Modeling Grid Applications

Christian Pöcher

Diploma Thesis
Aachen, May, 8, 2007

Prepared at the Chair of Distributed Systems
University of Tartu
Presented at the RWTH Aachen University

First examiner: Prof. Dr. Eero Vainikko
Second examiner: Prof. Dr. Felix Wolf
Supervisor: Dipl.-Inform. Ulrich Norbisrath

One day I realized that sadness is just another word for not enough coffee.
Dilbert

Abstract

At the moment GRIDs shift from batch processing to service oriented architectures. With the use of services, GRID application composition becomes possible. It is necessary to preserve existing legacy software and make it available to service oriented GRIDs even if the architecture is different. In this thesis a case study is done that shows how legacy applications can be refactored to fulfill the non-functional requirements of a component. The application is then wrapped as a Web Service. It is shown that the application can be reused by another component to form a higher order service. Then a way is shown how to split off components from a legacy application to increase flexibility and reuse.

Acknoledgement

This work was possible only because of the support of many people. Especially, I would like to thank:

- Prof. Dr.. Eero Vainikko for offering me the chance to do a very interesting thesis in a very interesting country, for making sure all my needs were taken care of and for the organization of our weekly ”workshops”,
- Dipl. Inform. Ulrich Norbisrath for inviting me to join him to Tartu and for all the constructive criticism, that improved this thesis so much,
- Prof. Dr. Felix Wolf for his lecture on GRID computing and for co-evaluating my thesis,
- BSc Oleg Batrashev for his invaluable testing framework, the cookies and the great discussions we had,
- Daniel Arruda, Manuel Anaya and Björn Kuhlmann, my international team of proof readers and
- my family, who always supported me, even though they are in a difficult situation themselves.

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, 8.Mai 2007 Christian Pöcher

Contents

Chapter 1

Introduction

1.1 Motivation

The history of semiconductors is characterized by an exponential increase of performance metrics commonly called *Moore's Law*. While Moore's Law held for four decades, Moore himself states that the law might soon be no longer valid because a fundamental limit is reached [Dub05]. Even now, the energy consumption limits the increase computational power of single core CPUs which is why multi core CPUs are becoming available on the market. When increases in CPU performance will get smaller, distributed computing gets more and more important because applications like climate and weather simulations [SKM$^+$02] or the CERN Large Hydron Collider [LCG05] but also all other kinds of applications still demand more and more power. The GRID offers computational power and storage on demand and can be viewed as a huge virtual computer. Still, problems in GRID computing are very similar to most distributed computing environments: Latency between processors can make parallelization impossible, typical GRID applications are hard to develop and even harder to debug and parallel algorithms are often much harder to understand than their serial counterpart.

At the moment GRIDs shift from batch processing to service oriented architectures. While service-oriented GRID frameworks are already available and some production GRIDs are doing the migration, most GRID users are still thinking and talking in terms like libraries, scripts and executables instead of services and components. Not only the terms used by typical users but also software engineering support is lagging behind. Most modeling tools focus on object oriented methods but GRID users often use Fortran or C for their applications.

This thesis will focus on DOUG a typical old-style GRID application with very large computational and memory demands. Using DOUG as an example, this thesis will explore how legacy applications can be made available as a Web

Service and how they can be refactored to benefit even more from a service oriented architecture.

In detail, the thesis will show how a wrapper around DOUG can be used to form a Web Service. To show the possibility of application composition on the GRID, an Eigenvalue solver is built that uses the DOUG Web Service as component while it is exposed itself as a Web Service. Different approaches will be examined for the construction of the exchanged messages. As a result the formats of the data files of DOUG will be changed so that interoperability as well as efficiency is achieved. Structured refactoring of the Fortran code of DOUG will be examined and a candidate for a refactoring pattern will be presented. Finally an architecture for componentization of DOUG application will be described and its properties are explained.

1.2 Overview

In the next section some important terms are defined which are either ambiguous or might not be known to the reader. Chapter 2 explains the most important techniques used in this thesis. After the base of the thesis is covered, the current research is described in chapter 3. There it is explained how modern GRIDs are built, what research has been done in the field of componentizing legacy applications, what DOUG is and which challenges the developers of it face and at last a current overview of refactoring in Fortran.

There are two distinct programming examples described in this thesis. The first is the creation of an Eigenvalue solver using DOUG as a whole as a component. The solver itself is also exposed as a component and easy to run in a service oriented architecture. Design and implementation issues are illustrated in chapter 4. A special focus was put on refactoring and the data exchange formats.

The next chapter is about splitting off the preconditioner as component, an integral part of DOUG, so that it can be used externally by other applications or exchanged at will easily or even dynamically. Although it has not been implemented for doubts regarding performance, the findings in the architectural design phase are given and hints for future work in componentization of legacy applications are pointed out.

Work of researchers which is related to this thesis is outlined in chapter 6. In chapter 7 the findings are summarized and recommendations for future work based on the findings are given.

1.3 Definition of terms

In this section definitions of several important terms a given, either because some readers might be unfamiliar with them or because there is no clear consensus on the definition. In the latter case other definitions are discussed, after which the definition that serves as the base for this thesis is given.

The section is structured into different parts, therefore readers with a strong background in a field can safely skip the corresponding part.

1.3.1 Component

Different definitions exist for the term component. There is general agreement that a component is a unit of software that has some defined purpose and capability is independent and can be composed with other components so that they form a component based software system.

Councill and Heinemann [CH01] were the first to define a component. Their definition is as follows:

> *A software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.*

Their definition focuses on the standards involved in component development, deployment and usage. It also emphasizes that a component's behavior should only be influenced by the input data and the configuration not by modifications to the code. It describes the value of a component for an enterprise.

Szyperski [SGM02] focuses more on a description of the characteristics:

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

He also mentions that standards are involved but he limits his observation to the contracts that make up the interfaces. All dependencies should be explicit that is either through the interfaces or through documentation.

In order to be practical, the author combines some aspects of the former definitions. For this thesis, the term is defined as follows: A *software component*

- is an *independent unit.* It can be compiled and used without other non-declared dependency. Usage of required sub-components is declared therefore components can be composed from existing entities.

- uses *standardized communication, lifecycle and deployment* facilities. It can but does not have to use a *middleware* (see below) that implements these.
- has *defined provides and requires interfaces* and — if the interface alone is not sufficient — a document describing the contracts of using those interfaces. Other dependencies should be avoided but have to be documented if they exist.
- must exist as a *deployable package.*
- must have at least a *documentation of the syntax* of the component's interfaces so that users can utilize the component in their software.
- can be exchanged if used in a component-based software development process and the contracts that is interfaces and semantics match.

Although the author sees a fine distinction between the terms service (see chapter 1.3.5) and software component, they are very similar. The main difference is that components are implementations of services. This also reflects in the similarities of the definitions of components and Web Services, which share similar properties. Web Services will be explained in chapter 2.1.

1.3.2 Middleware

A *middleware* is a runtime environment that offers functionality that should not put directly into the application. It separates logic code (a.k.a. business logic) from physical code (e.g. protocol classes) and is mainly used in distributed system environments. If the access to the physical layer is only done through the Application Programmer's Interface (API) of the middleware, the application does not depend on the physical layer e.g. the operating systems network protocol implementations. That application can then be run on any host that can run the middleware and is decoupled from underlying physical properties of the host. In general, middleware replaces non-distributed functions of the operating system with distributed functions (e.g. remote procedure calls, distributed databases and also CPU sharing in GRIDs) thus making the application network-aware.

A look onto figure 1.1 explains why this layer is called middleware: It resides "in the middle" between the logic layer with its business applications that implement the solution to a domain related problem and the physical layer that implements the infrastructure like network protocols. The integration of applications becomes possible since the applications do not rely on different communication protocols of different platforms but on standardized communication of the middleware [Ber96]. In figure 1.1, a communication between two business applications aided by middleware is shown. The initiating application calls the API of the middleware which

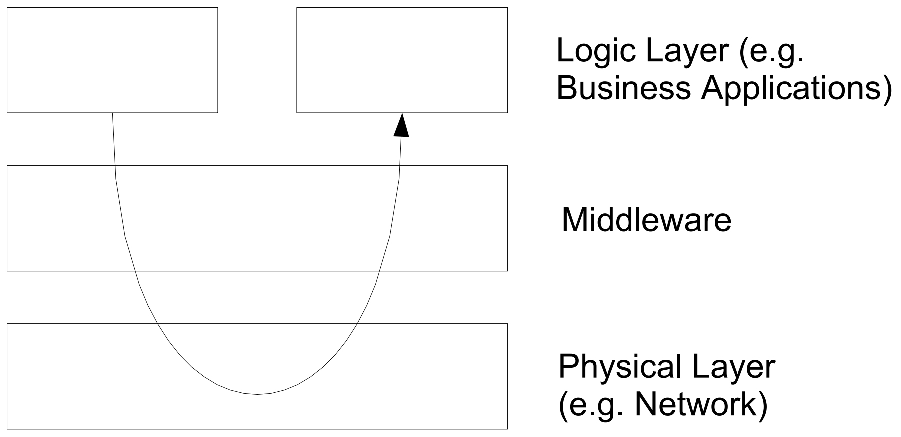


Figure 1.1: Communication aided by middleware.

handles interoperability issues and does the necessary communication through the physical layer. The middleware — either on a different host or on the same — processes the message from the network and presents the result to the receiving application.

1.3.3 Module

Although the term *module* will not be used in this thesis, it will deepen the understanding of the concept of a component by focusing on the differences between those two. Modules have been around in computer science for a few decades now and have been made integral parts of several older languages like Pascal, Ada and Fortran 90.

