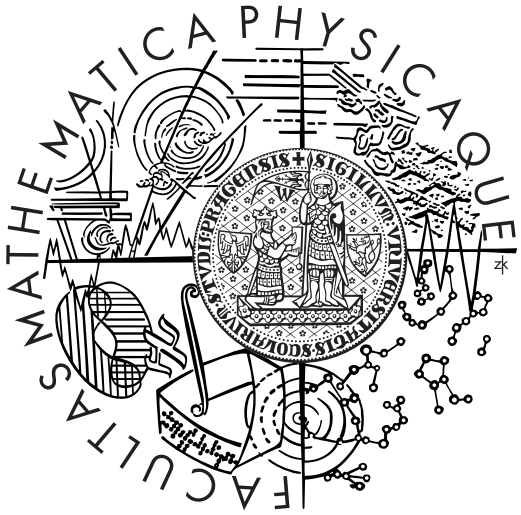
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



ŠTEFAN IGNÁTH

Aktivní XML

Active XML

Katedra softwarového inženýrství
Vedoucí diplomové práce: Doc. Ing. Jan Janeček, CSc.
Studijní program: Informatika, obor Softwarové systémy

Děkuji Doc. Ing. Janu Janečkovi, CSc. za cenné rady, náměty a konzultace, kterými přispěl k napsání diplomové práce. Dále děkuji svým rodičům, protože bez jejich mnohaleté podpory během mého studia by tato práce vzniknout nemohla. Také děkuji všem přátelům, které jsem během studia poznal.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne Štefan Ignáth

Název práce: Aktivní XML

Autor: Štefan Ignáth

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: Doc. Ing. Jan Janeček, CSc.

E-mail vedoucího: janecek@fel.cvut.cz

Abstrakt: Tato práce se zabývá využitím takzvaného aktivního XML pro implementaci distribuovaných algoritmů. Aktivní XML je představeno ve formě webových služeb (Web Services) a protokolu XML-RPC. Obě tyto technologie jsou založeny na výměne XML zpráv. Zatímco protokol XML-RPC je nyní téměř nepoužívanou a zapomenutou technologií, webové služby se za posledních několik let dostaly na výsluní. Komunikace webových služeb probíha za pomoci protokolu SOAP a rozhraní těchto služeb je definováno pomocí WSDL. Kromě přiblížení výše vzpomenutých technologií se tato práce zabývá jejich možným přizpůsobením pro implementaci distribuovaných algoritmů. Jelikož jsou požadavky distribuovaných algoritmů, tak jak jsou pojaty akademickou sférou, diametrálně odlišné od požadavků bussiness aplikací, pro které jsou webové služby primárně určeny, jsou nutné rozsáhlé kompromisy. Praktická část této práce ukazuje možná řešení pro nasazení webových služeb na výpočet distribuovaných algoritmů na ukázkové implementaci. Taktéž je implementován distribuovaný algoritmus za pomoci protokolu XML-RPC a použité technologie jsou porovnány.

Klíčová slova: webové služby, distribuované algoritmy, SOAP, XML-RPC, middleware

Title: Active XML

Author: Štefan Ignáth

Department: Department of Software Engineering

Supervisor: Doc. Ing. Jan Janeček, CSc.

Supervisor's e-mail address: janecek@fel.cvut.cz

Abstract: This work is concerned about the usage of so called active XML for the implementation of distributed algorithms. The active XML is introduced in the form of Web Services and the XML-RPC protocol. Both of these technologies are based on the exchange of XML-based messages. While the XML-RPC protocol is nowadays an unused and almost forgotten technology, the Web Services are getting more and more attention over the last few years. The communication of Web Services is based on the exchange of SOAP messages. The interface of these services is described by WSDL files. Besides the description of these already mentioned technologies, this thesis is concerned with possible modifications of these technologies in order to implement distributed algorithms. Since the requirements of distributed algorithms, as they are conceived by the academics, are diametrically different from the requirements of business applications, for which were the Web Services developed, great compromises are necessary. The practical part of this work points at possible solutions for the usage of Web Services in the distributed algorithm computing on an exemplary implementation. Another distributed algorithm is implemented with the usage of the XML-RPC protocol and the comparison of these implementations is done.

Keywords: Web Service, distributed algorith, SOAP, XML-RPC, middleware

Table of Contents

Chapter 1 INTRODUCTION

The purpose of this thesis is to discover and contemplate on differences between the world of the *client/server architecture* (most of the internet as we know it today) and the world of the *distributed algorithms* as described by academics. In the academic world the basic premise is the existence of processes (or users, or nodes); the combination of server and client role and functionality in a single point. The world of internet is suitable for implementing only a small subset of distributed algorithms known in the academic world. These are the asynchronous algorithms based on message sending.

It is impossible or very difficult to implement other types of distributed algorithms in the client/server architecture based environment. Other types of distributed algorithms include synchronous algorithms based both on message sending and on memory sharing, asynchronous algorithms based on memory sharing and several other subtypes like semi synchronous algorithms.

XML is a markup language widely used as an underlying language for many internet protocols. As it is just a text protocol, it has been used in the past for exchanging static text information (for example XHTML protocol). Over the last few years has the usage of XML grown very much. The so called active XML allows us to achieve such goals as never seen before.

Many different attempts have been made on developing distributed systems. Recent (last couple of decades) focus is of course on the distributed systems that would work over the internet. The distributed systems like CORBA and MS DCOM/COM have been developed with such focus in mind. But they have failed and are nowadays mostly used as legacy systems. The most common problem, that these systems encountered, was the so-called firewall problem. The companies' policy on firewall and router settings blocked every communication that went through any non-standard ports. With this in mind different protocols based on the exchange of XML documents have been developed. The important fact

is that the XML documents can be exchanged on the standard port 80 with the usage of HTTP.

One of the simpler XML-based protocols is the *XML-RPC* protocol. It is a simple protocol that supports only remote procedure calling with few different variable types. The only available message exchange pattern is the request/response pattern as the only underlying protocol, that is supported, is HTTP. The main advantage of this protocol is its simplicity.

The most common and used protocol is *SOAP* (Simple Object Access Protocol). SOAP Version 1.2 (SOAP) is a lightweight protocol designed for exchanging structured information (such as XML) in a distributed network environment. It uses XML technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics [32].

1.1 Theoretical Part

For the theoretical part of this work we will cover the history of middleware from the oldest systems like CORBA to the newest middleware technologies – *Web Services*. We will describe the technologies used by Web Services and compare and describe the frameworks available for their development. Later, the problems and implied solutions of the usage of Web Services for a distributed algorithm implementation are discussed.

1.2 Practical Part

In the practical part of this thesis we will describe the implementation of distributed algorithms using different chosen technologies in order to show the necessities to make the chosen technologies work as distributed algorithms. One implementation was created using Web Services as the communication environment for the *resource reservation* algorithm. The second implementation was done with the usage of XML-RPC as a simpler protocol. The second implementation is needed for comparison of these technologies (SOAP and XML-RPC) as both are active XML technologies. The XML-RPC protocol is nowadays only a legacy technology. The practical part is also concerned with performance tests done for both protocols in order to compare them successfully.

1.3 Overview of Chapters 2-7

Chapter 2 describes different XML-based internet technologies. It describes the client/server architecture as the architecture used by those technologies that is this work concerned about.

It continues with the definition and the description of different middleware technologies as CORBA, DCOM and Web Services. This chapter ends with the description of three protocols that are the core of technologies used in this work (SOAP, WSDL and XML-RPC).

In Chapter 3 we discuss the decision to use Java to implement the chosen distributed algorithms. First, we compare different SOAP frameworks that were available at the time. Then, we describe the implementations that we have chosen to use with short example codes of the creation of server and client classes.

Chapter 4 contains the informal and formal description of distributed algorithms. Distributed algorithms are in this work formally described as I/O automata. The last section of this chapter explains the translation of I/O automata into client/server applications.

Chapter 5 is concerned about the differences of the Web Services environment and the environment assumed by the distributed algorithms. We iterate one by one through the different problems that arise from these differences and we devise and describe solutions to these problems.

Chapter 6 presents the implementation of a leader election distributed algorithm with the usage of the XML-RPC protocol. At first we describe the leader election algorithm in details, giving its formal description and the complexity analysis. Then we discuss the implementation details. At the end of the chapter we present the performance measurements done for the XML-RPC protocol and briefly compare them with other technologies.

Chapter 7 has a similar structure to Chapter 6, but it concentrates on an algorithm for resource reservation in a distributed environment and its implementation in the JAX-WS framework. The last section of this chapter presents the performance measurements done for the SOAP implementation in the JAX-WS framework and compares them to the results of the XML-RPC protocol.

Chapter 8 brings the conclusion of this work and discusses a possible usage and consequences of findings and methods devised in this work. It also briefly mentions the work that can be done in this field.

Chapter 2 XML-BASED INTERNET TECHNOLOGIES

2.1 Client/Server Architecture

Although there is no universal agreement upon definition of what the term 'client/server' means, it is generally accepted to mean any system that splits its data processing between two distinct parts or programs: the requester and the provider, or the client and the server, where the client makes requests for services provided by the server [7].

"Servers are entities that passively await requests from a client and respond to them. Servers fill one specific need and encapsulate the provided service so that the state of the server is protected and the means by which the service is provided are hidden from the client" [23].

Clients, on the other hand, are active entities that send requests to servers. They await messages only as the response to their requests and are not opened to any other incoming communication. The distribution of responsibility, authority, and intelligence between a server and a client allows us to differentiate between the so called fat and thin clients. The server portion of a client/server system almost always holds the data, and the client is nearly always responsible for the user interface. The shifting of the application logic constitutes the distinction between *fat clients* and fat servers (with *thin clients*).

A thin client is a client, which delegates most of the actual workload (data handling, algorithm computing, and data storage) to the server. As the fat server harbors most of the functionality it is usually easier to update the application logic since a new client does not need to be distributed.

A fat client does most of the workload by it self. As the actual work is mostly done by the clients, the server handling fat clients can serve more clients while having the same computing power. The disadvantage of the usage of the fat clients is usually the higher bandwidth usage (that is being eliminated as the bandwidth has grown considerably over the last few years).

The main advantage and disadvantage of the client/server architecture is the existence of a *central node* (server). It is a disadvantage, as it is a single point of failure (system comprising 10,000s of clients and a server will fail in the case of a failure of the server). That is why a system based on the client/server architecture usually lacks robustness. On the other hand, the server is usually the only place, where data is stored and thus higher security can be provided.

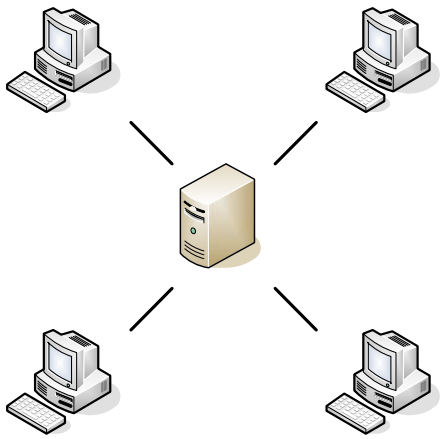


Figure 1: The client/server network architecture

2.1.1 Client/Server vs. Peer 2 Peer

The peer-to-peer (P2P) architecture does not have the notion of clients or servers, but rather every node fulfills the roles of both 'the server' and 'the client' with respect to other nodes. The advantage of a P2P network is that each node provides the resources (bandwidth, storage space, and computing power), thus the total capacity of the system increases with the increased number of nodes. This also provides the robustness of P2P networks, where *no single point of failure* exists.

As mentioned before, the distributed algorithms as described by Lynch [24] use only the P2P architecture. The XML technologies such as SOAP on the other hand are implemented only for usage in the client/server architecture. There are some exceptions beginning to emerge, such as WSPEER and WSKPEER [13].

The aim of this work is not to explore the usage of middleware such as WSPEER in P2P networks, but rather to use the client/server technologies to simulate the P2P architecture.

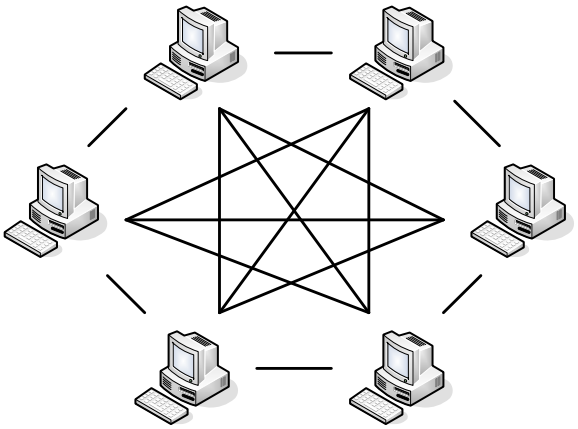


Figure 2: The peer-to-peer network architecture

2.2 *Middleware*

When the Internet Society held a workshop on middleware [3], the participants decided that the definition was dependant on the subjective perspective of those trying to define it. Nevertheless this is a definition that we have found more or less suitable for the scope of this thesis: "The main purpose of middleware is to *overcome the heterogeneity* of the distributed infrastructure. It establishes a new software layer that homogenizes the infrastructure's diversities by means of a well-defined and structured distributed programming model. Abstractions provided by middleware systems hide heterogeneity of the networking environment, support advanced coordination models among distributed entities and make the distribution of computation as transparent as possible" [17].

The taxonomy of middleware platforms allows a categorization according to the coordination model they implement. *Transactional* middleware systems use transactions to guarantee a consistent system state. They use the two-phase commit (2PC) to implement the transactions. The Distributed Transaction Processing (DTP) protocol defines a programmatic interface for 2PC. The disadvantages of transactional middleware are the unnecessary overhead, if transactions are unnecessary or undesired and automated (un)marshalling is in most systems not provided. Such middleware is used in databases, telecommunications and safety critical systems [11].

A *tuplespace-based* middleware builds upon the distributed shared memory concept. The tuplespace is globally shared in between components and it enables complete decoupling of the components, such they do not need to co-exist at the same time or have reference to each other. A message-oriented middleware can be seen as a special case of the tuplespace-based middleware, where tuples are implemented as messages.

Remote procedure call (RPC) is the oldest coordination paradigm and it allows a component to invoke procedures executed on remote hosts without explicitly coding the details for these interactions. The cross-platform nature (due to bindings for multiple operating systems and programming languages) is a huge advantage of this type of middleware, but it also suffers from a number of disadvantages (such as small fault tolerance, limited scalability, no support for multicast or asynchronous communication). All of that has resulted in limited use of RPC in modern distributed systems. One such middleware is XML-RPC.

An object and component oriented middleware represents the evolution of RPC-based middleware. This kind of middleware allows a creation of stubs, obtaining of reference to remote objects, establishing synchronous communication and invoking requested method by marshalling and unmarshalling data. The best known middleware initiatives were Object Managements Group's CORBA (Common Object Request Architecture), Microsoft's COM/DCOM (Component Object Model) and Sun's EJB (Enterprise Java Beans).

A *service-oriented* middleware is a further step of evolution toward the Service Oriented Architecture. In these systems resources are available as autonomous software services that can be accessed without any knowledge of their underlying technologies. The three basic architectural components are a service provider, service consumer and a service registry. The service-oriented middleware introduces languages for exact description of services in order to hide the heterogeneous underlying environment. The service-oriented middleware is usually based on XML standards and example of such middleware is Java API for XML Web Services (JAX-WS) [21].

