implementations and their versions, to ensure that the different components could be integrated with each others.

One of the advantages of SOAP is its programming language independence, which can be useful to integrate an existent environment, i.e. there is no need to reimplement the whole infrastructure in Java. The possibility for SOAP to use HTTP as communication protocol makes the environment simpler.

Another interesting remark emerging from our result is that both technologies are not perfectly scalable. This could be a problem in a productive environment.

The big disadvantage of SOAP is the overhead of parsing the XML documents, which makes it very slow in respect of Jini. Some improvement could perhaps be done in the parsing software or in the message encoding structure.

The big disadvantage of Jini is the high network traffic generated by the exchange of Java classes between the different Java Virtual Machines, which makes Jini quite unusable for slow network environments, such as mobile phone communication.

A complementary remark on the Java Virtual Machine is that it is not so suited to manage a large number of threads, this causes sometimes some problems (like JVM crashes).

The web services technology is rapidly evolving and seem to be a very interesting topic for example for the future of e-business. At the moment it is still too young to be used in sophisticated productive environments, but we forecast that it will grow and become more stable and usable in the next two or three years.

As a motivation for possible future works, others very interesting aspects that could be considered are the security protocols provided by this two technologies. This is a very important aspect for each large scale network environment especially if business information is exchanged.

# Appendix A

## Glossary

### A.1 General

**Client** device or software component that would like to use a service

**DTD** Document Type Definition

**HTML** HyperText Markup Language

**HTTP** HyperText Transmission Protocol

**Middleware** Software interface between two abstraction layers

**PDA** Personal Digital Assistant

**RMI** Remote Method Invocation

**RPC** Remote Procedure Call

**Service** functionality made available to other users which can be accessed remotely across the network. It includes devices and software components.

**SGML** Standard Generalized Markup Language

**SMTP** Simple Mail Transmission Protocol

**TCP/IP** Transport Control Protocol/Internet Protocol

**URI** Uniform Resource Identifier

**URL** Uniform Resource Locator

**WAP** Wireless Application Protocol

**XML** eXtensible Markup Language

**XSL** eXtensible Style Language

### A.2 SOAP

**SOAP** Simple Object Access Protocol

**WSDL** Web Service Definition Language

**UDDI** Universal Discovery, Description and Integration

## A.3 Jini

**Jini** Java Intelligent Network Infrastructure, but... Jini Is Not Initials

**Leases** time restricted use of services

**Lookup service** helps clients to find and connect to services. It acts as a broker between the needs of the client and the services it knows about accross the network.

**Jini Proxy** Secure, network-aware, on-demand device driver

# Bibliography

- [1] W3C XML Schema Part 2: Datatypes. <http://www.w3.org/TR/xmlschema-2>.
- [2] W3C Simple Object Access Protocol (SOAP) 1.1. <http://www.w3.org/TR/SOAP>.
- [3] W3C Web Service Definition Language (WSDL) 1.1. <http://www.w3.org/TR/wsdl>.
- [4] Discovery Universal Description and Integration of Business for the Web. <http://www.uddi.org/specification.html>.
- [5] Jini Network Technology. <http://www.sun.com/software/jini/>.
- [6] The Jini Community. <http://www.jini.org>.
- [7] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web Services with SOAP*. O'Reilly, 2002.
- [8] RFC 2068 HTTP/1.1 Specification. <http://rfc.net/rfc2068.html>.
- [9] RFC 821 SIMPLE MAIL TRANSFER PROTOCOL. <http://rfc.net/rfc821.html>.
- [10] RFC 793 TRANSMISSION CONTROL PROTOCOL. <http://rfc.net/rfc793.html>.
- [11] RFC 791 INTERNET PROTOCOL. <http://rfc.net/rfc791.html>.
- [12] RFC 1831 RPC: Remote Procedure Call Protocol Specification Version 2. <http://rfc.net/rfc1831.html>.
- [13] RFC 793 RPC: Remote Procedure Call Protocol Specification. <http://rfc.net/rfc1050.html>.
- [14] International Standard Organisation/Open System Interconnect. <http://www.iso.org>.
- [15] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.
- [16] W3C The World Wide Web Consortium. <http://www.w3.org/>.
- [17] XML-RPC Home Page. <http://www.xmlrpc.com/>.
- [18] W3C XML Schema. <http://www.w3.org/XML/Schema>.
- [19] XML.com: A Brief History of SOAP. <http://www.xml.com/pub/a/2001/01/01/soap.html>.
- [20] W3C Namespaces in XML. <http://www.w3.org/TR/REC-xml-names/>.
- [21] W3C Overview of SGML Resources. <http://www.w3.org/MarkUp/SGML/>.
- [22] W3C HTML 4.01 Specification. <http://www.w3.org/TR/html4/>.
- [23] Erik Wilde. *Wilde's WWW, Technical Foundations of the World Wide Web*. Springer, 1999.
- [24] W3C The Extensible Stylesheet Language (XSL). <http://www.w3.org/Style/XSL/>.

- 
- [25] Wireless Application Protocol: Specifications. <http://www.wapforum.org/what/technical.htm>.
- [26] Guide to the W3C XML Specification.  
<http://www.w3.org/XML/1998/06/xmlspec-report-v21.htm>.
- [27] *UDDI Technical White Paper*. Ariba, International Business Machines Corporation and Microsoft Corporation (uddi.org), 2000.
- [28] The Source for Java Technology. <http://java.sun.com>.
- [29] Java Remote Method Invocation (RMI). <http://java.sun.com/products/jdk/rmi/>.
- [30] Keith W. Edwards. *Core Jini*. Prentice Hall, 2001.
- [31] *Jini Technology and Emerging Network Technologies*. Sun Microsystems, 1999.
- [32] Scott Oaks and Henry Wong. *Jini in a Nutshell*. O'Reilly, 2000.
- [33] *Jini Architectural Overview*. Sun Microsystems, 1999.
- [34] Keith W. Edwards and Tom Rodden. *Jini Example by Example*. Prentice Hall, 2001.
- [35] Cay S. Horstmann and Gray Cornell. *Core Java 1.2*. Prentice Hall, 1999.
- [36] *Why Jini Technology Now?* Sun Microsystems, 1999.
- [37] *UDDI Executive White Paper*. Ariba, International Business Machines Corporation and Microsoft Corporation (uddi.org), 2001.
- [38] Steve Graham, Simeon Simeonov, Toufic Boubez, Doug Davis, Glen Daniels, Yuichi Nakamura, and Ryo Neyama. *Building Web Services with Java*. SAMS, 2001.