A module is a structural unit of software which can (ideally) be compiled independently and which has defined interfaces. It cannot be as easily exchanged as components. Usually the same platform and compiler must be used and the actual module exchange cannot be done while running the application (*"hot swap"*). A component framework can support hot swap which will be important for upgrading GRID services.

Sometimes the term module is also used as a synonym for *plug-in*. Plug-ins are usually made in a way that they implement certain interfaces of a host application. Therefore they are only useful with that specific application. While they can be exchanged — sometimes even in hot swap manner — those plug-ins cannot be reused in a different application. In this sense plug-ins are less sophisticated than components.

While of course modules are used in the DOUG application source to structure the code, they are not sufficient for the goal of the thesis because of the lack of independence and reuse. Also plug-in modules are not adequate. Therefore components defined as above had to be used.

1.3.4 Service Oriented Architecture

A *Service Oriented Architecture (SOA)* implements an architectural model not only for IT but also for the way a business is structured. The concept of SOA was developed by extending the scope of software structures but nowadays also includes other stake holders. It can be considered as a blueprint for enterprises how to structure IT-related business to benefit from increased flexibility.

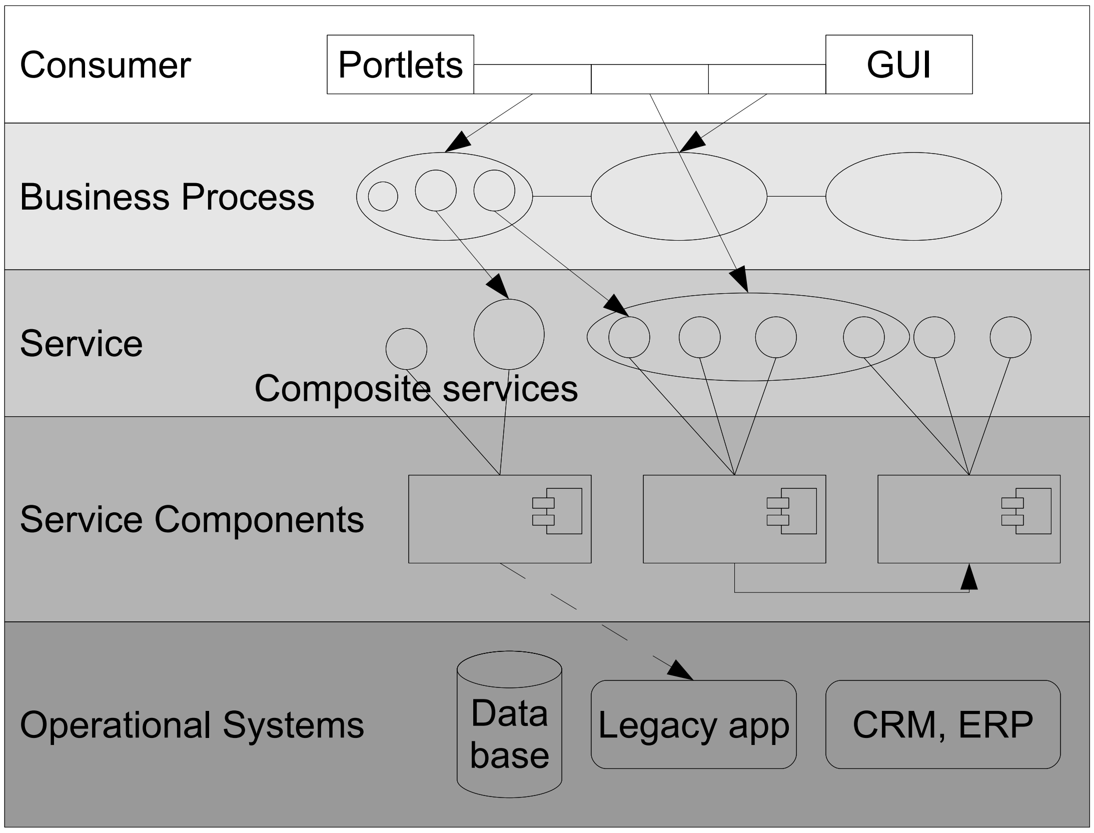


Figure 1.2: Five layers of an service oriented architecture [Ars04]

For IBM, Arsanjani [Ars04] has defined an abstract view on SOA by identifying seven different layers to an SOA. The five main layers[1] depicted in figure 1.2 are:

[1]Arsanjani keeps expanding and refining his architecture model but the five layers presented

- *Consumer Layer:* Any consumer of a service is in this layer.
- *Business Process Layer:* A business process can use just one service or orchestrate several services e.g. with BPEL.
- *Service Layer:* The services and their documentation are in this layer.
- *Service Components:* The actual implementation of a service which can use a multitude of technologies e.g. EJB, .NET or Web Services.
- *Operational Systems and Data:* Legacy systems and data storages are on this level.

This abstract view will serve as a base for this thesis. Now we have the tools to finally define the term service.

1.3.5 Service

There is no consensus of a definition of the term *service*. When people are talking about services, they often have *Web Services* (see below) in mind ignoring the fact that a service could be much more. The authors of [PL03] have a very broad definition of a service that includes consumer, technical, provider and business perspectives which all define a service. Therefore they are defining a service in a broader meaning than just the layers that IBM referred to in their architecture model. This does not help the work for this thesis because it is still very vague about what to do to create a service. In other words, their definition is not operationally enough.

Therefore the term service will be defined with the help of IBM's SOA architecture model. By referring to a service in this thesis an entity of the third layer is meant. A service should have a certain set of functionalities and dependencies that are exposed by provides and required interfaces. The interfaces must be documented in a way that makes them available for consumers. While it may ok to have a provides interface with the operation `doTheDance(int i)`, it is only valid for a service if documentation explains in detail what to expect from this operation. The service definition fixes the contract of the interfaces.

On the other hand, implementation is not scope of a service anymore. With the fixed contract, the implementation as well as the technologies used for implementation can change. This would reside on the forth layer of the architecture model.

To summarize this into one sentence: *Services are defined separately and implemented by service components.*

here are the core of the model. More information is published regularly on Arsanjani's blog.

1.3.6 Discrimination

It is important to emphasize the relation of the terms defined in this chapter. They are distinct only by subtle differences. Generally spoken, they form a hierarchy in terms of abstractness (see figure 1.3): A SOA is purely abstract, a service is less abstract and a service component is even more concrete.

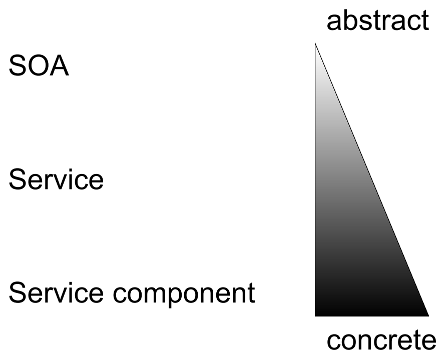


Figure 1.3: The main terms introduced here are abstract to a different degree.

Why is that so? A SOA is an architectural model defining how the system is structured. A service is one entity within an SOA which already has a clear function within the system and has defined interfaces. A service component is a component that implements at least part if not all of the service's logic.

The distinction between a component and a service component is that the latter is used by a service. There are component frameworks that are not built to form a SOA, for example ActiveX or OLE [Cha96].

1.4 Solution Sketch

Taking the definition of a component as a list of requirements, the legacy application DOUG will componentized in two steps. First, DOUG will be used to form a component which contains all of DOUG to make it available in SOAs like modern GRIDs. DOUG will be refactored internally and later extended by introducing a new data format that introduces independence of platform and compiler for the first time. The possibility of application composition will be proven by building an Eigenvalue solver that uses this component. Afterwards, it will be looked into splitting DOUG into several components to make the parts of it available as building blocks of GRID applications and to exchange single components of DOUG very fast for evaluation of new algorithms.

Chapter 2

Used Techinques

In this chapter some of the used techniques will be explained briefly. A basic understanding of those is necessary for the work described later. First Web Services are explained, a technique to implement components and services, which is the base of modern GRIDs. The capabilities of Web Services, namely the interfaces, their semantics (as comments) and the bindings to a transport are described by the Web Service Description Language which is explained next. After that an introduction to the Message Passing Interface is given which is important for chapter 5.

2.1 Web Service

Web Services (WS) are one specific technique to implement services. At first invented by IBM and Microsoft Corporation, the standards are now governed by the W3C and define a way to do application integration. Because the W3C oversees specification work in the family of WS standards, interoperability between different WS frameworks is relative good although not totally without problems.

The definition of a Web Service has changed over the passing of time. The current definition by the W3C is as following:

> A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

In figure 2.1 is an example SOAP message that occurred in the work for this thesis. A more abstract description of SOAP can be found in [KNN03]. As a funny side note, it should be noted that originally SOAP meant *Simple Object*

Access Protocol but critics claim that SOAP is neither simple nor used for object access [Vog03]. Today the term SOAP is not considered an abbreviation anymore.

The benefits of Web Services include the formulation of well defined, non-propietary standards. Compared with older techniques such as CORBA, the standards reach much farther and leave less ambiguities. The number of standards in the WS-* family is still increasing steadily on one hand showing the need to solve specific problems of Web Services but on the other hand removing insecurities in interoperability. Web Services are loosely coupled leading to more modularity and flexibility. Since Web Services are defined dynamically, they can be upgraded automatically if the interfaces do not change. Because SOAP messages are human readable debugging is fairly easy.