2.2.1 CORBA

Common Object Request Broker Architecture or CORBA is an architecture offering an access to services in the form of *objects with a unified interface*. In other words, the main purpose of CORBA is the interoperability of CORBA-based programs independent on the vendor, operating system, programming language and/or the network environment. CORBA was developed and standardized by the Object Management Group [26].

The Object Model Architecture (OMA) is composed of an *Object Model* and a *Reference Model*. While the Object Model defines the description of objects in a heterogeneous environment, the Reference Model describes interactions between these objects. So called *Object Request Broker* (ORB) is the component mainly responsible for facilitating communication between clients and objects. In the client/server architecture the objects would

be considered the server side. In CORBA a client and a server are merely roles, as they can change on a per-request basis (as a "client" containing an object can serve as a "server" for another service).

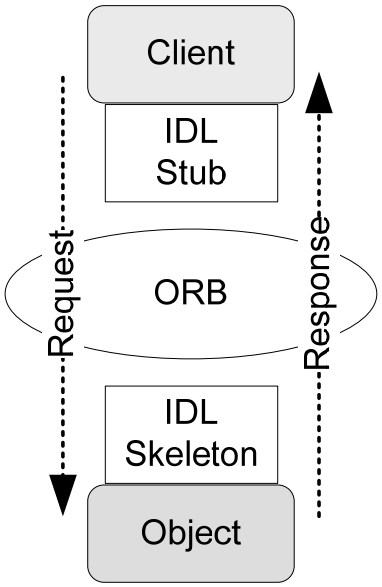


Figure 3: The basic architecture of CORBA

The main features of CORBA according to Vinoski [31] are ORB Core, OMG Interface Definition Language, Interface Repository, Language Mappings, Stubs and Skeletons, Dynamic Invocation and Dispatch, Object Adapters.

The uniqueness of CORBA lies in the usage of ORB. ORB delivers requests to objects and returns responses to the clients. The key feature of ORB is the *transparency* as it hides the objects location, implementation, execution state and the communication mechanism.

Some of other features as stubs and skeletons usage are nowadays used in other technologies as JAX-RPC [19]. A stub is a mechanism that creates and issues request on behalf of a client, while skeleton is its server-side mirror and it serves by delivering requests to objects (in case of CORBA).

Even-though CORBA has never become the *next-generation technology* for e-commerce as it was envisioned during the 90's there are still some systems based on CORBA in use and even in development. CORBA is nowadays used mainly to run inside companies' networks, where the communication is protected. In the area of *embedded* and *real-time systems* the use of CORBA is actually growing.

The main problems and reasons of the downfall of CORBA are the *technical issues*, such as the high complexity, that caused some parts of the CORBA specification never to be

implemented. Other technical issues were insufficient features. CORBA is lacking on security and versioning. The unencrypted traffic enables eavesdropping and different attacks if CORBA was to be used over the internet. The other technical difficulty was a conflict with corporate security policies. CORBA needed special ports to be opened in companies' firewalls, unlike SOAP that uses port 80 to its advantage. There are also a vast number of other technical issues such as design flaws, redundancy in on-the-wire encoding, missing threading support and the lack of mappings for different programming languages [15].

Also many *procedural issues* within the OMG organization together with the high price of CORBA from commercial vendors contributed to the downfall of CORBA. In the 1999 SOAP has emerged as a technology using XML as the encoding for remote procedure calls. SOAP has taken the place of CORBA as the technology for e-commerce.

2.2.2 MS DCOM/COM

Distributed Component Object Model or DCOM is a proprietary technology developed by Microsoft for software components across several networked computers to communicate with each other [33]. The traditional COM components can only perform interprocess communication across the process boundaries on the same machine. DCOM with the usage of RPC (Remote Procedure Call) enables sending and receiving of information between the COM components over the network.

DCOM addresses many different issues such as a location independence, connection management, bandwidth and latency, scalability, security, load balancing, fault tolerance, protocol and platform neutrality [25].

In comparison to CORBA DCOM *addresses security issues* and is able to tunnel TCP to run on the port 80. It distinguishes between four fundamental aspects of security: access, launch, identity, and connection policy. DCOM supports not only single-threaded apartments (STA) in which each COM object lives in a single thread, but also *multithreaded apartments* (MTA) in which clients from any thread within the process can directly call any object inside the MTA. In MTA COM objects do not live in any specific thread.

One of the key mechanisms of COM is for establishing connections to components and creating new instances of the components. These mechanisms are commonly referred to as *activation mechanisms*. The creation of new components is handled by the communication of service control managers (SCM) on each side (client and server). In order to communicate the server name and the class identifier are needed.

Other important feature of COM is *marshalling*. Marshalling is the serialization and deserialization of arguments and return values of method calls in order to allow their transport over different protocols and media. This is needed in order for remote procedure calling to work. Parameters marshalling is nontrivial because parameters can be arbitrary complex (such as pointers to arrays or pointers to structures). The counterpart to marshalling is *unmarshalling*, the process of reading serialized data and recreating their original structure. For a COM object to marshal and unmarshal all parameters correctly the data types are defined by the distributed computing environment remote procedure call (DCE RPC) and the method signatures are described by the interface definition language (IDL).

DCOM also handles the *distributed garbage collection*. This aims to solve the problem with garbage collection in distributed environment, dealing with process crashes and connection problems. DCOM uses the *reference counting* as the primary mechanism for controlling object's lifetime. As the remote reference counting would work only if clients did not terminate abnormally when holding a remote reference, DCOM uses *pinging* to determine a client's abrupt termination [16].

The Microsoft DCOM technology has been deprecated in favor of the Microsoft .NET architecture. DCOM was developed in the year 1997 as the answer to CORBA. As .NET was released in the year 2002, DCOM has died as a perspective technology.

2.2.3 Java RMI

The Java Remote Method Invocation (Java RMI) framework offers an alternative to the direct handling of sockets in Java. It enables the programmer to call remote methods as if they were local, while the framework packages and ships the arguments to the target of the remote call. Stubs are used as *local surrogate objects* on which the call is invoked and they manage the invocation on a remote object. The Java RMI framework integrates the distributed object model into the Java programming language while preserving as many features of the Java language as possible.

Java RMI handles three different things. The first is the *location* of remote objects. This is done by the `rmiregistry`, where an application can register remote objects. The second is the actual *communication* with remote objects and the third is the *loading* of class bytecodes of objects passed as references to and from remote objects.

Java RMI uses a standard mechanism of RPC systems. It employs the *stubs and skeletons* in order to manage the communication between a local application and remote objects. The implementation of remote objects has to be thread-safe as they might be executed in separate

threads. The garbage collection is handled by reference counting of live references within each Java virtual machine (JVM).

The Java RMI calls are made directly from a socket to a socket, but this kind of traffic is usually not allowed on many firewalls. The newest version of Java RMI provides an alternative to run RMI on a HTTP-based mechanism. The authors of Java RMI specification [1] and others [20] warn that the calls transmitted via HTTP requests are at least an order of magnitude slower than the direct calls.

2.2.4 Web Services

W3C defines Web Services as "a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks. Web services are characterized by their great interoperability and extensibility, as well as their machine-processable descriptions thanks to the use of XML. They can be combined in a loosely coupled way in order to achieve complex operations. Programs providing simple services can interact with each other in order to deliver sophisticated added-value services."

The Web Service architecture is based on the exchange of SOAP (Simple Object Access Protocol or Service Oriented Architecture Protocol) messages between a service *requester* (client) and a service *provider* (server). Before the client can access the server, it has to learn about the Web Service. The Web Service discovery is done via a service *broker*. The service broker distributes via UDDI (Universal Descrition Discovery and Integation) protocol WSDL (Web Service Description Language) documents describing given services.

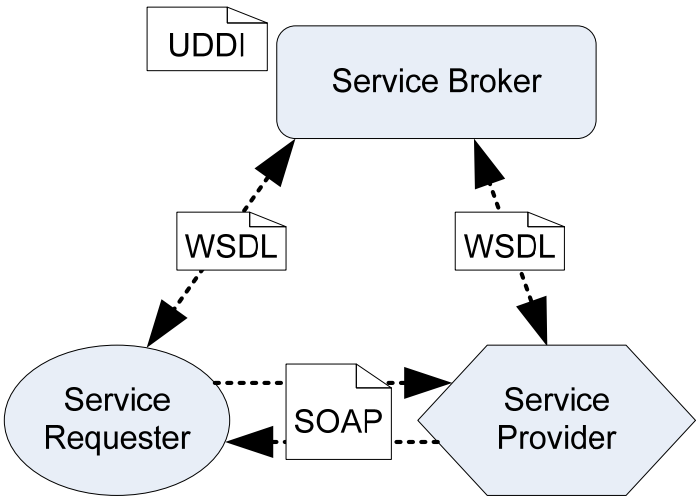


Figure 4: The Web Service architecture

Many different specifications exist that add *new capabilities* to Web Services. These specifications are generally referred to as WS-*. Some of them are: WS-Security, WS-Reliability, WS-ReliableMessaging, WS-Addressing and WS-Trasanction [34].

Web Services have been evolved with the knowledge gained from older middleware frameworks, such as CORBA or Java Remote Method Invocation. The main emphasis was on the *interoperability*. Web Services have been developed to enable a service-oriented architecture (SOA) for business aplications, to intergrate with existing legacy systems, to increase the programming productivity by simplifying the development.

2.3 XML-RPC

XML-RPC is a very simple XML-based protocol for remote procedure calls. It defines a framework for transmitting the method calls and the resulting responses between processes across hosts. The XML-RPC approach differs from the traditional RPC systems in several ways [4].

XML-RPC does not generate stubs in order to allow invoking the remote procedures in a client. It rather provides several *primitives* that allow the programmers to invoke the remote procedures. This reduces the amount of work required by the programmers, because they do not have to tightly specify the procedures for a stubs generator. On the other hand, as the stubs are not generated, better knowledge of the underlying system is required.

XML-RPC uses a standard XML encoding strategy, therefore is it highly interoperable. Since the XML-RPC systems are only loosely coupled by XML documents they exchange, it is possible for them to communicate as long as they obey the specification.

Since XML-RPC uses HTTP (Hyper Text Transfer Protocol) [6] it is really easy to integrate with other applications. The usage of HTTP also provides the convenience of being able to pass through the most of firewalls, since it usually runs on the standard port 80.

The mayor inconvenience of this protocol is that it can handle *only simple data structures*. For this reason this protocol was not adopted by W3C [22] as a standard.

2.3.1 Protocol Structure

This short description is based on the full specification available at [35].

XML-RPC is a protocol based on transferring XML documents over HTTP. Each message contains a simple HTTP header and a XML body. The protocol differentiates between request and response messages. The request contains a <*methodCall*> element while the response contains a <*methodResponse*> element. The <*methodCall*> contains an element

<*methodName*>, with a unique method name and an element <*params*>. The element <*methodResponse*> contains only the element <*params*>.

The element <*params*> contains a list of parameters of the given method. Each parameter has its own element <*param*>. XML-RPC supports only few scalar value types such an integer, Boolean, string, double, data and base64-encoded binary. It also supports <*array*> and <*struct*>, that can contain a list of named (in case of <*struct*>) or unnamed (in case of <*array*>) values of different types. Both <*struct*> and <*array*> may contain even another <*struct*> and/or <array>.

The element <*methodResponse*> may contain an element <*fault*> instead of the element <*params*>, but only in case of some error happening.

An example of a XML-RPC message follows:

```
POST /RPC2 HTTP/1.0
User-Agent: Mozilla/5.0 (Windows NT 5.1; en-US) Firefox/2.0.0.9
Host: localhost:8080
Content-Type: text/xml
Content-length: 166

<?xml version="1.0"?>
<methodCall>
<methodName>example</methodName>
  <params>
    <param>
      <value>
        <i4>1</i4>
      </value>
    </param>
  </params>
</methodCall>
```

2.4 SOAP

„SOAP is fundamentally a stateless, one-way message exchange paradigm, but applications can create more complex interaction patterns (e.g., request/response, request/multiple responses, etc.) by combining such one-way exchanges with features provided by an underlying protocol and/or application-specific information. SOAP is silent on the semantics of any application-specific data it conveys, as it is on issues such as the routing of SOAP messages, reliable data transfer, firewall traversal, etc. However, SOAP provides the framework by which application-specific information may be conveyed in an extensible manner. Also, SOAP provides a full description of the required actions taken by a SOAP node on receiving a SOAP message.“ [32]

Simple Object Access Protocol (SOAP) is a protocol based on exchanging XML-based messages enabling remote procedure calls. As SOAP message is just a XML document, the protocol allows the communication of applications written in different languages and deployed on different platforms over the network. In comparison to the XML-RPC protocol, SOAP can handle any *arbitrary complex data* type and was adopted by W3C as a standard.

The common transportation protocol for SOAP is HTTP as the HTTP binding (that enables SOAP to be transported over HTTP) is described directly by W3C. SOAP can be transported also over other network protocols like SMTP, UDP and TCP with the usage of the SOAP *binding framework* [27]. The other transportation protocols besides HTTP are used and supported only on rare occasions. For instance, SOAP-over-UDP is a suitable solution for mobile applications.

SOAP supports even other message exchange patterns (MEPs) than the request/response MEP. A MEP has to be supported by the underlying transport protocol binding in use, otherwise it is not really usable. The *request/response and response* MEPs are supported by the HTTP binding (the only binding provided by W3C). Other MEPs have to be defined and supported by other bindings.

The main advantage of SOAP and the reason why it is today an industrial standard is its *interoperability* and the binding to the most common application protocol (HTTP), which solves the firewall issue that was encountered by the older rivals (CORBA, DCOM).

The disadvantage of SOAP is the *overhead* it carries, such the computational complexity of XML parsing and processing, usage of HTTP when it might not be needed or the fact that the application protocol (HTTP) is used as a transport protocol.

2.4.1 Protocol Structure

The SOAP message is contained within an element <*env:Envelope*>, which may contain one or more <*env:Header*> elements and exactly one element <*env:Body*>. Other elements are always contained within these two elements.

The element <*env:Header*> is optional and serves to pass control information (not related to the payload inside the <*env:Body*> element) to final or intermediary nodes. These elements may be inspected, inserted, deleted or forwarded by SOAP nodes encountered along the SOAP message path. The element <*env:Body*> is obligatory and contains the whole payload aimed for the ultimate SOAP receiver.

SOAP recognizes three role attribute values: *next*, *none* and *ultimateReceiver*. If an element has a role, each type of SOAP node (sender, intermediary, ultimate receiver) knows

how to act in given role. Users may define other roles and the behavior of nodes in those roles.

Headers may contain an attribute *mustUnderstand*, and if this attribute is set true, the processing of this header block is mandatory. Headers may contain also a *relay* attribute; if this is set true, the header has to be relayed if is not processed.

If an error occurs while processing a SOAP message, a message containing a <*env:Fault*> element as the only child of the <*env:Body*> element has to be created and send to the originator of the faulty message.

In order to invoke the SOAP RPC, the following information is needed: the address of the target SOAP node, the procedure name, the identities and values of the input and output parameters and the MEP to be employed to convey the RPC.

An example of a SOAP message follows:

```
Host:  localhost:8080
User-agent: Mozilla/5.0 (Windows NT 5.1; en-US) Firefox/2.0.0.9
Accept:text/xml,application/xml,text/html;q=0.9,text/plain;
Accept-language: en-us,en
Accept-charset:  ISO-8859-1,utf-8
Content-type: text/xml
Content-length:  355

<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns:resourceReservation
     xmlns:ns="http://cz.cuni.mff.coloring/">
      <resourceId>99</resourceId>
      <processId>99</processId>
    </ns:resourceReservation>
  </soap:Body>
</soap:Envelope>
```

2.5 WSDL

WSDL or Web Service Description Language is a XML-based language used to describe a public interface of a Web Service. The WSDL file is a XML document used by a service consumer to learn about the service. WSDL defines an XML grammar for the description of network services as collections of communicating nodes capable of exchanging messages.