The most often pointed out problem of Web Services is inferior speed compared to competing middleware technologies like CORBA, because non-binary messages are used. In [PTL04a] the authors experimentally compared Web Services with (the less flexible) MPI. In the example in listing 2.1 the size of the message is 1638 byte compared to 72 byte in binary payload of the message. That is an increase of 2275%. Because messages can get big so easily, a lot care has to be taken to minimize the necessary communication between services and service consumers.

Two techniques should be mentioned here because they could be considered as successor of Web Services. First this is *Representational State Transfer*, more commonly known by it's abbreviation *REST*. It was first mentioned in [Fie00] and is supposed to be less bulky than Web Services. Often commercial enterprises offer both a Web Service and a RESTful service. Problematic with this architectural model is that it is hard to develop state-aware services.

An extension of Web Services is the second technique that shall be mentioned here: The *Web Service Resource Framework (WSRF)* which adds stateful resources to Web Services. WSRF is the base of modern GRID systems. More about WSRF will be explained in chapter 3.1.

2.2 Web Services Description Language

The *Web Services Description Language (WSDL)* [CCMW01] describes several elements of Web Services. Those are:

- *types:* Data type definitions besides those that are know to the system per default. Usually *XML Schema Definitions (XSD)* are used for that purpose.
- *messages:* The typed data that is transferred when calling operations.
- *operations:* The description of the operations that the service offers.
- *port types:* An abstract set of operations that endpoints offer.

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/xml;charset=utf-8
Date: Sat, 11 Feb 2006 11:33:16 GMT
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
   <ns1:runAssembledResponse
    soapenv:encodingStyle=
            "http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:ns1="http://doug.math.ut.ee/">
    <runAssembledReturn href="#id0"/>
   </ns1:runAssembledResponse>
   <multiRef id="id0" soapenc:root="0"
    soapenv:encodingStyle=
            "http://schemas.xmlsoap.org/soap/encoding/"
    xsi:type="ns2:DoubleVector"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:ns2="http://doug.math.ut.ee/">
    <vector soapenc:arrayType="xsd:double[9]"
     xsi:type="soapenc:Array">
     <vector xsi:type="xsd:double">-1.3248258958028</vector>
     <vector xsi:type="xsd:double">1.371192505789</vector>
     <vector xsi:type="xsd:double">-0.9431052535296</vector>
     <vector xsi:type="xsd:double">0.83135799644846</vector>
     <vector xsi:type="xsd:double">-2.7102213579422E-4</vector>
     <vector xsi:type="xsd:double">0.29152348710796</vector>
     <vector xsi:type="xsd:double">-0.17966396898322</vector>
     <vector xsi:type="xsd:double">-0.24831102223255</vector>
     <vector xsi:type="xsd:double">0.20205667328996</vector>
    </vector>
   </multiRef>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing 2.1: A typical SOAP message

- *binding:* A binding specifies which port type is offered over which network protocol.
- *port:* A concrete endpoint which is a combination of a binding and a network address at which the operations can be reached.
- *service:* A collection of endpoints that are related. This is what is usually called a service.

Optionally, it can also contain documentation. In theory, a WSDL file is a good starting point to build a Web Service because it forces the programmer to think about the contracts first. In practical work though, most developers decide to write the service first and then generate the Web Service description because it seems easier that way. Note that this violates the contract-first-principle of Web Services. Maybe over time tools are being developed, which make creating WSDL files more easily.

The WSDL files created for this thesis are too long to print them here therefore a smaller example has been given in listing 2.2. This service can be found at the (hypothetical) end point address URI `http://example.com/getTerm`. It has one port with the name `glossaryTerms` which has only one operation called `getTerm`. This operation has an input and an output message so it is a request-response operation. The messages are specified in the `<message>` tags and consist of one string each. In the `<binding>` tag you can see that HTTP transport is chosen with literal encoding for both input and output. A more comprehensive example can be found in [CCMW01].

2.3 Message Passing Interface

To understand how DOUG achieves parallelization and to evaluate the problems of the usage of the *Message Passing Interface (MPI)* on GRIDs one must have knowledge about MPI at least on a conceptual level. In this section a basic intoduction into MPI is given. For a more complete MPI tutorial have a look at [Nat].

MPI is a technique to allow coordination of a program running as multiple processes in a distributed memory environment. It is standardized [For] therefore code which uses MPI can be run on any machine that has the MPI library installed. The specification is limited to defining operations, their signature and semantic but not the protocol or the implementation. A huge variety of MPI implementations is available some of them with very special feature. An example of a popular implementation is LAM/MPI [BDV94] which is also used for the development of DOUG. Examples for special purpose MPI are MPICH-G2 which is a grid-enabled

```xml
<?xml version="1.0"?>
<definitions name="getTerm"
             targetNamespace="http://example.com/getTerm.wsdl"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

<message name="getTermRequest">
   <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
   <part name="value" type="xs:string"/>
</message>
<portType name="glossaryTerms">
  <operation name="getTerm">
      <input message="getTermRequest"/>
      <output message="getTermResponse"/>
  </operation>
</portType>
<binding type="glossaryTerms" name="b1">
<soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http" />
  <operation>
    <soap:operation
     soapAction="http://example.com/getTerm"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>

</definitions>
```

Listing 2.2: An example WSDL file

implementation of MPI or Fault Tolerant MPI (FT-MPI) which can survive the crash all but one processes of the job and restart the crashed jobs.

One can distinguish two different parallel computer architectures: *distributed memory* and *shared memory*. *Distributed memory* parallel computers work on the same problem, but each of the serial computers involved has rapid access to it's local network, while access to the memory of other computers is given through the exchange of messages over a network. *Shared memory* computers have several processors that access the same global memory using a high-speed data bus. The number of processors in a shared memory parallel computer is usually limited to 2-16 because the memory bus becomes a bottleneck. Nowadays, parallel computers often use a mixed shared/distributed memory architecture in which groups of 2-16 processors are considered one node and several nodes are interconnected by a network. MPI is used to build a software infrastructure to send messages in a distributed memory environment.

While MPI uses message passing via library calls from a vast variety of standard programming languages *directive-based data-parallel languages* use a different approach: Serial code is made parallel by adding directives as comments into the code which are then used by the compiler to decide how data and work is distributed on the processors. The details of this process is left to the compiler. Popular examples are High Performance Fortran (HPF) and OpenMP. They are usually used in shared memory architectures because the unified memory space makes it easier for compiler developers to improve performance.

MPI assumes a *standard model of parallel computation*. In this model a parallel computation consists of a number of processes that work some local data each. The processes have only local variables. There is no mechanism to access the other processes' variables directly. Instead, data sharing is done by exchanging explicit messages between processes. A process does not necessary have to run on exactly one processor. It is possible to start several processes that share one processor.

As mentioned before, MPI uses function/subroutine calls from a library to achieve its functionality. The calls can be classified in four groups:

- Calls to initialize, manage and terminate communications.
- Calls for point-to-point communication.
- Calls to perform communications involving a group of processors.
- Calls to create new data types.

Here only the second and third group are explained. For deeper knowledge seek the literature, for example [Nat].

Point-to-point communication refers to one process that sends a message and one that receives this message. Both sides must declare that they want to send/recieve with a MPI call: Not only an explicit send is required, but also an explicit receive. This is sometimes also called *two-sided point-to-point communication.*

A send or receive consists of an envelope in which sender and receiver are specified and the body which contains the payload data which is actually transferred. In each MPI send or recieve the body has to be specified by three parameters:

- The *buffer* which is the starting location of the memory in which the outgoing data can be found or the incoming data can be stored. This is for example an array variable in Fortran.
- The *data type* which is to be sent. This can be a primitive type that is already built into MPI or a arbitrary user-defined type.
- The *count* which is the number of items of that type to be sent. Note that the size of one item in bytes is determined by MPI through the specification of the data type.

A send can be either *blocking* or *non-blocking.* With a few exceptions, there are blocking and non-blocking versions of every MPI send or receive call. A blocking send or receive only returns from the subroutine call when the operation is finished. The programmer can rely that all data is accessible and can be used or overwritten. The downside is that the computation is stopped while the communication is taking place. Non-blocking sends or receives return directly after the subroutine call. The communication is done in the background. The programmer can test later if the communication is finished or issue a wait statement which forces computation to stop until the data is completely sent or received. During the communication the programmer cannot assume that the buffer variables can be used or overwritten. Non-blocking communication interleaves communication and computation and helps therefore to speed up programs. The downside is that those programs are harder to write and maintain.

More complex communications, which involve a number of processes, are also standardized and called *collective communications.* These routines could be composed by point-to-point send and receive statements by the application programmer but this is inferior because the code needed for that is often complex and thus error prone. It would make the code less readable. Also several non-obvious performance enhancements can be made which need expert know-how. In [CHPvdG04] it is claimed that even in the most popular MPI implementations the optimum is not yet reached. While collective communications define one-to-all, all-to-one or all-to-all schemes, the scope of the operations can be narrowed down by using *communicators* which build sub-groups thus achieving e.g. one-to-all-in-group communication. It would exceed the scope of this introduction to go into those in detail.

A *broadcast* is the simplest kind of collective communication. In this one-to-all operation the sender sends a copy of some data to all other processes. The operation is illustrated in figure 2.1. As an example, let there be four processes P1 to P4. At one distinctive memory position — the buffer — they have all different data stored. This is shown by the characters A, B, C, and D. The initiator of the broadcast sends a copy of its data, that is A, to all other processes which overwrites the data in the memory. After the operation all processes have the data A in the buffer.