A Web Service defined by a WSDL document is not dependent on implementation on any particular platform and programming language. The interface of the Web Service is described

as an abstract *PortType*, defined by the input and output SOAP messages for each procedure. The WSDL document also defines the location and implementation style of given service.

The input and output message types for each method are usually defined using the XML Schema Definition (XSD) language. The type definition might be separated into a XSD file, which is imported into the WSDL file with the <*xsd:import*> element.

A WSDL document serves as an input for automated generation of proxies as stubs, skeletons and templates, needed to implement the Web Service and/or the client. The development of Web Services with automated generation of proxies does not require any (or minimal) knowledge of SOAP.

While the current recommended version by W3C is 2.0, the version 1.1 is still the only supported version by most of the software development kits for Web Services.

2.5.1 Protocol Structure

As defined by [9]:

"A WSDL document defines services as collections of network endpoints, or ports. In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings. This allows the reuse of abstract definitions: messages, which are abstract descriptions of the data being exchanged, and port types which are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service. Hence, a WSDL document uses the following elements in the definition of network services:

- *Types*– a container for data type definitions using some type system (such as XSD).
- *Message*– an abstract, typed definition of the data being communicated.
- *Operation*– an abstract description of an action supported by the service.
- *PortType*–an abstract set of operations supported by one or more endpoints.
- *Binding*– a concrete protocol and data format specification for a particular port type.
- *Port*– a single endpoint defined as a combination of a binding and a network address.
- *Service*– a collection of related endpoints."

Chapter 3 IMPLEMENTATIONS OF XML-BASED PROTOCOLS IN JAVA

3.1 Why Java?

Over the last few years many implementations of SOAP in different programming languages have been developed. Most of these are of course in *Java* and *C++*, but there are some implementations available in other languages such as PHP (PHP SOAP), Perl (SOAP::Lite [29]), Ruby (SOAP4R) and some other languages developed on academic grounds.

The best known open source implementations of SOAP are *Apache Axis* [5] (primarily Java, but now also C++) and *gSOAP* [12] (C++). While Apache Axis is a robust system, gSOAP is just a modest framework with a set of tools. The performance of a Web Service system is largely dependant on the performance of the XML parser in use. The gSOAP's XML parser performs really well in comparison with the other parsers and XERCES (Java and C version), parser used by Apache Axis, performs terribly in comparison to gSOAP [14].

The best known proprietary implementations are *GlassFish JAX-WS* and *.NET*. Both of these implementations provide a huge framework consisting of many additional features. JAX-WS used to be a part of Sun's proprietary J2EE platform, but nowadays it is developed as a part of GlassFish community project [18] as an open source, production-quality enterprise software. The .NET platform is Microsoft's proprietary technology.

We took three of the relevant frameworks into consideration: gSOAP, JAX-WS and .NET. Each of these has its advantages and disadvantages.

3.1.1 gSOAP

The gSOAP framework is the only framework out of the chosen three that is developed on academic grounds. It is somewhat '*lightweight*' in comparison to the other two frameworks. It does not have its own IDE, but it can be integrated into the Microsoft Visual C++. gSOAP provides its own web server and supports even SOAP-over-UDP and the XML-RPC protocol. Its 'lightness' contributes to its high performance according to its developers and some

research [14]. gSOAP is also very *portable* as it covers not only the common platforms like Win32, Unix-like platforms and Max OS X, but handheld platforms like WinCE, Palm OS and Symbian too. gSOAP is distributed under several different licenses and it is easy to find the right one.

The downside of gSOAP is that the Web Service creation is not fully automated and it is uses a generation *old approach* compared to its opponents. The creation of Web Services is very similar to Java API for XML Remote Procedure Call (JAX-RPC [19]), which is the ancestor of JAX-WS.

3.1.2 .NET

The .NET framework is a proprietary framework developed by the Microsoft Corporation. The .NET framework in its current version 2.0 (3.5 is being released) comprises a large variety of tools and not only for the development of Web Services. This framework supports fully automated creation of Web Services in its own IDE MS Visual Studio. The free version of this IDE is available for personal usage, it is provided with a web server for the development and testing.

The Web Service development and testing is fully automated within the provided IDE. A specialized package called Web Service Enhancements (WSE) is available in version 3.0. This package ads the possibility to use *SOAP-over-TCP* without the usage of HTTP.

The main disadvantage of the .NET framework is its dependence on the *Windows platform*. The only possible way to run any application developed under .NET on other platforms is Mono Project [28] that allows for .NET applications to be run and developed on Unix-based systems. However, since this allows only partial compatibility, it is possible to say that .NET is dependent on the Windows platform.

3.1.3 JAX-WS

JAX-WS as a part of the GlassFish project is developed by Java Community Process as an open source project. The GlassFish project is an application server implementing newest Java EE 5 platform features such as Java Server Pages (JSP), Java Server Faces (JSF), Servlet, Enterprise Java Beans, JAX-WS, Java Architecture for XML Binding (JAXB) and many others.

JAX-WS is distributed as a part of the GlassFish project together with the NetBeans IDE. The current version is 2.1. The Web Service development and testing is fully automated and integrated into the NetBeans IDE as the development of clients is.

GlassFish is as *platform independent* as Java Enterprise Edition, therefore it is possible to run in on most of known platforms (including handheld platforms for clients).

The only disadvantage of this framework is its performance, which might not be as high as the performance of gSOAP, due to Java's reputation and some comparisons [14]. Since the comparisons were done on much older implementation, it is hard to take them into account.

We have decided to use the JAX-WS framework. The reasons were:

- *the platform independency*, that opens large testing possibilities needed for distributed algorithms
- *automated Web Service and client generation* speeds up the development process, as it allows to concentrate on the algorithmic part, as opposed to having manually create interfaces or WSDL files
- *previous experience* with Java, which sped up the development process a lot

3.2 XML-RPC

For comparison purposes we have decided to create an implementation of a distributed algorithm not only in a full fledged SOAP/WSDL framework, but also in a much simpler XML-based protocol, namely XML-RPC. For this purpose we have decided to use the Apache XML-RPC, version 3.1, implementation [35].

This implementation is in Java and for testing purposes it supplies a very simple embedded web server. A brief description of the creation of basic server and client classes follows.

3.2.1 Client Classes

The creation of a client is very simple; just two classes are to be used: `XmlRpcClient` and `XmlRpcClientConfigImpl`. `XmlRpcClient` is a stateless, thread safe object. `XmlRpcClientConfigImpl` is a simple object allowing setting of few properties of the HTTP client. The creation of the client as described by [5]:

```
import org.apache.xmlrpc.client.XmlRpcClient;
import org.apache.xmlrpc.client.XmlRpcClientConfigImpl;

XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
config.setServerURL(new URL("http://127.0.0.1:8080/xmlrpc"));
XmlRpcClient client = new XmlRpcClient();
client.setConfig(config);
Object[] params = new Object[]{new Integer(33), new Integer(9)};
Integer result = (Integer) client.execute("Calculator.add", params);
```

3.2.2 Server-Side XML-RPC

The server consists of an `XmlRpcServer` class and a class implementing methods that the server will be providing. The purpose of the `XmlRpcServer` class is to receive and execute XML-RPC calls made by the clients. It can be embedded into another HTTP server, and the simplest possibility is to embed it into the minimal web server, that comes with XML-RPC.

The only non-trivial action is the process of letting the `XmlRpcServer` class know, where to find the implemented methods. This can be done in several ways. It is possible to let the server load a property file, that describes the class with the implementations of methods, or just simply to add the class with the implementations to the server's handler. The code for creating the server part by [5] follows:

```
import java.net.InetAddress;

import org.apache.xmlrpc.common.TypeConverterFactoryImpl;
import org.apache.xmlrpc.demo.webserver.proxy.impls.AdderImpl;
import org.apache.xmlrpc.server.PropertyHandlerMapping;
import org.apache.xmlrpc.server.XmlRpcServer;
import org.apache.xmlrpc.server.XmlRpcServerConfigImpl;
import org.apache.xmlrpc.webserver.WebServer;

public class Server {
  private static final int port = 8080;

  public static void main(String[] args) throws Exception {
    WebServer webServer = new WebServer(port);

    XmlRpcServer xmlRpcServer = webServer.getXmlRpcServer();

    PropertyHandlerMapping phm = new PropertyHandlerMapping();

    /* Load handler definitions from a property file.
     * The property file might look like:
     *   Calculator=org.apache.xmlrpc.demo.Calculator
     */
    phm.load(Thread.currentThread().getContextClassLoader(),
                  "MyHandlers.properties");

    /* You may also provide the handler classes directly,
     * like this:
     * phm.addHandler("Calculator",
     *     org.apache.xmlrpc.demo.Calculator.class);
     */
    xmlRpcServer.setHandlerMapping(phm);

    XmlRpcServerConfigImpl serverConfig =
      (XmlRpcServerConfigImpl) xmlRpcServer.getConfig();
    serverConfig.setEnabledForExtensions(true);
    serverConfig.setContentLengthOptional(false);

          webServer.start();
      }
  }
```

3.3 JAX-RPC

Java API for XML Remote Procedure Call (JAX-RPC [19]) was a first generation of SOAP frameworks in Java. Nowadays it is a part of JAX-WS, which is the second generation of SOAP framework in Java.

Two major implementations exist. One is the implementation by The Apache Foundation: AXIS version 2 and the second is the GlassFish project by Java Community Process, implementing JAX-RPC as part of JAX-WS. I have decided not to use JAX-RPC, but the newer JAX-WS.

The AXIS framework provides a set of tools for developing Web Services. The tool *Java2WSDL* serves to create a WSDL file from a Java interface. The tool *WSDL2Java* creates a stub, a skeleton and a Java interface from a WSDL file. The framework supports different MEPs and transport over various protocols such as SMTP and FTP. The support for the Apache Tomcat server is great and deployment is fairly easy. AXIS extends the JAX-RPC specification by great measures and offers a wide range of additional features that are not in the JAX-RPC specification.

The JAX-RPC specification basically defines the following: a WSDL to Java mapping, Java to WSDL mapping, SOAP binding, JAX-RPC core APIs, models for Service Client and Endpoint, Service Context, SOAP message handlers and other implementation details.

3.4 JAX-WS

As mentioned before, JAX-WS is a part of the GlassFish project by Java Community Process. The JAX-WS specification defines what has been already defined in the JAX-RPC definition and adds to it many new features to it.

JAX-WS uses *Java annotation* to simplify the development of both the client and the server. It uses JAXB for XML to Java and Java to XML bindings instead of private binding facilities as in JAX-RPC. JAX-WS supports asynchronous calls, non-HTTP transport, direct access to underlying message exchange and an enhanced session management.

3.4.1 Server Side Development

As JAX-WS is fully integrated into the NetBeans IDE, the development of a Web Service is really easy. It is accomplished by following these few steps:

1. Create a *Web Application* project
2. Right-click on the created project and select *New – Web Service*
3. Create operations of the newly created *Web Service* in the graphical editor.
4. Right click on the Web Service and select *Deploy*
5. The Web Service is deployed and running on the pre-selected web server

3.4.2 Client Side Development

The client creation is also fully automated in the NetBeans IDE. By following these few steps a client is created:

1. Create a *Java application* project
2. Right click on the created project and select *New – Web Service Client*

3. Choose a *WSDL file* for a client to be created from
4. Right-click inside any class of the created project and select *Web Service Client Resources – Call Web Service Operation*, choose a service operation to be invoked
5. A method call is inserted into the chosen place

```
try { // Call Web Service Operation
  testws.TestService service = new testws.TestService();
  testws.TestService port = service.getTestServicePort();
  // TODO initialize WS operation arguments here
  java.lang.String par = "";
  // TODO process result here
  java.lang.String result = port.test(par);
  } catch (Exception ex) {
  // TODO handle custom exceptions here
}
```

Chapter 4 DISTRIBUTED ALGORITHMS

A distributed algorithm is an algorithm designed to run and function in a distributed system. A distributed system is a computer application, where several computers or processors cooperate in some way. This includes networks of different kinds (local or wide area networks) and multiprocessors systems. We will refer to the computers or processors (or even processes) as to nodes. The nodes have to be *autonomous* – have their own control and have to be *interconnected* to allow the exchange of information. This is a definition according to [30].

There are distributed algorithms of different kinds. They vary in different attributes according to [24]:

- *The interprocess communication (IPC) method*: most common algorithms use access to *shared memory* or *sending point-to-point* or *broadcast messages*.
- *The timing model*: basically processes can be *synchronous* or *asynchronous*, but algorithms with different level of synchronization exist and are called *partially synchronous.*
- *The failure model*: the algorithm may have *no fault tolerance* or tolerance to failures of some degree. Most commonly it is the tolerance to *stopping failures* (where process just stops) or to more severe *Byzantine failures*, where process can behave arbitrarily. Faults may occur also on communication channels as *message loss* or *duplication*.
- *The problems addressed*: typical problems addressed are resource allocation, communication, consensus, database concurrency, deadlock detection, global snapshot, synchronization

We will be focusing on implementing only a small subset of distributed algorithms known. These are the *asynchronous algorithms* based on *message sending without any failures* (no process or communication failures). In the two implemented algorithms we will focus on

resource allocation and *leader election* problems. Formal definitions of these problems are provided in the later chapters focused on particular implementations.

The implementation of other types of distributed algorithms in the client/server architecture based environment is rather difficult and we will not be focusing on it. In fact, only partially synchronous or asynchronous distributed algorithms based on message sending with different failure models can be implemented directly. The synchronous algorithms and algorithms based on memory sharing have to be simulated.

4.1 Formal Description

We have decided to use a formal description as defined by [24]. The distributed system is formally described as a graph or digraph (directed graph) $G = (V, E)$, where V is a set of nodes (or processes, or computers in the network) and E is a set of (directed) communication channels between these nodes. We use letter n to denote the $|V|$. We use the notation $out\text{-}nbrs_i$ and $in\text{-}nbrs_i$ to denote the outgoing and incoming neighbors in the digraph, or simple $nrbs_i$ in the graph. We suppose to have some fixed message alphabet M, and *null* as a placeholder for the absence of a message. We model the processes and the channels as I/O (input/output) automata.

With each process (node) $i \in V$, we associate an I/O automaton, P_i. P_i has defined input and output actions to communicate with an external user. Additionally P_i has outputs of the form $send(m)_{i,j}$, where j is an outgoing neighbor of i and m is a message, element of M. P_i has additional inputs of the form $receive(m)_{j,i}$, where j is an incoming neighbor of i. Except for these restriction P_i can be an arbitrary I/O automaton. An I/O automaton A consists of five components:

- $sig(A)$, a signature is a triple consisting of three disjoint sets of actions: the input actions, $in(sig(A))$, the output actions, $out(sig(A))$, and the internal actions, $int(sig(A))$. We define locally controlled actions as $local(sig(A))$ to be $out(sig(A)) \cup int(sig(A))$, $acts(sig(A))$ to be all actions of $sig(A)$.
- $states(A)$, a (not necessarily finite) set of states.
- $start(A)$, a nonempty subset of $states(A)$ known as the start states or initial states.
- $trans(A)$, a state-transition relation, where $trans(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$. This relation is described in a precondition effect style. The code specifies the condition, under which the action can take place, as a predicate on the pre-state s.