Gather is a all-to-one operation that fetches data from all processes and arranges them in the buffer of one process. Assume the data is distributed in processes P1 to P4 and the send buffer contains is A, B, C and D as in 2.2. One process, here P1, gets all the data, concatenates and stores it in the receive buffer. The other processes' memory is not altered.

The operation *AllGather* (see figure 2.3) is a variant of Gather, in which the gathered data is transferred to every node instead of just the root node. The result is the same as in the last example but the data sequence ABCD is now in all processes P1 to P4.

Scatter operations are exactly the opposite of Gathers. They distribute some data equally to all processes. Assume P1 has ABCD in memory then P1 copies A into its receive buffer, P2 gets data B, P3 gets data C and P4 gets data D. Note that the send buffer of P1 containing ABCD is not overwritten.

The last class of collective communication are the *reduction* operations. A reduction performs a basic calculation (e.g. sum, minimum or logical operations) on data distributed in the send buffers of several processes and stores the result into receive buffer of the root process. A diagram is shown in figure 2.4. The send buffers contain the integer numbers 1, 3, 0 and 3 in the send buffers on the processes P1 to P4. In this example, they are gathered and summed up. The result of 7 is stored in the receive buffer of P1.

2.4 UML component diagram

The *Unified Modeling Language (UML)* is a graphical specification language used to help software architects and developers to model object-oriented systems in an abstract way. Therefore, they can focus on the design and hide the details until later. Because of the graphical notation, UML diagrams are more intuitive for most humans than text. By defining a standard set of diagrams, it is assured that the diagrams can be used to communicate with other specialists. The first version of UML was released in 1997 and was followed by the current UML 2.0 in 2004 and

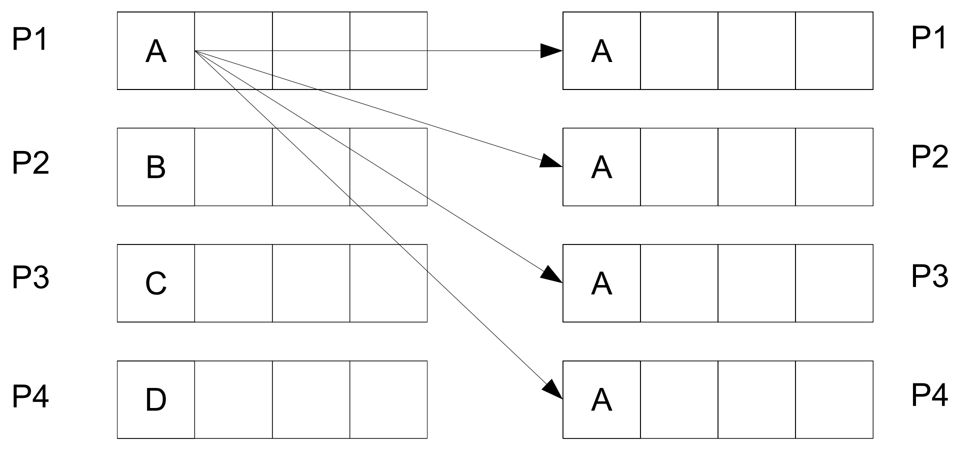


Figure 2.1: Communication schema for `MPI_Broadcast`

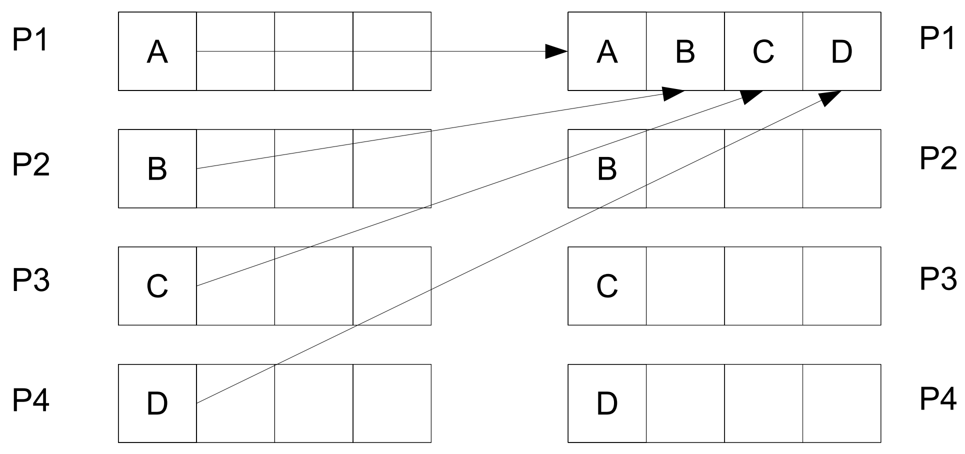


Figure 2.2: Communication schema for `MPI_Gather`

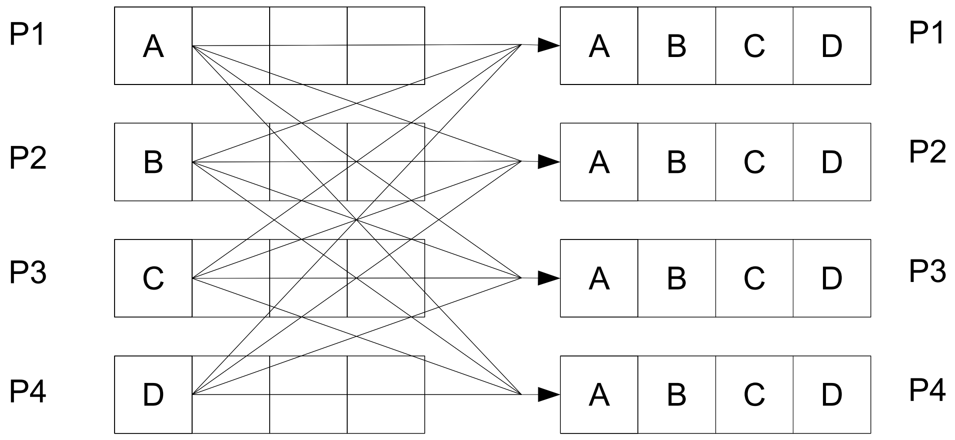


Figure 2.3: Communication schema for `MPI_AllGather`

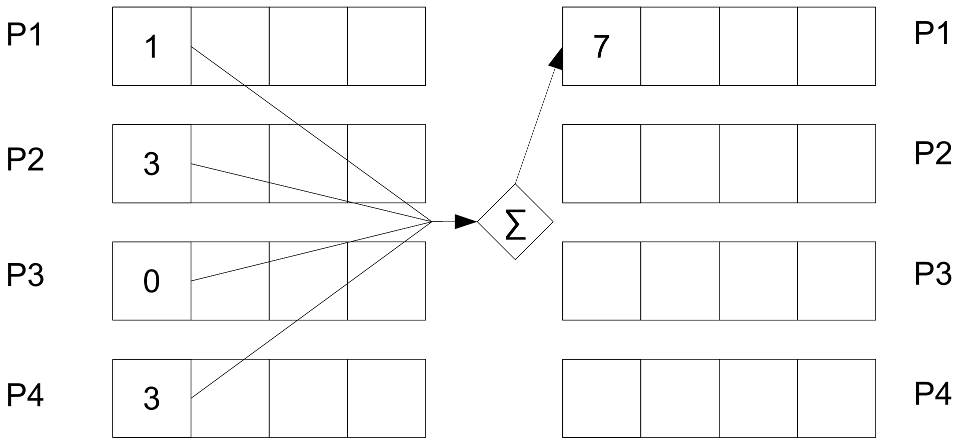


Figure 2.4: Communication schema for `MPI_Reduce`

2005. The models of software systems are expressed in diagrams of which UML defines 13 types. In this thesis only component diagrams have been used which describe the structure of the component-based system.

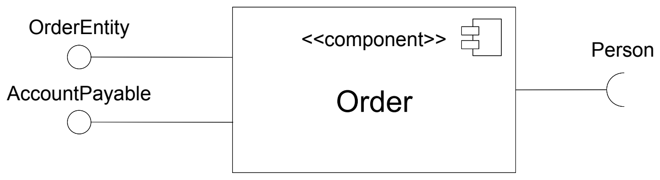


Figure 2.5: A generic component diagram. (taken from [Bel04])

Take a look at the simple component diagram figure 2.5. It shows only one component with three interfaces. Note the elements: The big rectangle is the components itself. A textual *stereotype* is added in `<<brackets>>` and shows that this rectangle is a component. A visual stereotype could also be used but usually only one type is given. *Dependencies* are shown by the lines connecting the components. A circle is an *interface* and can be also named which is not done here because there are not so many interfaces as that it would be unclear which one is meant. If a line connects directly to a circle, then the interface is a *provides interface* for the connected component. That means that the component provides the functionality described by the interface. If a line connects to a circle indirectly by using a half circle (The author uses the picture of grabbing hands as mnemonic hook.), then the interface is a *requires interface* for the connected component which means that the component requires the functionality described by the interface. Without a matching provides interface the component cannot work. It should be noted that some people prefer to draw arrows with a dashed line instead where the arrow can be read as "depends on". It is also possible to layer components that is to put a component diagram inside the one component rectangle to show a components inner structure. In figure 4.1 you see a complete UML 2 component diagram that is used for this thesis.

Component diagrams are usually used early in the development of a new software product but they can also be used for reengineering an existing software system to clarify or introduce a component structure. The terminology of component diagrams already suggests on which features one should focus when making a component model. The scope of the component, its interfaces and dependencies need to be specified. Where necessary, the contracts should be further explained in notes. Names should reflect the nature of the component (e.g. Data Access, User Interface or Warehouse Logic) and interfaces (e.g. Persistent Data, Volatile Data or Inventory).