Then it describes the actions that change *s* to *s'*. The code gets executed indivisibly, as a single transaction.

- *tasks*(*A*), a task partition, which is an equivalence relation on *local*(*sig*(*A*)).

The channels are modeled as simple I/O automata. We will consider only reliable FIFO channels. Let communication channel $C_{i,j}$ be such a channel. Let *M* be a fixed message alphabet. Then the I/O automaton looks like this (*start*($C_{i,j}$) are described in terms of a list of state variable and their initial values):

Communication channel I/O automaton

Signature:

Input:
$send(m)_{i,j}, m \in M$

Output:
$receive(m)_{j,i}, m \in M$

States:

queue, a FIFO queue of elements of *M*, initially empty

Transitions:

$send(m)_{i,j}$
Effect:
add *m* to *queue*

$receive(m)_{j,i}$
Precondition:
m is first on *queue*
Effect:
remove first element of *queue*

Tasks:

$\{receive(m)_{j,i} : m \in M\}$

Algorithm 1: The FIFO channel

4.2 I/O Automata in Client/Server Architecture

The precondition effect style description of I/O automata enables easy conversion into programmer languages. The difficulty for us arises in the necessity to use the client/server architecture. The client/server architecture prohibits direct translation of the I/O automata. The usage of network technologies developed for business application brings different problems. These are described in the next chapter. This section describes possible translation of I/O automata into client/server based programming.

The *receive* and *input* transitions are to be translated into methods of a server (remote methods). These methods change objects containing the implementation of the states. The server has to run in independent threads, in order to be able to react to incoming method calls.

The *states* can be translated as objects containing the declared variables. These can be either passive (not implementing any functionality), or they might react to changes done on them. The translation of some *internal* and *output* actions can be done in the states objects, but these actions must contain preconditions. The reaction on the change of a state would in fact be a check of the precondition and the following action would have to have the necessary effect. As changes on state objects are done from the client and server, any functionality in these objects has to be put into separate threads, in order not to block the part of application that has changed the state.

The *tasks* are translated into a client. The client has to perform *output* and *internal* actions defined in tasks or actively wait for the server to get the *input*. The send and output transitions are translated into the client methods. The client has to check actively the state objects for changes done by the server methods and react to them accordingly. The client usually runs in the main thread of the application, but the output actions are preferably handled in separate threads, as the calls made, are usually blocking in the client/server architecture.

Chapter 5 SOAP VS. DISTRIBUTED ALGORITHMS

The SOAP protocol has been created to be a building stone for the development of Web Services. Its main emphasis is the interoperability of all of the components. The Service Oriented Architecture (SOA) allows for the functionality only in the form of services. SOAP provides the tools that SOA needs.

The distributed algorithms theory does not know the term services (or Web Services). Instead it operates with nodes, located in a graph or a digraph, that communicate with their neighbors. Several problems arise from the differences of these two assumed environments. Other problems are simply caused by the characteristics of the SOAP protocol. The following sections will describe and deal with these problems.

5.1 Data Management

The size of data sets in the implementation of many distributed algorithms may vary a lot. Small data sets can be included into a SOAP message without any problems. The problems arise when the data sets grow. The (de)serialization of data becomes a *performance issue*, the inclusion of large data in SOAP envelope is a problem, due to the restriction on the envelope size imposed by many implementations and due to the overhead it creates (a large data set creates even larger XML data).

Unfortunately, there is no way to solve this problem entirely. More powerful XML parsing engines are being developed and used in each new generation of SOAP frameworks in order to deal with the (de)serialization performance problem. The envelope size limitation can be addressed by dividing the remote call into two calls, but this brings a huge overhead.

5.2 Asynchronous Calls

SOAP was primarily designed to handle the *synchronous communication*, although it supports even asynchronous calls. In distributed algorithms the synchronous interaction model might not be desired or possible many times, as certain computation might take up a long time.

Some implementations impose a time limit on the wait for a response, which might prove to be a problem, as a client might signal an error on overdue calls. The problem also arises when an overly large SOAP message is sent, as it might take a long time to transport it.

Luckily, this problem has been solved in the newest generation of SOAP frameworks, as the support for asynchronous calls was added. Also the timeouts can be set to a higher value.

5.3 Performance

The main performance issue arises from the usage of *XML messages* and the need to parse them. According to [20] up to 27% of the server response time is needed for the XML serialization and deserialization. The XML deserialization is done by XML parsers and the speed of the XML parsers varies a lot as shown by [14]. The XML parsers based on the DOM architecture are performing very poorly, when it comes to handling large data structures.

Another factor contributing to the overall low performance is the usage of *HTTP*. Unfortunately, the HTTP protocol, as the main transportation protocol, is not very suitable. HTTP is in fact an *application protocol* and not a transportation protocol. This contributes to the overhead that the usage of HTTP brings. It is not possible to optimize the data size (so it fits in a packet) as a HTTP packet might be divided into several TCP packets.

The data overhead caused by the usage of HTTP instead of TCP or UDP is somewhere around 50% and the response time is up to three times higher compared to UDP, according to Phan et al. [27].

In the last few years the performance of different frameworks has been improved, mostly by improving the XML parsers, using SAX based XML parsers, allowing the usage of different protocols (SOAP-over-UDP, SOAP-over-TCP). Never-the-less, SOAP technology is not build to bring high performance and simple RPC brings always a higher performance mainly because it does not need to deal with the XML serialization, as shown by [20].

5.4 Client/Server as P2P Node

The communication in the client/server architecture is based only on the request/respond message exchange pattern (MEP). The client/server architecture strictly separates the roles of the requester (client) and the responder (server). On the other hand the peer-to-peer (P2P) architecture is based on independent nodes, each fulfilling the role of the server and the client alike. The MEPs in use in P2P architecture vary greatly and are not restricted to the request/respond MEP.

The P2P architecture can be simulated with the usage of the client/server technologies by *combining a server and a client into one*. Such entity from outside behaves like a node and internally uses the server and client technologies running concurrently in multiple threads.

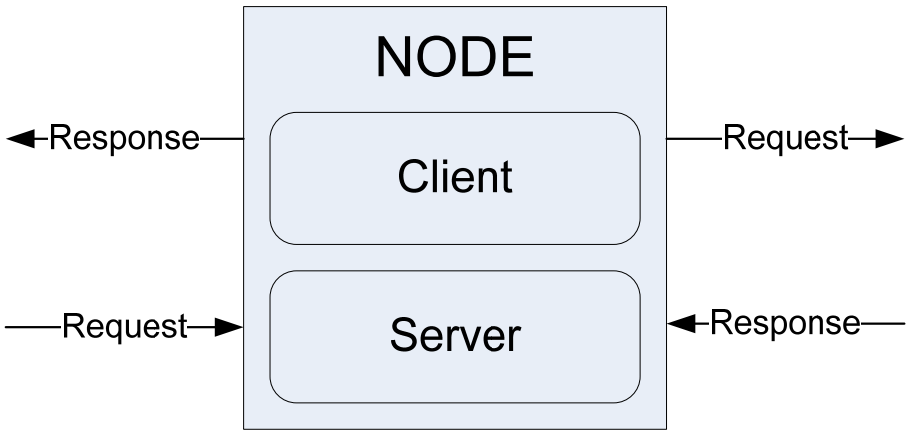


Figure 5: A P2P node simulated in the client/server architecture

A problem arises when a different MEP than the request/respond is required. With a set *request/respond pattern* it would be impossible to emulate certain other MEPs without a considerable overhead. For example, to emulate a simple *request pattern* with the request/response pattern would mean the need to send an empty response. So the *overhead* to sending one message is 100% - a sending of another message.

The newest generation of SOAP frameworks implements not only the request/response MEP, but also other MEPs. Most of the frameworks can also provide a suitable environment for a client/server combined application to be developed and run.

5.5 WSDL Target Endpoint Problem

Allow us to reiterate that according to WSDL each WSDL file contains a *service element* that contains one or more *port elements*. Each port element is an abstract *endpoint* defined by associating a network address with a reusable binding. This constitutes a problem for the distributed algorithm computing.

The usual scenario encountered in the distributed algorithm computing is a network of nodes of an *unknown topology*, *size* and *diameter*. The nodes presume only the knowledge of the existence of their neighbors. Many times not even the direction of the neighbors is to be known. The WSDL file of every node is essentially the same, as they have the same functionality, except for the port definition. The port definitions contain different addresses for each node. With such premises it is not possible to have a *common WSDL file* for every

node, as it would have to contain the list of ports comprising the addresses of every node about to participate in the distributed algorithm.

The solution is to build the client part of each node from a default WSDL file with some *default port* defined. Then at runtime change the port address attribute, defined in the client, just before invoking a method. Each node will use the same client to reach each of its neighbors. The change of a port during runtime is supported in most of the frameworks, but it is not desirable or advisable.

5.6 Network Topology

As mentioned previously the distributed algorithms computing scenario assumes for each node the knowledge of the neighboring nodes in the network. In present switched networks, the topology of the network is usually a lot different from what is assumed by distributed algorithms. Mainly a node does not have any knowledge about any 'neighboring' nodes.

In fact, there are usually *no neighboring nodes* as each node is connected to a switch. It is possible talk about a 'neighborhood' or a local area network (LAN), as a network 'hidden behind a switch'. One of the usual characteristics of a LAN is that each node can be reached by a *broadcast* within the LAN. The broadcast is nowadays by default stopped by routers. Any other configuration of routers could cause serious problems in the network.

As for the simulation of the environment for a distributed algorithm computing, we have to consider two different cases. The first case is that nodes participating in an algorithm computation are not in a single LAN, or their location is unknown or wide spread. That is a *large scale* case. The second case that all nodes to participate are in a single LAN, or in neighboring LANs, if even the connecting router is participating in the computation of the algorithm. That is the *small scale* case.

5.6.1 Large Scale Solution

The Universal Description Discovery and Integration (UDDI) [10] protocol provides a neat utility for discovering Web Services. It serves as a *catalog* of registered Web Services. A Web Service can register into a UDDI server. Afterward, it is possible to search for this service and download its WSDL file.

For the large scale scenario of the distributed algorithm computing, it is possible for nodes to register at a chosen UDDI server. Then search the UDDI server for a certain service type (the same as the node just registered) and download WSDL files of those services. These WSDL files would be used just to gain the knowledge of ports of those other services. These

ports would become addresses of the "neighboring" nodes in the distributed computing. This of course requires access to a *private UDDI server*.

The proposed solution has few more problems. The first node to register into a UDDI server would not discover any 'neighboring' nodes and the last service registered would have all nodes as 'neighbors'. This can be solved by all nodes sending the list of their 'neighbors' to all of their 'neighbors' after the end of the discovery phase of the distributed computing. A *full graph* would be formed by this action.

Another problem is the *synchronization*. All nodes have to be synchronized as when to end the discovery phase and be ready to start the algorithm computation. A solution to this depends mainly on the purpose of the algorithm. One of the solutions might be a fixed time interval for renewing the graph (repeat the sending of the list of 'neighbors') then repeat the algorithm. While the registration to the UDDI server would be possible at any time, only the nodes, which have registered before the fixed time, would participate in given 'round' of the computation.

5.6.2 Small Scale Solution

In the case of the small scale environment it is not necessary to employ such a heavy weight solution as an UDDI server. As all the nodes are within the range of a *broadcast* or at least they are connected by nodes in the range of different local broadcasts, it is easy to use broadcasting as a discovery method.

Each node broadcasts its address and receives a broadcast from other nodes in the range. Each node creates a list of addresses it received in the broadcasting (or discovery) period. After the end of its broadcasting period it sends to each discovered neighbor a confirmation that it is ready to participate in the computing of the algorithm. Each node waits to receive a confirmation from each of its neighbors and only then it starts the computation of the distributed algorithm.

The *UDP broadcast* is mostly *not supported* by the SOAP frameworks. Therefore is it necessary to implement broadcasting using directly the UDP protocol. The messages to be broadcasted do not have to be SOAP messages as they will not be received and processed by a SOAP server, but rather by a private server implemented to receive the UDP broadcast.

5.6.3 Other Solutions

It is, of course, possible to simulate the environment for the distributed algorithm computing in many ways. The two previously mentioned solutions are just examples and many others are possible, depending on the distributed algorithm and the network environment.

One solution is to provide each node at start with a *list of its neighbors*. This list can be provided *manually* or by some completely *independent application*.

The topology of a *connected graph* is not the only topology that can be simulated in today's network environment. Sometimes it might be necessary or desired to simulate topologies like *ring*, *torus* or *full graph*, depending on the distributed algorithm in mind.

Chapter 6 XML-RPC LEADER ELECTION IMPLEMENTATION

We have implemented an algorithm solving the problem of a leader election in an asynchronous network environment. The distributed algorithm is based on the principle of *broadcast* and *convergecast* and is described in details in the first section. The implementation of this algorithm is done by using the XML-RPC implementation from The Apache Foundation in Java language. The second section is concerned about the problems that had to be solved in order to implement this algorithm as well as with other implementation details. The third section presents the performance results from the testing of the XML-RPC protocol.

6.1 Problem Definition

A leader election algorithm solves the problem of *breaking symmetry* in a distributed environment. In the distributed environment it is sometimes necessary to have a *central coordinating node* (a leader), but it might not be possible to designate it manually. As the processes at each node are identical, with the exception of their universal identifiers (UIDs), it is necessary to have a leader election in order to decide on who is the leader. By the end of the algorithm, every node has to know the UID of the leader and the leader must be aware of the fact that it is actually the leader.

6.2 Algorithm Description

We have implemented a leader election algorithm for the asynchronous network environment. This algorithm is based on a *broadcast* and a *following convergecast*. The convergecast is a mirror process to the broadcast, where messages originating from multiple nodes converge to a single endpoint. This endpoint is the originator of the corresponding broadcast. As usual in

the distributed algorithm computing the leader is elected the node with the highest universal identifier (UID).

The algorithm starts with each node (root) making its own spanning tree with child pointers across the network. This is done by sending *setAsChild* messages to all of node's (root's) neighbors and waiting for the response. If a node receives the *setAsChild* message, it responds to it positively, unless it has already received a *setAsChild* message in the process of creating a spanning tree for given root. The node sends a negative response, if it does not send a positive one. After a positive response to the *setAsChild* message a node repeats the process in the same way as root did, except for not sending the *setAsChild* message to its parent. If a node receives negative responses from all of its neighbors, it knows that it is a leaf in the spanning tree of given root. The spanning tree construction is done for each node and the constructions of multiple (possibly all) of the spanning trees are done at the same time.

The second phase of the algorithm begins when a leaf node is determined. At that moment the leaf node begins the convergecast to the root of the spanning tree, in which it is a leaf. The convergecast is begun by sending a *sendMax* message to the parent node in the spanning tree. The *sendMax* message contains the UID of the leaf node that is sending it. If a node receives the first *sendMax* messages from one of its children, it compares the received UID with its own UID and saves the greater one as the maximum. As the node continues to receive the *sendMax* messages, it always chooses the greatest UID to be saved as the maximum. After the node receives the *sendMax* messages from all of its children and had already received responses from all of its neighbors, it sends the *sendMax* message to its parent in the spanning tree. The *sendMax* message contains the greatest UID that has the given node seen. This way the convergecast continues up to the root of the spanning tree. The spanning tree root behaves just like any other node, with the sole exception, that after hearing from all of its children, it does not send any *sendMax* message, but instead it knows the leader, as it has the value of the highest UID.