A component should implement a single, related set of functionality. If classes or modules are tightly coupled, then this is an indicator that they would be better in the same component to reduce the number of messages that have to be passed over the network. It is also useful to focus on the size on individual messages. Cyclic dependencies between components should not be tolerated because then independence is not given. An internal change in a component, for instance the change of a class or module, should not affect other components. It should only be necessary to change other classes/modules within this component (Common Closure). If all these guidelines are satisfied, then there is a good probability the design will serve its purpose.

Chapter 3

Current Research

In this chapter, an overview of the current research in related fields is given. In detail, it will be explained why modern GRIDs use Web Service related techniques and what extensions are necessary for that. Then the previous efforts for componentizing of legacy application — like DOUG — are pointed out. After that, an overview of the development of DOUG is given, what it does and what are current challenges.

3.1 GRIDs

GRIDs[1] are used for some time now to increase the utilization of computing and storage hardware by sharing them in a virtual organization. They are also used to tackle very big problems that could not be solved on computers available on local networks. The actions around the creation of GRIDs aim to deliver practically unlimited computational power and storage on demand very much like current electrical grids supply us with electricity. With GRIDs CPU time and storage could become a commodity that can be traded. For this purpose the users of a GRID are grouped in *virtual organizations* that put providers and consumers into a relationship. This is very similar to a neighborhood community that produces, shares and consumes fruits or vegetables. Often providers and consumers are the same persons.

In figure 3.1 you can see GRID users who are grouped into the virtual organizations VO1-VO3. One user is member in two different virtual organizations.

[1] In this thesis the term grid is used ambiguously: Most of the time the computer architecture for distributed computing is meant, but sometimes also the topology used in mathematics. In the Chair of Distributed Systems of the University of Tartu, where this work was produced, it has been proven to be a good practice to name the former GRID while the latter is refered to as grid. The different capitalization helps to clarify the meaning. GRID is not an abbriviation.

There are also three GRID resources displayed (R1-R3) which are grouped into two resource clusters RC1 and RC2. The virtual organization VO2 is allowed to use RC2. VO3 can use RC1. The virtual organization VO1 is not allowed to use any of the resources shown here.

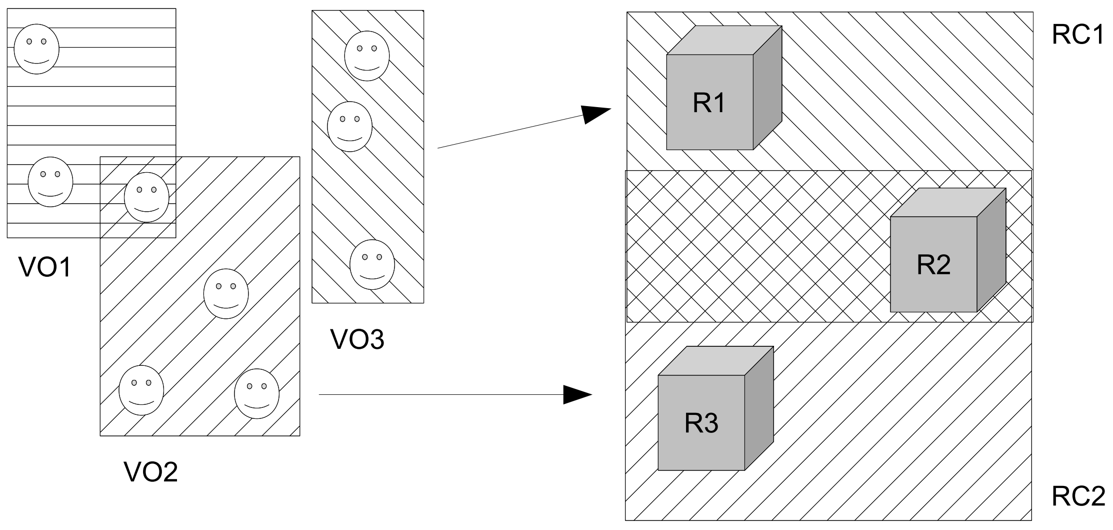


Figure 3.1: Virtual Organizations accessing different and overlapping sets of resources. (repainted from [Wik05])

Typical GRIDs are built from resources (computational and storage hardware), an interconnecting network and the GRID middleware that enables the submission and execution of user jobs, accounting, authorization, authentication and resource brokering (allocating a job to a resource).

Several proposed architectural models for GRIDs exist. These can be classified as *traditional batch-oriented*, *service-oriented* and *peer-to-peer approaches.* While the peer-to-peer model is mainly used in storage GRIDs with very popular file sharing software (e.g. BitTorrent [Coh03], Gnutella [Rip02] or eMule [KB05]), computational GRIDs shift from traditional middlewares like Globus Toolkit 1 & 2 or BalticGrid, to service oriented architectures (SOA). This is mainly fueled by two standards. The first is the Open Grid Services Infrastructure (OGSI) standard which has been incorporated into Globus Toolkit 3. The second is the state of the art Web Service Resource Framework (WSRF) standard which is the driving standards behind Globus Toolkit 4 [Fos05] or UNICORE [ES01].

Plain Web Services are not enough to form GRIDs. Applications often demand special properties of a host. For example an algorithm could been specialized to utilize command extensions of a special processor or it has been optimized for limited memory usage at the cost of CPU time demand. For accounting reasons, computation time is also often limited — a property which is not in scope of the old Web Service standards. All this had to be taken into account in the work towards OGSI. OGSI extends the specification of Web Services by the following topics [CFF+04]:

- WSDL constructs and operations for representing, querying and updating service data which includes meta and state data.
- The Grid Service Handle which is used to address a GRID service.
- A common base fault[2].
- A set of operations for creating and destroying GRID services.
- Mechanisms for asynchronous notifications of changes, which is resembles an Observer pattern [GHJV95].

After the specification of OGSI, the still young field of Web Service world evolved. New Web Service related standards have been made and the continued usage resulted in the emergence of best practices and patterns. There was also a lot of criticism towards OGSI. The main points of criticism on OGSI are the following [CFF+04]:

- *OGSI is too large.* There is not a clean factoring (separation) of topics in the specification. For example, the mechanisms for notification are useful also in other contexts.
- *OGSI has problems with existing Web Service tools.* It extends WSDL 1.1 using XML Schema extensively. For example, there have been problems with JAX-RPC[3].
- *OGSI is too focused on object oriented design.* Web Services are not a technique for distributed objects and they have different properties. The main difference is that objects have a lifecycle that starts with the instantiation of the object then a consumer issues operation on the object and later releases the instance so the memory can be reused. A web service does not have instances. Another important difference is that distributed object systems have extensive support for handling references to other objects. [Vog03]

[2]A fault is the output of a Web Service operation in case an error occurs. In that case, the fault replaces the normal output message of the operation.[CCMW01]

[3]JAX-RPC [Rob03] is a Java library that supplies functionality for mapping Java types to XML and therefore hiding the details of XML from the Java developer. It is one of the pillars of Web Service implementations in Java.

- *OGSI extended WSDL 1.1 although it should have waited for WSDL 2.0.* OGSI needs constructs which are introduced in WSDL 2.0, but because of delays in the specification of WSDL 2.0 the authors of OGSI made their own extensions to WSDL 1.1 instead. This lead to a less elegant solution and to the already mentioned problems with tooling.

The specification of WSRF was the answer on this criticism. WSRF is actually a family of specifications hence the 'F' for framework. It comprises five single specifications as showed below.

- *WS-RessourceProperties (WSRF-RP):* Describes how to combine stateful resources and Web Services to a WS-Resource and how its properties can be manipulated.
- *WS-RessourceLifetime (WSRF-RL):* Describes how to destroy a WS-Resource at a given time.
- *WS-RenewableReferences:* Describes how a new reference to a WS-Adressing endpoint can be obtained when the current reference becomes invalid.
- *WS-ServiceGroup (WSRF-SG):* Describes how to build and use a collection of Web Services.
- *WS-BaseFault (WSRF-BF):* Describes a standardized base fault type.

The WSRF specification alone does not represent a complete substitute for OGSI. *WS-Notification (WS-N)* is family of specifications [OAS07] that defines how to implement the Observer pattern with Web Services. Also needed to replace functionality of OGSI is the *WS-Addressing* specification [W3C06] which defines how Web Services and messages can be addressed in a transport-neutral way. This is necessary because Web Services do not need to be run by a web server but could for example be made available by SMTP.

In figure 3.2, you see a grouped overview of the specifications needed for GRIDs. Together these specifications make up a factored substitute for OGSI. Therefore, the critique that OGSI is too large has been addressed. Since WSRF is described in terms of WSDL, it does not add extensions and is therefore more compatible to existing tools. The terminology in WSRF has been changed so that the needed extensions separate more clearly the component oriented view on services from the object oriented features added in the specification. This is done by the introduction of the ResourceProperties which decorate a service and create a WS-Ressource.

The shift to a service oriented GRID — OGSI based or even better WSRF based — enables GRID resources to expose their capabilities through standardized interfaces which allows GRID users to integrate these services in their own applications through open-standard interfaces.