The convergecast is done simultaneously for each spanning tree that has been constructed in the broadcasting phase. This way by the end of the convergecast phase each node knows the greatest UID and the node with the UID equal to the greatest UID knows that it is the leader.

6.2.1 Formal Description

We assume the existence of an n-node undirected graph $G = (V,E)$. V is a set of nodes; each has its own UID. E is a set of communication channels between the nodes. W is the set of

UIDs of all nodes participating in the algorithm. The *M* set of messages is {("*setAsChild*",*w*) : $w \in W$} $\cup$ {("*sendMax*", *w*, *u*) : *w*, $u \in W$} $\cup$ {("*ack*",*w*) : $w \in W$} $\cup$ {("*noAck*",*w*) : $w \in W$}. We will refer to the set of neighbors of given node as to *nrbs*. The formal description of the leader election algorithm run at each node *i* is as follows:

AsynchronousLeaderElection$_i$

Signature:

Input:
$receive(m)_{j,i}$, $m \in M$, $j \in nbrs$

Internal:
declareWinner

Output:
$startCCast_{i,k}$, $i \in nbrs$, $k \in W$
$continueCCast_{i,k}$, $i \in nbrs$, $k \in W$
$send(m)_{i,j}$, $m \in M$, $j \in nbrs$

States:

leaderElected, a Boolean, initially *false*
$max_k : k \in W$, a set of values $w \in W$, initially *i*,
$parent_k : k \in W$, a set of values $u \in nbrs \cup \{null\}$, initially *null*
$acked_k : k \in W$, a set of subsets of *nbrs*, initially Ø
$children_k : k \in W$, a set of subsets of *nbrs*, initially Ø
$childAck_k : k \in W$, a set of subsets of *nbrs*, initially Ø
for every $j \in nbrs$*:*
    send(*j*), a FIFO queue of messages in M, initially contains a single element ("*setAsChild*",*i*)

Transitions:

send$(m)_{i,j}$
Precondition:
m is first on *send*(*j*)
Effect:
remove first element of *send*(*j*)

receive("*ack*", *k*)$_{j,i}$
Effect:
$acked_k := acked_k \cup \{j\}$
$children_k := children_k \cup \{j\}$

receive("*noAck*", *k*)$_{j,i}$
Effect:
$acked_k := acked_k \cup \{j\}$

receive("*sendMax*", *k, w*)$_{j,i}$
Effect:
$childAck_k := childAck_k \cup \{j\}$
if $max_k < w$ then
$max_k := w$

$declareWinner_i$
Precondition:
$acked_i = nbrs$
$childAck_i = children_i$
Effect:
leaderElected := *true*

receive("*setAsChild*", *k*)$_{j,i}$
Effect:
if $parent_k = null$ then
$parent_k := j$
add ("*ack*",*k*) to *send*(*j*)
for all $l \in nrbs-\{j\}$ do
add ("*setAsChild*", *k*) to *send*(*l*)
else add ("*noAck*",*k*) to *send*(*j*)

$startCCast_{i,k}$ (for $i \neq k$)
Precondition:
$parent_k \neq null$
$acked_k = nbrs - \{parent_k\}$
$children_k = \emptyset$
Effect:
add ("*sendMax*", *k*, *i*) to *send*($parent_k$)

$continueCCast_{i,k}$ (for $i \neq k$)
Precondition:
$parent_k \neq null$
$acked_k = nbrs - \{parent_k\}$
$children_k \neq \emptyset$
$childAck_k = children_k$
Effect:
add ("*sendMax*", *k*, max_k) to
send($parent_k$)

Tasks:
for every $j \in nbrs$:
$\{send(m)_{i,j} : m \in M\}$

Algorithm 2: The Leader Election

6.2.2 Complexity Analysis

The broadcasting phase of each node has the complexity $O(|E|)$, as maximum of four messages ("$setAsChild_{j,i}$", "$ack_{i,j}$", "$setAsChild_{i,j}$", "$noAck_{j,i}$") is send along each edge. $|E|$ might be equal to n^2 in a full graph. As the broadcasting is done by each node, the complexity of the whole broadcasting phase is $O(n^3)$.

During the convergecast phase the convergecast to each node has the complexity $O(n)$ as only one message is send along each edge of the spanning tree. The convergecast is done to each node, so the complexity of the whole phase is $O(n^2)$.

The algorithm has a high message complexity $O(n^3) + O(n^2) = O(n^3)$. On the other hand it deals with the stopping problem commonly encountered in distributed algorithms.

The time complexity of both phases is $O(n(l + d))$, where l is the maximum time between two tasks in each process and d is the maximum network delivery time. Both, the broadcast and the convergecast might travel on the path of maximum of n nodes.

6.3 Implementation Details

The whole implementation handles the leader election as a set of concurrent broadcasts started from different nodes. Each of these broadcasts is followed by a convergecast in the direction of the broadcast originator. A couple, the broadcast and the following convergecast, form a session for the purpose of variable keeping. Each of these sessions is handled by a separate thread. The sending and receiving is done in two completely separate threads.

The XML-RPC protocol supports only the request/response MEP, hence, all the method invocations are blocking. This somehow interferes with the formal specification of the algorithm, as it allowes for a request not be followed immediately by a response. In the implementation, this would require a separate thread for each method invocation. As the number of concurrent method invocations might be equal to the number of neighbors of the node in every separate broadcast (and convergecast, or so called session), the total number of threads would grow to the square of the number of neighboring nodes, which might not be acceptable.

The whole implementation is divided into three packages. The main package `cz.cuni.mff.leaderElection` handles the implementation of the algorithm itself. The other two packages are concerned with the preparation of prerequisites for the main algorithm. The package `cz.cuni.mff.leaderElection.broadcast` serves for the discovery of neighbors and the package `cz.cuni.mff.leaderElection.synchronizer` has the only purpose to synchronize the start of the main algorithm. The synchronizer and the main package use the XML-RPC protocol, while the broadcasting is done on the pure UDP protocol, without any usage of XML.

6.3.1 Package cz.cuni.mff.leaderElection

The `cz.cuni.mff.leaderElection` package is the main package and handles the leader election, the XML-RPC server and client and the embedded web server. The main

principle is to get synchronized with the other nodes after the UDP broadcast is done, and start the leader election by beginning the XML-RPC simulation of the broadcast. The UDP broadcast and synchronization is dealt with in the other packages.

The `Main` class *coordinates* every part of this program. At first the web server (class `WebServ`) is started in a separate thread. This is done to ensure that the program will listen for XML-RPC synchronization calls as soon as possible. Next the UDP broadcast is started by calling a `cast` method on an instance of a `Broadcast` class. After the UDP broadcasting is done, the set of neighboring nodes is known. The `Synchronizer` is setup with the list of neighbors. Next, the session handling is initiated (class `SessionHandler`) and a client (class `Client`) is initiated. The client is used to send out a ready signal to the neighboring nodes and the `Synchronizer` to wait for other nodes to get ready. After the synchronization is done, the leader election algorithm is started in the client.

The `WebServ` class sets up and runs the *embedded web server* (class `org.apache.xmlrpc.webserver.WebServer`). This class also sets up the XML-RPC server within the web server by setting the `Server` class as the handler for incoming XML-RPC requests.

Inside the `Server` class the server (receiving) part of the logic of the leader election algorithm is implemented. Methods `setAsChild` and `sendMax` implement the receiving of the corresponding messages as described in the algorithm description. This class also implements the method `setReady` for synchronization purposes. This method adds the invoker of the method (identified by a UID) to the list of ready nodes by calling the `addReady` method in `Synchronizer`. As XML-RPC does not support any other MEP beside the request/respond MEP, it is necessary to simulate them. The *sendMax* message does not require any response as it is a pure one way call, but it is necessary to send an empty response to it. This is done in the method `sendMax`.

The `Client` class serves few different purposes. The first is that it hides the private class `PhysicalClient`. The `Client` class uses a map of the instantiated `PhysicalClient` classes to serve as a universal client. When the `Client` class is started in a new thread it does the initiation of the broadcast by sending the *setAsChild* messages to all of the neighbors. After the messages are sent, the `Client` thread waits for all the children to respond and it defines the leader as the maximal UID found.

The private class `PhysicalClient` implements the client (sending), but not the logic of the leader election algorithm. Each instance of the `PhysicalClient` serves as a

connection to one of the neighbors. It comprises the methods `setAsChild`, `sendMax`, `setReady` and `execute`, which executes the call on the `XmlRpcClient` object instantiated within this class. These methods have to be declared as `synchronized` as they might be called from different threads at the same time and they are not reentrant.

The `SessionHandler` class is a simple class serving as a container for instances of `Session` classes. The `SessionHandler` does also the initialization of sessions. An instance of the `Session` class stores the variables for a broadcast and convergecast initiated by given node. It implements the rest of the server logic of the algorithm that is not done by the `Server` class. Its main purpose lies in allowing the access to all of the session variables. These are accessed by the `Server` and the `ClientThread` classes. The `Session` class also keeps track of children responses and keeps a status variable `allResponded`, defining whether all the children have already responded.

The `ClientThread` class is instantiated and run in a separate thread by the `Session` class, when a *setAsChild* message is received and the node is set as child of the sender in given session. Most of the client (sending) logic of the leader election algorithm is in this class, except for the initialization of the broadcast. After the separate thread is started, `ClientThread` continues the broadcast and waits for acknowledgements. Decides, whether the node is a leaf in given session and starts a convergecast if it is, by sending the *sendMax* message with its own UID. If the `ClientThread` class decides that it is not a child, then it waits until it hears from all of its children (the receiving is done by `Server`, `ClientThread` just checks `Session`). Afterward it continues the convergecast by sending the *sendMax* message to its parent and finishes its work.

6.3.2 Package cz.cuni.mff.leaderElection.broadcast

The `cz.cuni.mff.leaderElection.broadcast` package is an auxiliary package implementing the UDP broadcast server and client. The purpose is the discovery of neighboring nodes and the announcement of its own existence to them.

The `Broadcast` class serves as the main class of this package. Other classes are not accessed by other packages. The only method is the `cast` method that returns a collection of `Node` classes. This method starts the `Server` class and then `Client` class in separate threads. It waits the preset time and then interrupts the `Client` and the `Server`.

The `Node` class is a very simple class for storing a node's address and port. The port number is not really needed as it is not used anywhere.

The `Client` class has first to compute the broadcasting address in given LAN, as the default 256.256.256.256 may not be supported in every LAN. The broadcasting address can be computed from the *local IP address* and the local *subnet mask*, where *broadcastAddr* = (*localAddr* & *subnetMask*) | (~ *subnetMask*) (& is a bitwise and, ~ is a bit negation and | is a bitwise or). The problem is that the subnet mask can not be determined in the Java environment. It has to be set as an option by the application startup. The only other possibility is to use the default broadcasting address. After the `Client` class determines the correct broadcasting address, it broadcasts in a loop a simple message containing just a word. The broadcasted message does not have to contain the sender's address as it can be determined from the UDP package containing the message. The `Client` fulfills the role of announcing the existence of the node to the other nodes.

The `Server` class is responsible for collecting the addresses of neighbors into a list. This class does it by creating a UDP socket and calling *receive* in a loop. After a message is received successfully, the sender's address is added to the set of collected addresses, in fact it is a set of `Node` classes.

6.3.3 Package cz.cuni.mff.leaderElection.synchronizer

The `cz.cuni.mff.leaderElection.synchronizer` package contains only one class and it is the `Synchronizer` class. The purpose of this package (and class) is to synchronize the start of the leader election algorithm after the broadcast and other preparation is done. The `Synchronizer` class is a singleton class and it is automatically instantiated as soon as it is needed. The `Server` class from the `cz.cuni.mff.leaderElection` package calls the `addReady` method in order to add a neighbor to the list of ready neighbors. The method `allReady` serves for determining whether all neighbors are ready, but the checks are done internally in the private method `checkReady`, where the set of ready nodes is compared to the list of all neighbors of the node, if this list is already available. The list of all neighbors has to be given to the `Synchronizer` class; otherwise it is not possible to synchronize with all neighbors. As the `Synchronizer` might be instantiated before the list of all neighbors is known, method `setUpSynchronizer` serves to hand over the list to the `Synchronizer` class.

6.4 Performance Measurements

In the former chapters, we have several times mentioned the performance issue that is connected with the XML-based remote procedure call (RPC) systems. In this section we will identify the problems causing the performance issues, describe the testing environment, mention some results and do a comparison.

6.4.1 Underlying Problems

The XML-based RPC systems, like XML-RPC, have different disadvantages performance wise in comparison to RPC systems that transport binary data. The *data overhead* caused by the use of XML as the transport format, is very large, as shown by [4]. The XML data can take up anywhere from 6 to 300 times more space than the binary data would. This might not look so significant, as the enlargement of larger data is usually closer the lower boundary of the range (so they are approximately 6 times bigger), but all of it has to be transported over a network, which makes every byte count.

The XML format carries the problem of all text formats. In comparison to binary formats, the data *does not have a fixed length*. For instance, a double has always 64 bits in a binary format, but in the XML format, a "3.1", takes up 3 characters, but "0.234235222" takes up 11 characters. This problem prohibits any optimization of fitting data into packets (of whatever transport protocol is used).

The other problem, that we have already mentioned, is the *usage of HTTP* as a transport protocol. This only adds to the data overhead and even further limits (or prohibits) any possible optimization to be done.

The usage of XML brings up the need of *XML parsing*, where in the past the XML parsers were not known for their overall great performance. The DOM-based parsers were too slow and took too much memory, if they were to handle a bigger XML file. The SAX-based parsers handle the large XML files a lot better. The overall performance of XML parsers has improved in the course of the last few years.

6.4.2 Testing Environment

We have implemented a simple test to measure the speed of the XML-RPC protocol in the implementation by The Apache Foundation. The test was performed on two computers, one AMD Athlon XP 3000+, 1024 MB DDR RAM, serving as a client and the second Intel Core Duo 1.6 GHz, 2 GB DDR2 RAM, serving as a server. The network was 100Mbps LAN, without any noticeable traffic.

The test consists of an application client and server. The test implements six different methods. Four methods are short requests with short responses, but each with a different type. This is to determine whether the speed of *marshalling* depends on the type of the parameter. The types tested are *integer*, *string*, *double* and *Boolean*. The other two methods have a variable size of requests and responses, as each contains an *array of up to 20,000 arguments*. One method handles an array of *strings*, the second one handles an array of *doubles*.

The parameters sent and received are not the same, but are chosen *pseudo-randomly* from a static array of random pre-generated arguments. This way it is not possible for the client or server to cache the messages and this way to optimize the handling of messages. The methods with arrays change only the first attribute pseudo-randomly, the rest is fixed.

The *timestamps* were taken after *each single call* to a method has been made. All unnecessary code was moved out of the timestamp range on the client side. The returned value is not handled in any way, just saved into a variable. On the server side, the methods do not perform any operation; just return a parameter chosen in exactly the same fashion as the parameters in the client. After twenty calls have been made, the test continues with another method. The test was run at least five times.

The server part of the application was not deployed on any production server, but rather used the embedded web server. This did not affect the results in any way as the only traffic was caused by the test client application and all calls were made in succession. The purpose of this test was not to test the server's performance, but rather the performance of the XML-RPC framework.

6.4.3 Results

The results measured are shown in the Table 1. These results will be compared in the next chapter to the results measured for JAX-WS.