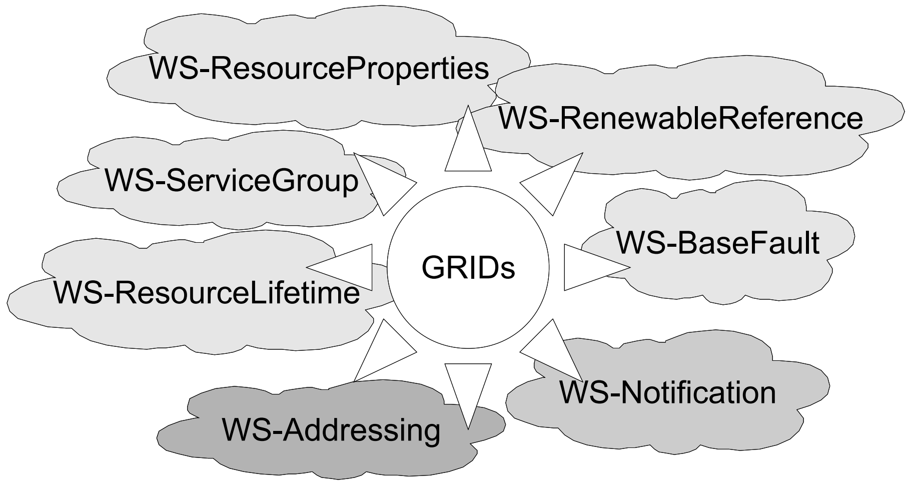


Figure 3.2: WS-* specifications used in modern GRID frameworks.

3.2 Componentizing legacy applications

While teaching can prepare users for the new programming paradigms, it is still unclear how legacy applications can benefit most from service oriented grids. Often an old application cannot be completely rewritten to fit into the SOA paradigm. A good example is the *HadCM3 AOGCM* climate model used by the desktop grid project *climatepredition.net* [SKM+02] which consists of millions of lines of Fortran code that have been developed over 13 years. Due to the pressing issues of climate change there is no time to rewrite and validate the model.

[GKT+05] suggests to build a wrapper around a legacy application to make it available in a SOA. While this approach bears not much workload for the GRID user, several drawbacks have been identified. The biggest problem is the lack of interaction with the job, when it has been started. Any application which accepts input data during execution can not be considered for their approach.

More promising in terms of increased adaptability through adding, removing or replacing components is the componentization of the legacy application but there is not much research done yet. In [PMB+06] the authors describe the dissection of Jen3D, an Java application, which uses a middleware called *ProActive* that supports parallel, distributed and concurrent programming through distributed objects. The authors developed a software development process for componentization of legacy applications. They also showed that in their case the performance has not suffered significantly.

Many legacy applications that run on traditional GRIDs are written in non object oriented languages like Fortran or C. Often MPI is used. The object oriented paradigm is much closer to the service oriented paradigm than procedural languages like Fortran. Also forming components from applications using MPI seems harder than from applications using ProActive because the latter is made to fit well into the object oriented Java environment. Therefore, the author believes it is necessary to research componentization of typical legacy grid applications.

3.3 DOUG

DOUG (Domain decomposition On Unstructured Grids) is a software package which is developed at the University of Bath, England and the University of Tartu, Estonia since 1997. It is a parallel solver for very large linear systems with up to several millions of unknowns. It handles finite element discretization of elliptic partial differential equations which is given as input in the form of element stiffness matrices (elemental input) or as a single sparse matrix (assembled input). DOUG can handle 2D or 3D problems. DOUG itself was first implemented in FORTRAN 77 but has been completely rewritten in Fortran 95. To achieve fast processing times, DOUG uses automatic parallelization and load-balancing. Parallelization is implemented through MPI and it was taken care of that non-blocking communication was used as much as possible (cf. chapter 2.3).

Basically, DOUG solves a linear system like

$$Ax = b \tag{3.1}$$

where A is sparse and the dimension of A is very large. This is done with the iterative *Krylov subspace method.* It converges guaranteed to the exact solution in N iterations where $N = dim(A)$. It does so by approximating

$$x = A^{-1}b \approx p(A)b \tag{3.2}$$

where p is a "good" polynomial. Many algorithms exist which employ the Krylov subspace method. The problem is that due to possible rounding errors that origin from the usage of floating point numbers in the implementation these algorithms can be less robust as the theory suggests. Also, for large N the convergence rate is by far too slow. For that reason, a *preconditioner* is used. A good preconditioner transforms the original linear system into one which has the same solution but an iterative solver needs much smaller number of iterations [Saa96]. DOUG implements the *Preconditioned Conjugate Gradient (PCG)*, the *Stabilized Bi-Conjugate Gradient (BiCGstab)*, the *minimal residual (MINRES)* and the *2-layered Flexible Preconditioned Generalized Minimum Residual method (FPGMRES)* with left or right preconditioning.

The general algorithm used for creating the sub-problems that are assigned to each CPU is *domain decomposition.* The idea of domain decomposition is to decompose the problem in n sub-problems each of which will be solved on one CPU. In a way, it is similar to a divide and conquer scheme but with domain decomposition there is communication on the borders of the sub-problems involved. Usually, communication is considered to be more costly as CPU time so that the decomposition algorithm tries to minimize the borders. This technique is so wide-spread in High Performance Computing that super-computers support it in hardware by the network topology of the inter-CPU connections. In figure 3.3, a decomposition of a problem with assembled input done by DOUG is shown. The problem is here divided into four sub-problems. On the borders of the sub-domains communication occurs.

An interesting property of domain decomposition is that it can be used as a preconditioning step for Krylov subspace methods such as the conjugent gradient method or the method of generalized minimum residual [SBG04]. DOUG employs 2-level preconditioning in which a coarse matrix is used which approximates the global matrix on a suitable chosen coarser scale. This reduces the total work of a preconditioned Krylov method (like PCG) to almost a constant number of iterations independent of N.

Recently, the development and research has been focused around aggregation based domain decomposition methods. Major progress has been made in determining an upper bound for the *condition number* of the preconditioner in case of highly variable coefficients. This allows a better estimate of the error and thus enables the solver to finish in less iterations. This approach has been implemented in DOUG and it has been show experimentally that it is of superior speed to comparable methods.[SV06a, SV06b] One big problem in the development of DOUG is to keep the source code manageable. There are two related problems in particular. The first is that two different version of DOUG exist one for elemental input (doug_main) and one for assembled input (doug_aggr). This is a usability problem, but also a problem for the programmers, because of duplicated code.

Due to research performed with the help of DOUG the code changes fast. As with most software systems, features which have been added are usually not removed and make easy understanding and maintenance more difficult if no countermeasures are deployed. The most important countermeasure to retain maintainability over time is *refactoring*.

3.4 Refactoring Fortran

While changing legacy applications with the aim of using them in a service oriented GRID, one must often use refactoring to improve the structure of the source code. Fowler [Fow99a] defines *refactoring* as

> a change made to the internal structure of software to make it easier

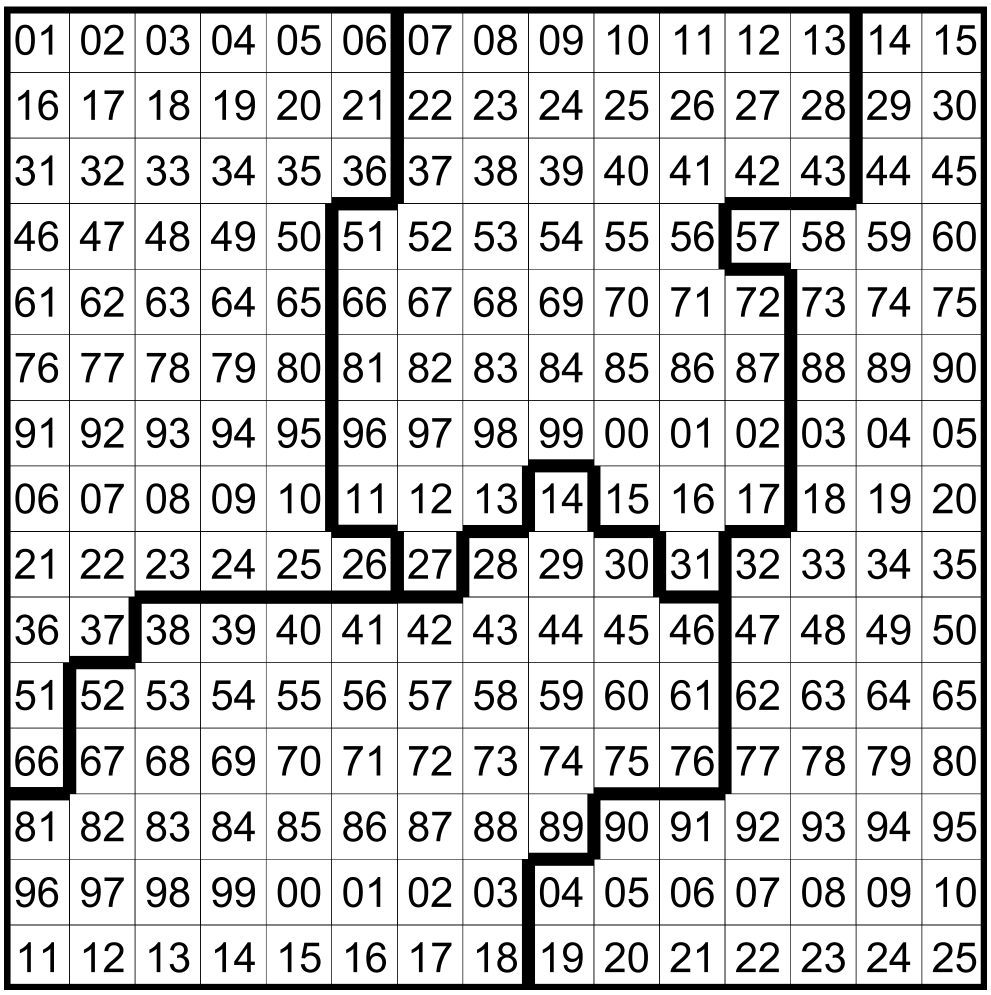


Figure 3.3: Domain decomposition of 15 × 15 matrix by DOUG.

to understand and cheaper to modify without changing its observable behavior.