The performance of XML-RPC was compared by [4] to the performance of `java.net` package. The size of the data in a XML-RPC message is up to fifty times larger than the size of the same data in the message created by `java.net` package, when we talk about a transfer of small data. The amount of data transferred is approximately six times bigger with XML-RPC than with `java.net` package, considering large data sets. The enlargement of small data by so large magnitude does not yield such a great difference in the performance, as both `java.net` and XML-RPC need just one (or few) TCP/IP package(s) to transfer the data. It is a lot worse with larger data. The time difference between XML-RPC and java.net package grows by a magnitude of an order.

The parameter type in the case of small request and small response does not affect the performance so much. The difference between string and Boolean is only 7.82%. The string is supposedly the simplest type to marshal, as it does not need any processing. The double, on the other hand, is the hardest type to marshal according to [8]. The measured differences are really small and the highest time achieved by string messages is probably caused the slightly bigger message size of string messages in comparison to Boolean messages. The average size of the string was 25.1 characters, where the average size of a Boolean is only 4.5 characters. Also the longest string was over 200 characters long.

The difference between marshalling an array of 10,000 strings and 10,000 doubles is 36.45%. This difference proves that marshalling of doubles is not a trivial task, as the array of strings was in fact a larger chunk of data.

Parameter type	Integer	String	Double	Boolean	Array String	Array Double
Time for 1 call (ms)	7	9	7	5	112	143
Time for 1 call (ms)	5	5	5	4	107	148
Time for 1 call (ms)	10	11	7	12	106	146
Time for 1 call (ms)	5	6	5	5	107	143
Time for 1 call (ms)	6	4	12	11	106	148
Time for 1 call (ms)	9	6	8	4	105	147
Time for 1 call (ms)	7	7	6	4	106	143
Time for 1 call (ms)	6	5	5	5	105	145
Time for 1 call (ms)	12	4	10	4	106	148
Time for 1 call (ms)	5	5	4	4	106	146
Time for 1 call (ms)	5	5	6	4	105	145
Time for 1 call (ms)	5	5	8	5	105	143
Time for 1 call (ms)	7	4	4	7	111	145
Time for 1 call (ms)	5	7	5	4	106	142
Time for 1 call (ms)	5	5	5	4	105	149
Time for 1 call (ms)	5	6	5	5	107	146
Time for 1 call (ms)	4	5	5	8	106	143
Time for 1 call (ms)	4	10	4	5	107	145
Time for 1 call (ms)	4	10	3	10	106	146
Time for 1 call (ms)	4	5	4	5	105	144
Average for 1 call (ms)	6	6.2	5.9	5.75	106.45	145.25

Table 1: The XML-RPC performance measures – parameter type dependency

The time dependency on the number of parameters was measured by increasing the size of the array of parameters in ten steps: 1, 10, 100, 1000, 2000, 3000, 5000, 7000, 10 000, 20 000. These tests were done for arrays of integers and doubles. The average size of a double value in the text form was 10.4 characters, where the average size of a string was 25.1 characters.

The time seems to grow linear to the number of parameters once the size of the array reaches one thousand parameters. Before that, the time seems to grow logarithmic to the

number of parameters. This is caused by the added overhead, as the message containing the parameters contains a large amount of data that is constant relatively to the number of parameters contained in the message. The difference between the string and double parameter type becomes apparent with the increasing number of parameters. For a larger number of parameters the time elapsed for an array of strings is 35% shorter than the time elapsed for an array of doubles.

The dependencies are shown on the two graphs below. The table containing the exact values is located below the graphs.

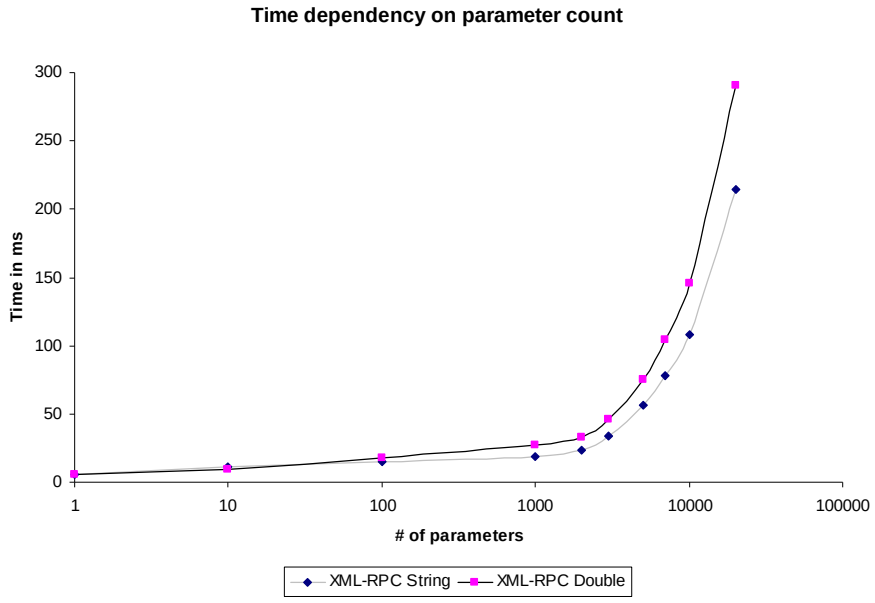


Figure 6: The time dependency on the parameter count in logarithmic scale

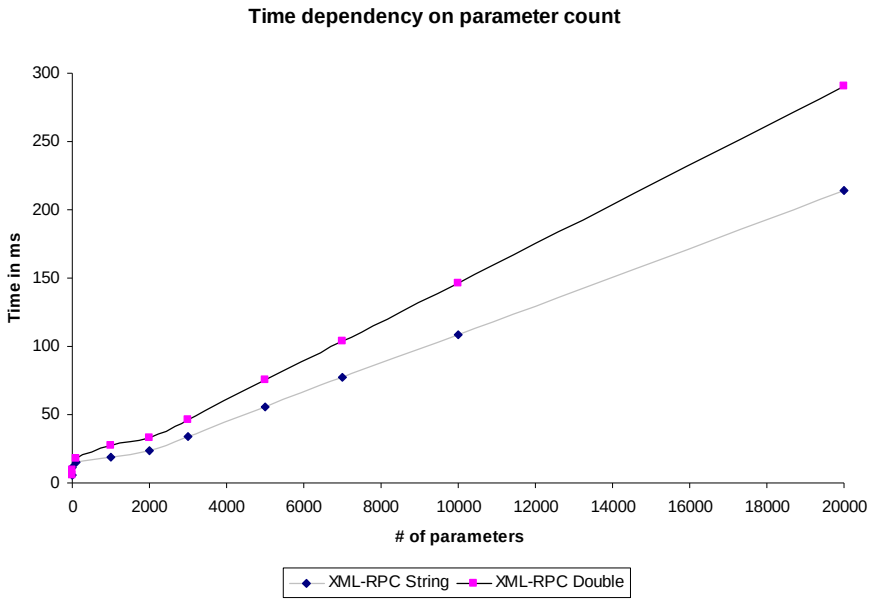


Figure 7: The time dependency on the parameter count in linear scale

	Values displayed are an average time in ms				
Number of parameters	1	10	100	1000	2000
Array of strings	5.40	11.24	14.97	19.18	23.67
Array of doubles	5.33	9.81	17.50	26.97	33.29
Number of parameters	3000	5000	7000	10000	20000
Array of string	33.95	56.08	77.65	108.44	214.54
Array of doubles	46.26	75.26	104.08	146.03	290.49

Table 2: The XML-RPC performance measures – size dependency

Chapter 7 JAX-WS RESOURCE RESERVATION IMPLEMENTATION

We have implemented a distributed algorithm for the resource-allocation in an asynchronous network environment. The distributed algorithm is derived from the *Coloring algorithm* for shared memory systems.

The Coloring algorithm for shared memory systems is based on coloring the *resource graph* for given resource requirements. The processes seek required resources in order based on the coloring in order *to avoid the deadlock*.

Our resource reservation algorithm simulates the Coloring algorithm by node (or process) simulating one process of the shared memory algorithm plus some subset of the resource processes. The reservation is done by sending a message to a process that simulates the desired resource.

The implementation of this algorithm was done by using the JAX-WS implementation by the GlassFish project. The second section describes the implementation details together with the problems that had to be solved to simulate the P2P environment by using JAX-WS. The third section presents the performance results of the JAX-WS implementation of Web Services. These results are briefly compared to the performance results gained from the XML-RPC testing.

7.1 Problem Definition

The resource-allocation problem is a problem commonly encountered in the distributed computing. The scenario is that several different processes (nodes) are trying to reserve (for exclusive use) some resources in order to execute whatever they need to execute. As each resource can be held just by one process at the time, some strategy is needed for the order of reservations for the processes to be able to execute their tasks. Without any strategy it is possible that the scenario would end in a deadlock.

The deadlock is easily explained on a simple scenario with two processes and two resources. Let us have processes *A* and *B* and resources *1* and *2*. Both processes (*A* and *B*) need both resources (*1* and *2*) for the execution of their tasks. If the process *A* reserves the resource *1*, while the process *B* reserves the resource *2*, then if the process *A* tries to reserve the resource *2*, it fails and has to wait until the resource *2* is released. Meanwhile, the process *B* tries to reserve the resource *1*, but it fails too, as it is currently held by the process *A*. Both processes (*A* and *B*) continue trying to reserve the resource currently held by their opponent and neither of them is able to finish its task.

More formally, a *deadlock* is a situation where oriented graph $G = (V, E)$, where V are resources and processes, and edges exist where a process is trying to reserve a resource (from the process to the resource) and a resource is held by a process (from the resource to the process), contains a circle.

An algorithm solves the *resource-allocation problem* if in any execution of the algorithm the *deadlock situation can not occur*. This way it is ensured that tasks of each process will end in a *finite time*.

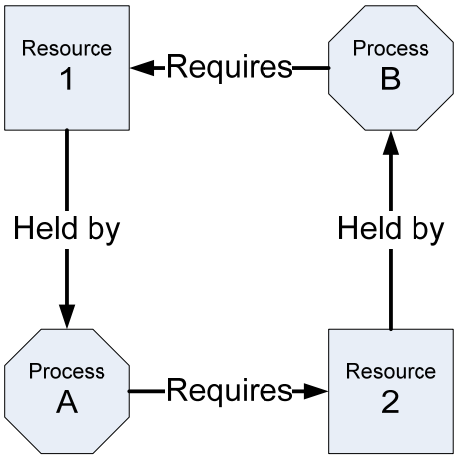


Figure 8: A graph of a deadlock situation

7.2 Algorithm Description

We have implemented a *resource reservation* distributed algorithm based on the coloring of the resource graph. This algorithm is a *simulation* of a similar algorithm for the *memory sharing* environment. The main purpose of the algorithm is to prevent the occurrence of deadlocks during the resource allocation, this characteristics is called the *lockout-freedom*.

A resource specification by [24] for n processes consists of a universal finite set R of objects known as *resources* and for every $i, 1 \leq i \leq n$, a set $R_i \subseteq R$. In other words, every process has a set of resources that it requires, which is a subset of all available resources.

A resource graph is a graph of a particular resource specification, where nodes represent resources and where is an edge from one node to another exactly if there is some process that uses both associated resources.

N, set of nodes $\{N_1, N_2, N_3, N_4\}$

R, set of resources $\{r_1, r_2, r_3, r_4\}$

Resource specification:

N_1 : $\{r_1, r_2\}$

N_2 : $\{r_1, r_3\}$

N_3 : $\{r_2, r_4\}$

N_4 : $\{r_3, r_4\}$

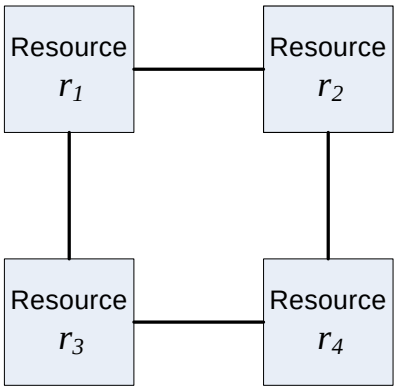


Figure 9: A resource graph example

A coloring is a *process*, where each node of a graph is given a color and no adjacent nodes have the same color. The problem is to *minimize* the number of used *colors*, as the trivial solution of assigning each node a different color would usually not serve the purpose, for which the coloring is constructed. The problem of obtaining minimal number of colors is a *NP-complete* problem.

The resource reservation algorithm, that we have implemented, first builds a resource graph. Then it colors the resource graph by a simple *greedy approach*. The greedy approach consists of taking an arbitrary node and assigning it a color. Then take all of its neighbors one by one assigning them a newly determined color. The determination of a new color is simple. If an already used color that is not used by the node's neighbors exists, then the node is assigned such color. If there are more colors that can be assigned, the assigned color is chosen in any way. This coloring process is then repeated for the neighbors. As the resource graph might not be connected, it is necessary to repeat this whole process for each node that does not have any color yet.

The coloring provides a *partial ordering* on the resources. This partial ordering has to be extended into a *total ordering*. This can be done by *topologically sorting* the resource nodes.

The resource reservation of each node is done in the order set by the total ordering. This way no deadlock situation can occur.

7.2.1 Formal Description

We assume an existence of a *n*-node undirected graph $G = (V,E)$. V is a set of nodes. E is a set of communication channels between the nodes. W is the set of UIDs of all nodes participating in the algorithm. R is a set of resources and R_i, where $R_i \subseteq R$, is the set of resources located at the node i. D_i is a list of resources that the node i requires in order to execute its task. The resource distribution function $d()$ maps a resource the to node at which it is located, ($d(r) = i : r \in R, i \in V$). C is the total ordering of resources computed from the coloring of the resource graph. The M set of messages is {("*reserve*",r) : $r \in R$} $\cup$ {("*acquired*",r) : $r \in R$ } $\cup$ {("*release*",r) : $r \in R$}. We will refer to the set of neighbors of given node as to *nrbs*. The formal description of the resource reservation algorithm run at each node i is as follows:

ColoringResourceReservation$_i$

Signature:

Input:
receive$(m)_{j,i}$, $m \in M$, $j \in nbrs$

Internal:
performTask
reserve

Output:
makeReservation
confirmReservation$_r$, $r \in R_i$
release
send$(m)_{i,j}$, $m \in M$, $j \in nbrs$

States:

resourcesQueue$_k$ **:** $k \in R_i$, a set of queues, each queue initially empty
canAcquire$_r$: $r \in R_i$, a set of Booleans, initially all *true*
for every $j \in nbrs$:
    send(j), a FIFO queue of messages in M, initially empty
resAcquired, a set of resources r, $r \in R$, initially Ø
startRerserving, a Boolean, initially *false*
reserving, a Boolean, initially *false*
taskDone, a Boolean, initially *false*

Transitions:

send(m)$_{i,j}$
Precondition:
m is first on *send*(j)
Effect:
remove first element of *send*(j)

receive("*reserve*", r)$_{j,i}$
Effect:
add j to $resourcesQueue_r$

receive("*acquired*", r)$_{j,i}$
Effect:
reserving := *false*
add r to *resAcquired*

receive("*release*", r)$_{j,i}$
Effect:
remove j from top of $resourceQueue_r$
$canAcquire_r$:= *true*

$confirmReservation_r$
Precondition:
$resourceQueue_r \cup \emptyset$
$canAcquire_r$ = *true*
Effect:
$canAcquire_r$:= *false*
add ("*acquired*",r) to *send*(*d*(k)), k is the top of the $resourceQueue_r$

reserve
Effect:
order D_i by C
startReserving := *true*

makeReservation
Precondition:
startReserving = *true*
reserving = *false*
$D_i \neq \emptyset$
Effect:
reserving := *true*
remove first r from D_i
add ("*reserve*",r) to *send*(*d*(r))

performTask
Precondition:
reserving = *false*
$D_i = \emptyset$
Effect:
taskDone := *true*

release
Precondition:
taskDone = *true*
Effect:
for every $r \in resAcquired$
add ("*release*", r) to *send*(*d*(r))

Tasks:
{*reserve*}

Algorithm 3: The Coloring Resource Reservation

7.2.2 Complexity Analysis

The complexity of resource-allocation algorithms is not measured in the number of messages sent (as this is fairly constant), but in the time that is needed for a process to acquire all of its resources and be ready to start performing its task. The *time complexity* depends on the *process delay* time (the delay between two requests for resources), *message-delivery* time, the *performed task* time, the *number of colors* and the *maximum number of processes* trying to access one resource.