Refactoring should be done when a feature, change or bug fix is estimated to take longer without prior refactoring. It should also be used when the implicit knowledge about the application in the programmer's head should be made explicit, for example when he leaves the project. Although good documentation also helps in this regard, good source code is self-documenting: It has a well defined design, uses just the right amount of comments, the scope of subroutines and function is so that they do only one thing, their name is well chosen and the structure inside the routines is easy to understand (e.g. a low cyclomatic complexity [Som07]).

Fowler also defined several hints to figure out when code needs refactoring. These hints are not given in a clearly defined way but more like a empiric measure. He coined these hints *smells*. For example the smell *Long Method* means that a long method (or subroutine in Fortran) suggests shortening it, e.g. by extracting smaller methods. Still, there can not be a definition when a method is long. Kerievsky suggests in [Ker04] a maximum of 5-10 lines of code as a rule of thumb but of course this also depends on the functionality.

In the field of object-oriented methods, refactoring has been very helpful to numerous developers and many tools exist to support easy application of refactorings to problematic code. Most of the refactorings are easily applicable to Fortran as well sometimes with minor changes. De [De04] made a list of refactorings that are usable in Fortran 90. Sadly, there are no production quality tools yet that can apply these refactorings to the code, so that careful manual refactoring is necessary. In [OXJF05], *Photran*, an Eclipse based IDE, has been introduced that supports automated refactorings. Unfortunately, it is still in an early stage and was not useful for our purposes because the Fortran parser used is not able to parse most of our source code correctly.

That does not mean that there is no support for refactoring in Fortran at all. The amount of possible refactorings is just very limited. For example, there is a tool that converts the older FORTRAN 77 to the more modern Fortran 90 [Met96]. There is also a checker for coding conventions in FORTRAN 77 that can also automatically tidy the code [Bun]. Nevertheless these tools can not be considered a suit of refactoring tools that would help in the everyday work of a Fortran programmer. They are specialized and can therefore be only applied in a very few number of uses.

There are also a small number of commercial tools (e.g. Foresys, Understand and McCabe Integrated Quality) that promise to be useful in refactoring Fortran applications but they have not been evaluated in this thesis.

While refactoring, the programmer has to make sure the observable behavior of the software does not change. This can be time consuming. When the code must be tested manually after each small change, times for checking the results will take too long: The time for testing gets longer and longer the more tests are added and the developer loses the motivation for testing because the process is too boring. Therefore *self-checking tests* are needed. Self-checking means that the output is either success or failure. This is achieved by specifying the expected answer in the test. Ideally, the tested scope is small so that in the case of an error the bug can be located faster. For this reason, there are *unit testing* frameworks that help in the creation of a test suite. If the programmer makes sure the altered code is covered by the tests, he can also make a system test but the smaller the tested units are the safer the test is. My college Oleg Batrashev wrote a test framework that does convenient self-checking tests for DOUG [Bat].

For high performance computing applications of course also the execution speed is important. Therefore, the software to be refactored should also be profiled. Mai-Liis Karring, a current master thesis worker at our chair, works at the moment on an mechanism to systematically measure performance of DOUG [Kar].

With functional and performance tests in place, the programmer can start refactoring in very small iterations. He should apply one refactoring then compile and test. If the tests take too long then a selection of the most important tests (a suite) should be run to reduce the time of iteration cycles. At some point, for example before a check-in into the version control system, all tests should be run, too, to make sure nothing unexpected broke by applying the refactorings.

Also very helpful for the author's understanding of the DOUG source code was the Fortran Doxygen extension [VB] developed by my colleague Oleg Batrashev and Anke Visser of the Central Institute for Applied Mathematics in the Research Centrum Jülich. The comments are parsed with the help of special leading characters (`!<`, `!>` and `!!`) very much like javadoc comments. By now, many subroutines and functions as well as their parameters have been properly documented and are available as HTML pages.

3.5 Necessary Research

At the moment GRIDs shift from traditional batch processing systems to service oriented architectures. Web Service techniques are used to establish the SOA. Legacy applications need to be converted to make them run on modern GRIDs. Typical GRID users as well as application programmers are often not familiar with the characteristics of components which are needed for SOAs. Often the applications are made in a way that they do not fulfill the requirements for components. That is why refactoring can be applied to such applications to improve the structure of the applications before adding new functionality.

By establishing a standard process to make legacy applications available on the GRID, the users and programmers are much more likely to make a successful transition. That is why software engineering knowledge is needed to establish such a process. DOUG is a good example for such a legacy application which is why it will be used as a case study.

Chapter 4

DougService

Goal of this project was to use DOUG as a component. This component is connected to other components to form an Eigenvalue solver. To know the Eigenvalue and/or -vector of a matrix can be useful in several applications: In computer science, Eigenvectors can be used to efficiently compress similar images like faces. In biometrical application, this can be used to identify humans. Quantum mechanics uses Eigenvectors to determine the ionization potential of molecular orbits. This is the energy needed to strip a molecule of one electron. Both elemental and assembled input can be given to the Eigenvalue solver and the nearest Eigenvalue to an also given value called *shift* is returned. The Eigenvalue solver is itself a component, too, and can be used for example by an graphical user interface, as done in this work, or another application.

4.1 Architecture

4.1.1 Component technology

The first architectural question that was asked was what technology could be used to define and connect components that will make up the Eigenvalue solver. Although the solver has not been run on the GRID, similar technology should be used so that the migration to the GRID could be done easily in the future. Because most modern GRID frameworks use Web Services, the choice was straight forward. The Web Service framework that was used was chosen with respect to further development. Since the premier service oriented GRID framework at the moment is the Globus Toolkit 4 and it uses Axis [SC06], the same framework was chosen. Another reason was that long term production use proved Axis' stability and flexibility.

Note that although Axis2 [Apae] is a newer, rewritten implementation and there is a post-beta release that is deemed fit for production use, first experiments were discouraging. At least for version 1.0 it must be said that the documentation is bad and outdated, sometimes plain wrong and there were several bugs in the software. For that reason the older version — Axis 1.4 [Apac] — was used. This is the last version before the rewrite and it was found to be suitable for the task.

4.1.2 Component scope

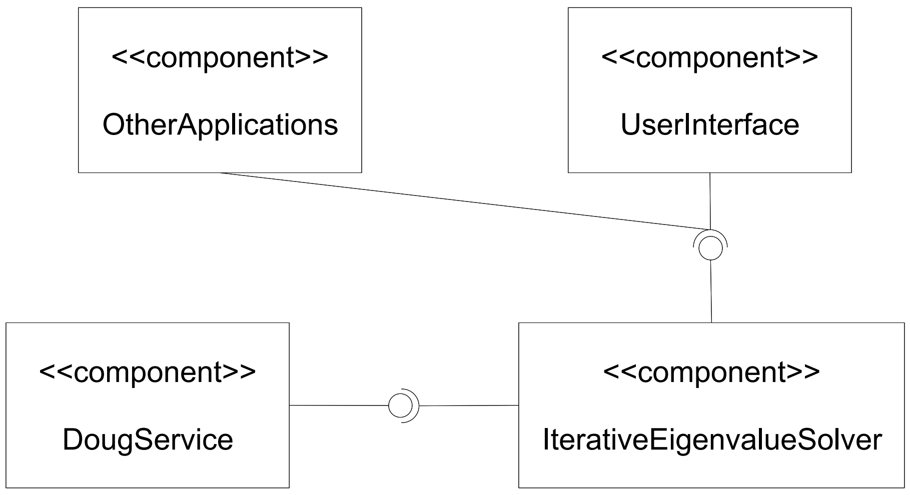


Figure 4.1: Component diagram of the Eigenvalue solver

With the component technology chosen, the next question was what components would be chosen and what interfaces should be exposed. As the reader can see in figure 4.1, the Eigenvalue solver is composed out of two components and can be used by other components. Each of the components is modeled as a Web Service and exposes its operations for other components to use.

The first component is the `DougService`. It contains the whole logic for DOUG and exposes its functionality through one single interface. The second component is the *IterativeEigenvalueSolver*. It contains the algorithm for the solver and also exposes its interface for other applications. It completely hides the `DougService`. Therefore the `DougService` can be easily exchanged with another linear equation solver as long as its interface is implemented. If necessary, an Adapter pattern [GHJV95] can be used to covert the interface.

4.1.3 DougService — Wrapper for DOUG

DOUG is needed as a component but it is written in Fortran while Axis requires C++ or Java. Therefore, a wrapper was built around DOUG which performs the following steps:

- Recieve and write the input files to the harddrive,
- call the DOUG executable and wait until it finishes,
- parse the solution files and return the result.

At a later stage it should be considered to use JNI to access a — yet to be created — library of DOUG. Since this work is protoypical, some flexibility in the configuration of DOUG has been limited intentionally to achieve faster development times. Nevertheless, it was taken care of that adding more flexibility in configuration can be done without much work.

The problem description at the beginning of this chapter gave rise to the interface specification in listing 4.1. The operation `runAssembled` solves the mathematical equation $Ax = b$ where A is the matrix in assembled form, b is the right hand side vector and x is the returned vector. The last parameter is the control file which is passed by the client to set the configuration of DOUG. There are two overloaded flavors of `runAssembled`: The first one uses JavaBeans and formatted text as input and the second version uses WS-Attachments and XDR binary files.[1] The differences will be explained in chapter 4.1.5. The operation `runElemental` also solves a linear system but it is given in elemental form. The last operation, `elementalToAssembled`, copies all elemental form files and uses DOUG to convert them to assembled form.