Let l be the upper bound on the process step time, let c be the upper bound on the time of the task that is performed after the resources are acquired and let k be the number of colors

needed to color the resource graph. Also let *m* be the maximum number of processes that require any single resource. Then worst-case time bound is $O(m^k c + km^k l)$ according to [24].

7.3 Implementation Details

The resource reservation algorithm implementation with the Web Service technology has to do a lot more than that is described by the algorithm. To meet the premises assumed by the algorithm a non-trivial amount of work has to be done.

First, the problem of an *unknown network topology* has to be solved. This can be done in different ways, as we have described previously. We have decided to use the same *UDP broadcast* as with the XML-RPC implementation, as this enables an easy way to test this algorithm. The large scale solution is only suitable for a permanent installation (or so to say deployment) of the application.

Then the *resource distribution* across the network has to be done. Each node claims to have a set of resources and has to let know the other nodes about them, that they might reserve them. After the node gains the knowledge about all the resources available in the network, it has to choose those that it wants to reserve and it has to let other nodes know about its choice of resources.

Afterward, the node has finally enough information to *build the resource graph* and run the *coloring* algorithm on it. Each node has to make exactly the same coloring, because the total ordering of the resources acquired from the coloring has to be the same.

Only after the coloring is known, the *resource reservation* can begin. With the resources acquired, the *critical tasks* can be performed. At last the resources are released. As this implementation is only an example and not a real-life critical application, the resources are only integer numbers and no real task is performed.

The implementation was done using the NetBeans IDE for Web Services development. The JAX-WS generates quite a few classes for the Web Service client. This implementation uses the default settings and does not perform any changes in the generated files.

The implementation of the leader election algorithm consists of four packages. The main package `cz.cuni.mff.coloring` consists of a Web Service implementing class and two other classes implementing auxiliary objects for the Web Service. The `cz.cuni.mff.coloring.client` package implements the client for the Web Service together with the resource reservation algorithm. The package `cz.cuni.mff.coloring.client.algorithm` is concerned with the preparation of

the resource graph and with its coloring. The `cz.cuni.mff.coloring.client.broadcast` package handles the UDP broadcast that serves to discover the neighborhood and to let the neighborhood know about the existence of the node.

7.3.1 Package cz.cuni.mff.coloring

The `cz.cuni.mff.coloring` package contains the main server class `ColoringServer`. It defines and implements the Web Service. This class implements two methods for the resource reservation and one method needed for the prerequisites' preparation. The methods `resourceReservation` and `resourceRelease` use the `Queue` class for keeping track of who requires the local resources. The `resourceReservation` method blocks the caller. It first puts the caller's UID into the queue and then actively waits for it to appear on the top of the queue. The `addRequirements` is an auxiliary function for collecting the requirements of other nodes. It uses the `NodeList` class to collect the requirements. The `activate` method is special and serves to initialize the whole algorithm. It starts and runs the `cz.cuni.mff.coloring.client.Main` class in a separate thread. This method has to be called in order to start the application within the deployed Web Service.

The `NodeList` is a singleton class serving actually two purposes. The first it is to collect all requirements from other nodes into a single set. The second is to synchronize the application run after the broadcast phase, as every node has to wait for the complete list of requirements in order to prepare the requirement graph. This is secured by `NodeList` (method `isListReady`) comparing the list of nodes that have already claimed their requirements with the list of all known nodes. The list of all known nodes is known only after the broadcast is done and `NodeList` has to be additionally informed about it (method `setFullSet`).

The `Queue` class is a simple singleton class providing access to a set of queues. For each local resource the algorithm needs one queue. This class provides basic methods as `peek`, `poll` and `add`.

7.3.2 Package cz.cuni.mff.coloring.client

The `Main` class coordinates all activities done by this application, except for starting the Web Service. As this application is written as a web application and it is distributed as a *.war* file, it has to be deployed to a web server.

First, the broadcast is initialized and the list of all participating nodes is collected. The `NodeList` class is given the list of the participating nodes. That is followed by generating local resources and initializing the `Queue` class with these resources. Then the requirement distribution is done and the list of requirements of the node is generated by the `ResourceDistribution` class.

Next, the resource graph is prepared and a coloring is done. The colored resource graph finally yields the total ordering (provided by the `Coloring` class). Afterward, everything is ready for the resource reservation algorithm. This is implemented by a *for cycle* through the ordered list of the required resources. The `ColoredResourceReservation` class is instantiated with the address of one of the required resources and on it a reserve call is made. This call is blocking. After it succeeds, the reservation is added to a list of reservations.

The application does not do any task after the resources are reserved, but immediately starts to release them. The resource releasing is done in a *for cycle* too. This would be preferably implemented with SOAP-over-UDP with the request MEP. That way no blocking would occur, as no request/response MEP is required. The usage of SOAP-over-UDP after the required request/response pattern would create larger overhead (both programmers and computing wise) than the simple empty response and the blocking call.

The `ColoredResourceReservation` class serves the only purpose to communicate with the Web Service of the node that it has been initiated to communicate with. This way the connection to a certain node does not have to be initiated multiple times, and the generated client classes are instantiated only once for each connection.

The change of the *endpoint* has to be done when instantiating a generated client, as the client is generated with a local endpoint in the WSDL file. The change is done by the shown code:

```
String url = “http://www.example.com/ColoringServer”
ColoringServerService service = new ColoringServerService();
port = service.getColoringServerPort();
BindingProvider bp = (javax.xml.ws.BindingProvider) port;
Map<String, Object> context = bp.getRequestContext();
context.put("javax.xml.ws.service.endpoint.address", (Object)url);
```

The `ResourceControl` class solves the problem of uniqueness of resources. As the resources are generated as random integer numbers, it is possible, although highly unlikely, that two different nodes may generate the same resource as their own. The `ResourceControl` checks for multiplicities in the resource list and erases every occurrence of a resource, if it is declared more then once. The method `getLocalResource` checks the list of local resources against the list of all resources for multiplicities. Another solution to the *uniqueness problem* is to take as an ID of a resource not only the generated integer, but a tuple consisting of the generated integer and its generator's ID.

7.3.3 Package cz.cuni.mff.coloring.client.algorithm

The `cz.cuni.mff.coloring.client.algorithm` package handles the coloring of the resource graph. The `Coloring` class implements the *greedy coloring algorithm* and determines the order of the resources. Methods `computeColoring`, `color` and `determineColor` are used for the computation of the greedy algorithm. The `computeColoring` method invokes the `color` method on each node of the resource graph. The `color` method adds a color to an uncolored graph node and recursively calls itself on neighbors of the uncolored graph node. The `determineColor` method is called by the `color` method to determine a suitable color for the uncolored graph node. This is done by checking colored neighbors of the node in order to determine the lowest unused color (as a color is just an integer). If all already used colors are used by the neighbors a new color is put into use. The `getOrder` method sorts the colored nodes on the color. This way a total ordering is created and the partial ordering determined by the coloring is preserved.

The `RequirementDistribution` class serves three roles. At first it creates the local requirements of the node. In order to do this the `makeDemandList` method randomly ventures through the resource list and adds a randomly chosen number (from 1 to 10) of randomly chosen resources. The second role is the distribution of such created requirements

to the other nodes. This is done in the `sendResourceRequirements` method by changing the endpoint context on a single Web Service client in a *for cycle*. This is not advisable, but we have not encountered any problems with this solution. The third role is to prepare the resource graph, but first it is necessary to wait until all data is ready for the creation of such graph. The waiting is done in the `prepareResourceGraph` method together with some data gathering. The graph making is done in the `makeGraph` method.

The classes `ResourceGraphNode` and `ColoredResourceGraphNode` are just auxiliary classes for storing the graph and colored graph nodes.

7.3.4 Package cz.cuni.mff.coloring.client.broadcast

The `cz.cuni.mff.coloring.client.broadcast` package contains almost the same implementation of the UDP broadcast as was used in the XML-RPC implementation of the leader election algorithm. For more details please refer to the **Package cz.cuni.mff.leaderElection.broadcast** section.

The difference is in the distribution of local resources in the generated broadcasted message. The `Client` class generates and sends a random number (from 1 to 10) of random integers. These integers are to be the local resources of the node. The `Server` class is modified to read the contents of the received message and add the resources into the `Node` class, which has been modified to contain them. The port number, on which the web server hosting the Web Service runs at each node, is distributed within these messages too. The `Node` class stores these port numbers as they are later used to connect to the Web Services.

7.4 Performance Measurements

The performance tests for JAX-WS Web Services are similar to the test done for the XML-RPC protocol. This section is concerned mostly with presenting the JAX-WS performance test results and comparing them to the results of the XML-RPC framework test.

7.4.1 Underlying Problems

Web Services as a RPC framework add few other issues to those of XML-RPC. The usage of SOAP creates even greater data *overhead* than the XML-RPC protocol. The overhead created by the SOAP protocol is best illustrated below. This is a simple Google Search messages generated by gSOAP by [2]. It carries the *three key words* for the search and few other parameters.

```
POST / HTTP/1.1
Host: api.google.com
User-Agent: gSOAP/2.7
Content-Type: text/xml; charset=utf-8
Content-Length: 664
Connection: close
SOAPAction: "urn:GoogleSearchAction"

<?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Envelope xm
lns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xm
lns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" xm
lns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:x
sd="http://www.w3.org/2001/XMLSchema" xmlns:api="urn:Google
Search"><SOAP-ENV:Body SOAP-ENV:encodingStyle="http://schem
as.xmlsoap.org/soap/encoding/"><api:doGoogleSearch><key>XXX
XX</key><q>Binghamton Grid Computing</q><start>0</start><ma
xResults>10</maxResults><filter>true</filter><restrict></re
strict><safeSearch>false</safeSearch><lr></lr><ie>latin1</i
e><oe>latin1</oe></api:doGoogleSearch></SOAP-ENV:Body></SOA
P-ENV:Envelope>
```

7.4.2 Testing Environment

We used the same testing environment as was used by the test conducted for the XML-RPC framework. The server application was run under the GlassFish Java Enterprise Server. The results were measured exactly the same way as the results for the XML-RPC framework to enable an easy comparison of the performance of these two frameworks.

7.4.3 Results

The performance *dependency* on the *parameter type* in case of a small request and a small response is again very *low*. The difference between the slowest and fastest type of parameter is only 0.32%, which is well under the accuracy of measuring tools. The results presented are not real life values, as in the measured values a *significant delay* caused by unknown reasons presented itself with periodical regularity. As the delay is probably caused by transfer or Java Virtual Machine and does not have anything to do with the speed of processing messages, we have eliminated affected values from the chart and further comparison.

Parameter type	Integer	String	Double	Boolean	Array String	Array Double
Time for 1 call (ms)	15	16	16	15	203	203
Time for 1 call (ms)	16	16	15	16	172	234
Time for 1 call (ms)	15	16	16	16	156	203
Time for 1 call (ms)	16	15	16	16	156	188
Time for 1 call (ms)	16	16	16	15	172	187
Time for 1 call (ms)	16	15	15	16	172	188
Time for 1 call (ms)	15	15	16	16	172	187
Time for 1 call (ms)	16	16	15	16	187	219
Time for 1 call (ms)	16	16	16	15	188	203
Time for 1 call (ms)	15	15	16	16	187	172
Time for 1 call (ms)	16	16	15	16	204	188
Time for 1 call (ms)	15	15	16	15	187	187
Time for 1 call (ms)	16	16	16	15	203	188
Time for 1 call (ms)	16	16	16	16	172	187
Time for 1 call (ms)	16	15	16	16	219	188
Time for 1 call (ms)	15	16	16	15	172	203
Time for 1 call (ms)	16	16	15	16	203	187
Time for 1 call (ms)	15	15	16	16	187	218
Time for 1 call (ms)	16	16	16	16	172	219
Time for 1 call (ms)	16	16	15	16	156	187
Average for 1 call (ms)	15.65	15.65	15.7	15.7	182	196.8

Table 3: The JAX-WS performance measures – parameter dependency

The time dependency tests were done in the exact same fashion as the tests of XML-RPC. The time dependency on the number of parameters shows the same behavior as was seen in the XML-RPC tests. The time seems to grow linear to the number of parameters once the size of the array reaches one thousand parameters. Before that, the time seems to grow logarithmic to the number of parameters. The cause is again the same; the constant data overhead caused by the usage of the XML-based protocol, in this case SOAP.

The difference between a string and a double parameter type is very small and does not depend on the number of parameters. The actual difference is less than 5%. The results show that the performance does not depend on the type of the parameter, but only on the number of parameters. This seems to be an improvement in comparison to tests done by [8].

The dependencies are shown on the two graphs below. The table containing the exact values is located below the graphs.

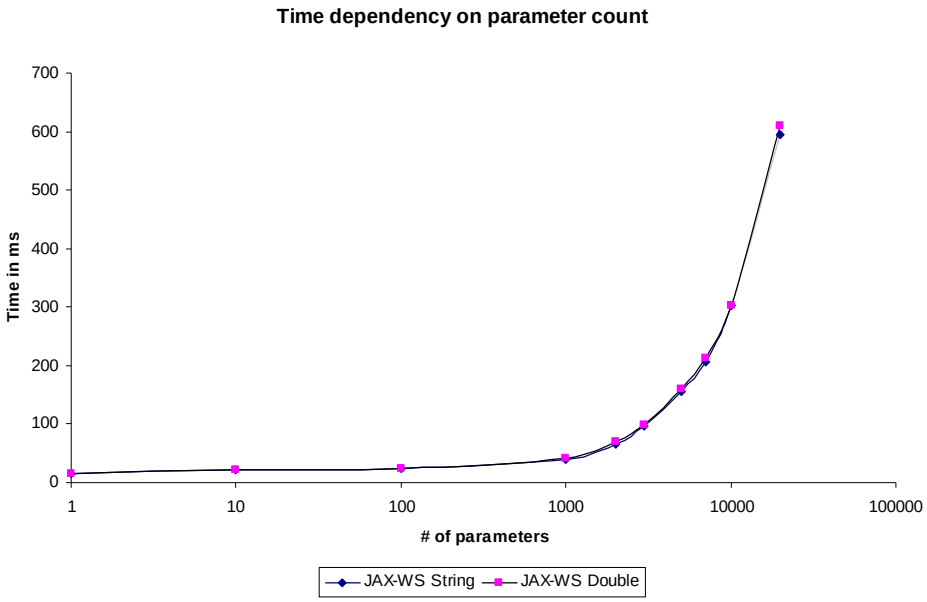


Figure 10: The time dependency on the parameter count in logarithmic scale

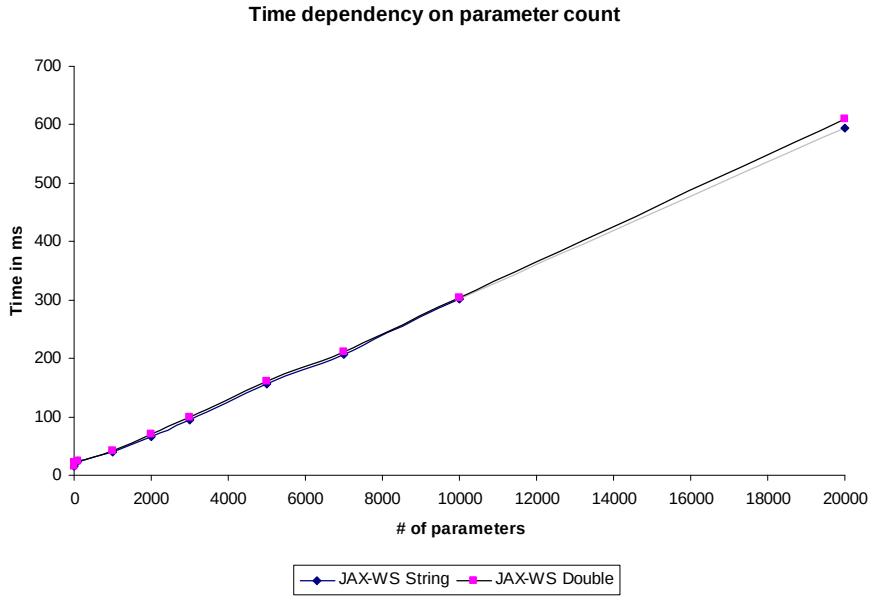


Figure 11: The time dependency on the parameter count in linear scale

	Values displayed are an average time in ms				
Number of parameters	1	10	100	1000	2000
Array of strings	15.68	22.97	25.17	40.08	66.73
Array of doubles	15.65	22.40	24.26	41.28	69.44
Number of parameters	3000	5000	7000	10000	20000
Array of strings	95.47	155.48	206.94	302.10	594.74
Array of doubles	98.11	160.26	212.31	303.04	609.71

Table 4: The JAX-WS performance measures – size dependency

7.4.4 Comparison to XML-RPC

The performance *differences* between JAX-WS and XML-RPC are *large*. The time elapsed by JAX-WS varies between 139% and 294% of the time elapsed by XML-RPC. The maximal difference was measured between the times elapsed for the message containing a single *double* value. The average time of XML-RPC is only 5.33 ms and the average time of JAX-WS is 15.68 ms, which is *194% more*. The minimal difference was measured for the message containing an array of one hundred doubles. The average time of XML-RPC is 17.50 ms and the average time of JAX-WS is 24.26 ms, which is 39% more. The time elapsed by JAX-WS stabilizes at around 237% of the time elapsed by XML-RPC with messages containing a large number of parameters.

This shows that the XML-RPC protocol is lot faster when it comes to handling any number of parameters. The cause of this lies in the large overhead that is introduced by SOAP.

Number of parameters	1	10	100	1000	2000
XML-RPC Array of strings (ms)	5.40	11.24	14.97	19.18	23.67
XML-RPC Array of doubles (ms)	5.33	9.81	17.50	26.97	33.29
JAX-WS Array of strings (ms)	15.68	22.97	25.17	40.08	66.73
JAX-WS Array of doubles (ms)	15.65	22.40	24.26	41.28	69.44
Ratio of strings	2.90	2.04	1.68	2.09	2.82
Ratio of doubles	2.94	2.28	1.39	1.53	2.09
Average ratio	2.92	2.16	1.52	1.76	2.39
Number of parameters	3000	5000	7000	10000	20000
XML-RPC Array of strings (ms)	33.95	56.08	77.65	108.44	214.54
XML-RPC Array of doubles (ms)	46.26	75.26	104.08	146.03	290.49
JAX-WS Array of strings (ms)	95.47	155.48	206.94	302.10	594.74
JAX-WS Array of doubles (ms)	98.11	160.26	212.31	303.04	609.71
Ratio of strings	2.81	2.77	2.67	2.79	2.77
Ratio of doubles	2.12	2.13	2.04	2.08	2.01
Average ratio	2.41	2.40	2.31	2.38	2.38

Table 5: The comparison of JAX-WS and XML-RPC

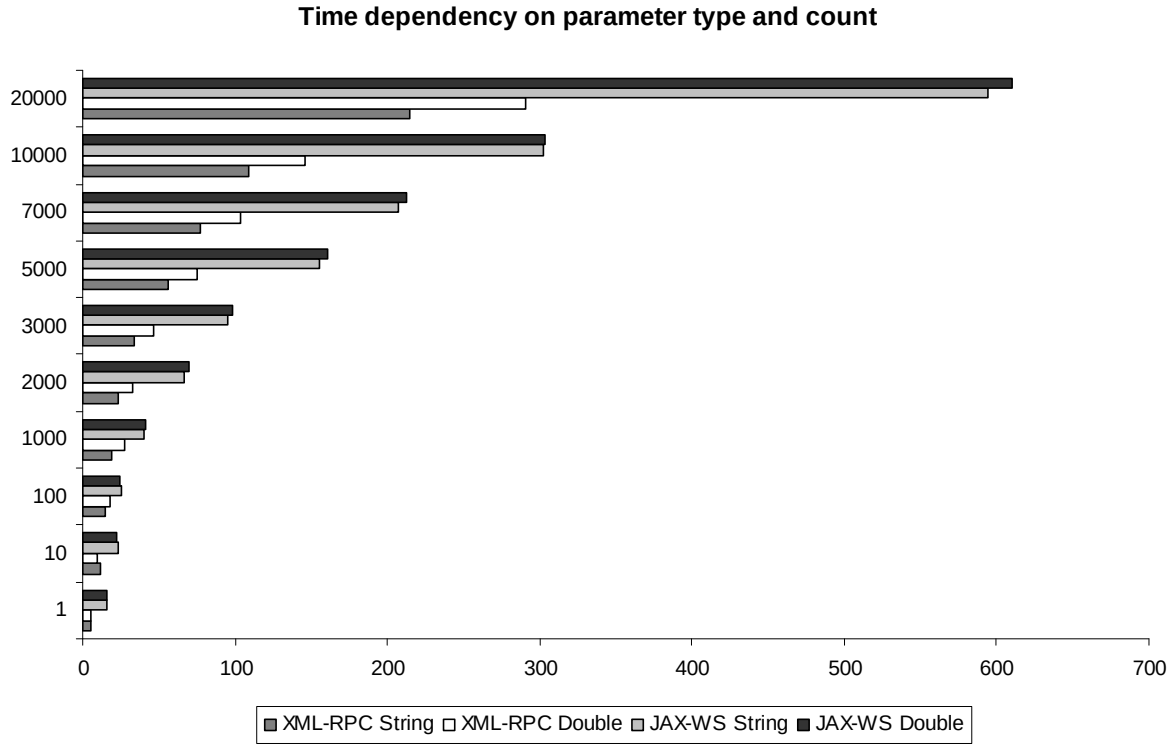


Figure 12: The time dependency on the parameter type and count

Chapter 8 CONCLUSION

The aim of this work was to use the *SOAP-based Web Services* as a tool or environment for computing the distributed algorithms. The XML-based SOAP protocol has been designed for the usage in business applications, with the interoperability being its highest priority. Only if we keep this in mind, it is possible to find a suitable usage for Web Services in the distributed algorithm computing.

The theoretical part of this thesis went through the history of *middleware* up to the newest technologies – Web Services. Then it described the protocols that are behind Web Services. We have shown the differences between the SOAP-based Web Services and the distributed algorithms as described by the academic world. We tried to find solutions to the problems implied by the differences of those two environments.

In the practical part we have implemented two simple distributed algorithms. The purpose of these implementations was to show the solutions necessary in order to make Web Services and similar technologies work as distributed algorithms. The first implementation was created using *SOAP* as the main XML-based technology used for remote procedure calls. The second implementation was done in *XML-RPC*, which is one of the first XML-based protocols for remote procedure calls. We have also created tests in order to measure the performance of the used technologies.

Web Services, even in their newest generation, proved to be *somewhat slower* than the older technologies, like XML-RPC. This is caused mainly by the *overhead* that the *interoperability* of the SOAP protocol brings. The only way that a distributed algorithm implemented with the usage of a Web Service can be comparable performance-wise to a distributed algorithm implemented using some private communication protocol is that it abuses the interoperability. Also it is quite necessary that the chosen distributed algorithm does not require a large number of messages, or large data to be send. Basically, the Web Service based distributed algorithm can cover a *large number of nodes*, but it is suitable only

for algorithms that are *heavy on the computing*, but *light on the communication*. Unfortunately, not too many algorithms fit this description.

The Web Services are suitable for *massively distributed computing* (as project SETI@Home or others). The Web Service would serve to distribute the data (small amount and not too often) and the clients would do the processor heavy computing. The development of the Web Service and clients for different platforms would be easy, as suitable integrated developer environments (IDEs) are available. The clients and services would be able to cooperate thanks to the interoperability granted by the usage of SOAP. This scenario does not correspond to the model of classic distributed algorithms as presented by [24] and was not the aim of this work.

Appendix A CONTENTS OF THE CD

The included CD contains the implementations of both algorithms. Both algorithms are included as the compiled *.jar* or *.war* file and as the NetBeans projects.

A.1 Leader Election Algorithm

The leader election algorithm is simply started by typing:

```
java –jar LeaderElection.jar option
```

in the *Distributions/LeaderElection/* directory. The `option` defines the subnet mask of the network, where the application is to be started. If no `option` is provided, the default (255.255.255.255) broadcasting address is used. The UDP broadcast server uses the port number 7026 and the web server runs at port 8080.

The NetBeans projects is stored at *Projects/LeaderElection/*. It was created in the version 6.0 Beta 1.

A.2 Resource Reservation Algorithm

The resource reservation has to be deployed to a running Java Enterprise server. The *.war* file is located in the *Distributions/ResourceReservation/* directory. After the application is deployed, the algorithm is started by calling the `Activate` method of the Web Service. The `Activate` method can be called by running the *Starter.jar* from the *Distributions/ResourceReservation/* directory:

```
java –jar Starter.jar option1 option2
```

The `option1` defines the port number of the web service at which the *.war* file is deployed. The default port number is 8080, this is used as default by the embedded web server in the NetBeans IDE. If you are using a different web server, please specify its port number. The port number has to be specified. The `option2` defines the subnet mask of the network, where the application is to be started. If no `option2` is provided, the default

(255.255.255.255) broadcasting address is used. The UDP broadcast server uses the port number 7026 and the Web Service runs within the web server.

Easier way to run this algorithm is to open the NetBeans project in the NetBeans IDE with the embedded Java Enterprise Server. After the project is opened in the IDE, the application has to be deployed. Then the easy way to start the computing is by invoking the Activate method in the Web Service test page. The NetBeans project is located in the Projects/ResourceReservation directory. It was created in the version 6.0 Beta 1.

BIBLIOGRAPHY

[1] Java Remote Method Invocation Specification. Technical report, Sun Microsystems, Inc., 2004.

[2] N. Abu-Ghazaleh and M. J. Lewis. Differential deserialization for optimized SOAP performance. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 21. IEEE Computer Society, November 2005.

[3] B. Aiken, J. Strassner, C. B, I. Foster, C. Lynch, J. Mambretti, R. Moore, and B. Teitelbaum. Network policy and services: A report of a workshop on middleware. http://www.ietf.org/rfc/rfc2768.txt, February 2000.

[4] M. Allman. An evaluation of XML-RPC. In *ACM SIGMETRICS Performance Evaluation Review*, volume 30, pages 2–11. ACM, March 2003.

[5] T. F. Apache. Web Services - AXIS. http://ws.apache.org/axis/.

[6] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0. http://www.ietf.org/rfc/rfc1945.txt, May 1996.

[7] A. Chaudhury and H. R. Rao. Introducing client/server technologies in information systems curricula. In *ACM SIGMIS Database*, volume Volume 28, pages 20–32. ACM, September 1997.

[8] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 246. IEEE Computer Society, 2002.

[9] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, March 2001.

[10] L. Clement, A. Hately, C. v. Riegen, and T. Rogers. UDDI spec technical committee draft, dated 20041019. http://uddi.org/pubs/uddi_v3.htm, 10 2004.

[11] W. Emmerich. Distributed software architecture using middleware. Advanced Software Engineering Course, 2002.

[12] R. v. Engelen and K. Gallivan. The gSOAP toolkit for Web Services and peer-to-peer computing networks. In *2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002)*, pages 128–135, Berlin, Germany, May 2002.

[13] A. Harrison and T. Ian. Service-oriented middleware for hybrid environments. In *ACM International Conference Proceeding Series*, volume 181, November 2006.

[14] M. R. Head, M. Govindaraju, R. v. Engelen, and W. Zhang. Benchmarking XML processors for applications in grid Web Services. In *Conference on High Performance Networking and Computing*, number 121, November 2006.

[15] M. Henning. The rise and fall of CORBA. *Queue*, 4(5):28–34, June 2006.

[16] M. Horstmann and M. Kirtland. DCOM architecture. Technical report, Microsoft Developer Network, July 1997.

[17] V. Issarny, M. Caporuscio, and N. Georgantas. A perspective on the future of middleware-based software engineering. In *Proc. Future of Software Engineering FOSE '07*, pages 244–258, 23–25 May 2007.

[18] C. P. Java. Glassfish project. https://glassfish.dev.java.net/.

[19] C. P. Java. *Java API for XML-based RPC JAX-RPC 1.1*. Sun Microsystems, 2003.

[20] M. B. Juric, B. Kezmah, M. Hericko, I. Rozman, and I. Vezocni. Java RMI, RMI tunneling and Web Services: Comparison and performance analysis. In *ACM SIGPLAN Notices*, volume 39. ACM, May 2004.

[21] D. Kohlert and A. Gupta. *The Java API for XML-Based Web Services (JAX-WS) 2.1*. Sun Microsystems, May 2007.

[22] R. M. Lerner. Introducing SOAP. *Linux Journal*, 83:Article No. 11, March 2001.

[23] S. M. Lewandowski. Frameworks for component-based client/server computing. In *ACM Computing Surveys (CSUR)*, volume 30, pages 3–27, March 1998.

[24] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996.

[25] Microsoft. DCOM technical overview. Technical report, Microsoft Developer Network, November 1996.

[26] OMG. The Object Management Group (OMG). http://www.omg.org/.

[27] K. A. Phan, Z. Tari, and P. Bertok. Mobile computing and applications (MCA): A benchmark on SOAP's transport protocols performance for mobile applications. In *Proceedings of the 2006 ACM symposium on Applied computing SAC '06*. ACM Press, April 2006.

[28] J. D. Quintana. Getting to know Mono. *Linux Journal*, 111:4, July 2003.

[29] R. J. Ray and P. Kulchenko. *Programming Web Services with Perl*. O'Reilly, December 2002.

[30] G. Tel. *Introduction to Distributed Algorithms, Second Edition*. Cambridge University Press, February 2000.

[31] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2):46–55, 1997.

[32] W3C. SOAP version 1.2 part 0: Primer (second edition). http://www.w3.org/TR/2007/REC-soap12-part0-20070427/, April 2007.

[33] Wikipedia. Distributed Component Object Model. http://en.wikipedia.org/wiki/Distributed_Component_Object_Model.

[34] Wikipedia. Web Service. http://en.wikipedia.org/wiki/Web_service.

[35] D. Winer. XML-RPC specification. http://www.xmlrpc.com/spec, June 2003.