For the sake of completeness, it should be said that while familiarizing with Axis a different approach was chosen. A Web Service without the use of WS-Attachments was considered where all the data was supposed to be on the server already, maybe put there via WebDAV or SCP. Then the Web Service's duty was just to start the right executable. The data returned from the call is an array of two Strings: the standard out (*stdout*) and standard error stream (*stderr*). The — hopefully successful — result was in the *stdout*.

Of course there are several problems with this approach:

- The user needs access to some portions of the file system and can put any files there.

[1]WS-Attachments or SOAP Attachments is a standardized extension of Web Services that adds a file to the SOAP message just like an attachment is added to an eMail. Actually, in Axis even the implementation of the JavaMail API is reused for that purpose. Attachments are represented by `DataHandler` objects in Axis.

```java
import javax.activation.DataHandler;

public interface I_DougService {

    public DoubleVector runAssembled(AssembledMatrix matrix,
            DoubleVector rhs, DataHandler control_file);

    public DataHandler runAssembled(DataHandler matrix,
            DataHandler rhs, DataHandler controlfile);

    public DoubleVector runElemental(
            DataHandler freedom_lists_file,
            DataHandler elemmat_rhs_file,
            DataHandler coords_file,
            DataHandler freemap_file,
            DataHandler freedom_mask_file,
            DataHandler info_file,
            DataHandler control_file);

    public AssembledMatrix elementalToAssembled(
            DataHandler freedom_lists_file,
            DataHandler elemmat_rhs_file,
            DataHandler coords_file,
            DataHandler freemap_file,
            DataHandler freedom_mask_file,
            DataHandler info_file,
            DataHandler control_file);

}
```

Listing 4.1: Interface `I_DougService`

- Security policies are implemented e.g. via Tomcat's user management (for using WebDAV from Tomcat) or system users (SCP). This is not suitable for GRIDs mechanisms of authentication and authorization with virtual organizations.

- Between putting the files in the right place and starting the calculation can pass time in which another user can overwrite the files. So the user gets results from someone else which might lead to wrong decisions if they are based on that data.

- *stdout* must not necessarily contain the result. If the constant `D_MSGLVL` in the file `globals.F90` is set to 0 or if the problem has more than 100 unknowns, then the result is not written to stdout.

For documentation purposes, the operation is still available but is marked *deprecated*. After getting more experience with Axis, the final implementation was made.

```
public interface I_IterativeEigenvalueSolver {

    public EigenSpace inverseIteration(AssembledMatrix a,
            DoubleVector initialGuess, double shift, double error)
            throws IOException, DougServiceException;
}
```

Listing 4.2: Interface `I_IterativeEigenvalueSolver`

4.1.4 IterativeEigenvalueSolver – Calculating Eigenvalues

Because the Eigenvalue solver was to be exposed itself as a service, an interface has been supplied to emphasize the component architecture. The sole algorithm that is availible so far is the Inverse Iteration described by [Gu00] but other numerical Eigenvalue solving algorithms could implemented later using the same design as in the current implementation. The Inverse Iteration calculates an Eigenvector with corresponding Eigenvalue thus `inverseIteration` returns a usually non-complete Eigenspace. It has four inputs: The first input is the matrix which Eigenvalue is to be calculated. The second is the initial guess of the Eigenvector. The third is the shift which is a scalar close to the returned Eigenvalue and the last is the error boundary which determines the precision of the solution and also influences the number of iterations needed.

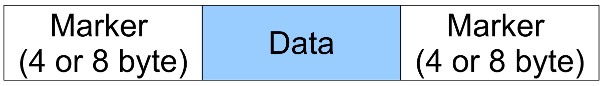


Figure 4.2: Unformatted data is written enclosed by markers.

4.1.5 Interoperability with text and binary data files

While the interface specification is very clear, there were some changes in DOUG that had to be implemented. Those changes were all about the flexibility how DOUG handles input and output files. In DOUG ”unformatted” Fortran binary files are used in contrast to text files which can be formatted by a special string. Because of limitations in unformatted files, that are based in the language specification of Fortran, the binary format could not be used without jeopardizing interoperability. Binary files produced by Fortran programs are not only architecture-dependant (in particular endianess is a problem: little endian vs. big endian) but also compiler dependent: Each binary file is a series of enclosing markers and data bytes (see diagram 4.2). In the marker, the number of enclosed data bytes are stored. GFortran chooses eight bytes for the header while the ifort compiler chooses only four bytes. This is not suitable for a diverse environment like the GRID because one cannot rely that the same compiler is used everywhere. For that reason ASCII based text formats have been used which inflates the size by a two digit factor. It also slows down the processing speed due to parsing and increases the memory footprint drastically. For very large problems, this is not feasible. Even for medium size test examples this is not a good solution. Therefore effort has been put into a new binary data format that suffers not from the drawbacks described above.

The FXDR library [Pie] was very helpful in the creation of this new format. FXDR is a Fortran wrapper around the XDR (eXternal Data Representation) library which is shipped with virtually any Unix system. In practical terms, FXDR allows programmers to read and write unformatted data that can be used on every computer where FXDR is installed. Therefore a huge gain in interoperability is achieved while a binary format is still possible. The XDR library was designed specifically for RPC so it is a good fit with Web Services. The bit representation of the data written with FXDR is standardized by the IEEE which is also an improvement over traditional Fortran unformatted I/O. Files in XDR format are even marginally smaller than traditional unformatted files because no makers are saved to the file.

FXDR support has been built into the assembled version of DOUG. The right hand side vector and the solution vector can be configured to be read or written either in formatted text, unformatted traditional Fortran binary or unformatted

XDR binary. The matrix can be used as formatted text or XDR binary. A converter — `txt2xdr` — has been written that converts formatted text matrices and vectors to XDR format. A service operation has been added that uses the XDR format files as SOAP attachments.

4.2 Implementation issues

4.2.1 Control file vs. set of default CLI parameters

As already mentioned in chapter 4.1.3, a run of DOUG is configured with a set of parameters that are given in the control file. A typical control file is shown in listing 4.3. One can argue if the control file should be passed by the client or if it should be generated by the service. As the client can set important configuration options for the operation of DOUG, this information has to be passed to the service and finally end up in the control file. Therefore *some* configuration data has to be passed. But in the control file there are also some parameter, which will not change over time.

Because of the prototypical nature of this implementation, an easy to implement solution has been chosen: The control file is sent by `DougWSClient`. A set of constants is shared by both `DougWSClient` and `DougService` to make sure the control file matches the location where the files are stored on the hard drive prior to execution.

The principle of *separation of concerns* refers to the requirement that distinct features overlap in functionality as little as possible. This eases maintainability of an software program because solutions of a sub-problem can be combined to form a solution to a bigger problem. An example where the separation of concerns is violated are the file names of the data files which are set by `DougService` anyway even if they are set in the control file. Separation of concerns dictates that they should not been written in the control file by `DougWSClient`.

The author believes that DOUG will benefit if it will not use a control file anymore but use default values for the controlled options that can be overridden by the usage of command line interface arguments. The rationale for this is that the user does not bother to understand and set all the options himself. This would not only lead to better usability of DOUG but also simplify the options management of connected components. Only the non-default parameters would have to be transferred to `DougService` and the wrapper would not have to manipulate anything in the control file, just add a command line parameter.

```
solver 2
method 1
input_type 1
matrix_type 1
info_file doug_info.dat
freedom_lists_file doug_element.dat
elemmat_rhs_file doug_system.dat
coords_file doug_coord.dat
freemap_file doug_freemap.dat
freedom_mask_file ./NOT.DEFINED.freedom_mask_file
number_of_blocks 1
initial_guess 2
start_vec_file ./NOT.DEFINED.start_vec_file
start_vec_type 2
solve_tolerance 1.0e-12
solution_format 0
solution_file ./solution.file
debug 0
verbose 10
plotting 0
assembled_rhs_file ./NOT.DEFINED.solution.file
assembled_rhs_format 0
dump_matrix_only true
dump_matrix_file ./dump_matrix.file
```

Listing 4.3: An example DOUG control file for a conversion from elemental input to assembled input

4.2.2 Two executables

Another problem is the split-up of DOUG in two different variations as described in chapter 3.3. The natural way of determining the input format (assembled or elemental) automatically is not easily possible without major redesign of the application. It could be implemented into the wrapper which is part of the Web Service but automatic input format recognition is clearly the responsibility of core DOUG because it is also used as a stand alone application. This added unnecessary complexity to the wrapper which could have been avoided by having only one single executable. While automatic recognition of the input data is mainly a usability problem — the user should not be bothered with tasks that can be done automatically, as Raskin states in [Ras00] — it also led to distinct Web Service code for the different varieties of DOUG. Additionally, other applications that use DOUG will be more complex.

4.2.3 Smelly: Code Duplication

Very close related is the problem that separate executables are compiled from different source files that share code. Parts of the changes in the Fortran code had to be done twice which is very prone to bugs and leads to even worse code quality the more will be changed. Fowler calls possible reasons to refactor *smell* and *Code Duplication* is the most common smell [Fow99b].

A good example is a code snippet from the source files `main.F90` and `aggr.F90` given in listings 4.4 and 4.5. The code snippets are exactly the same except of some additional whitespace in the latter and a different parameter to operation calls. A comment describes what the code does which is in itself a bad smell. Of course, comments are generally a good thing but if a block of code is commented then it is often so that a procedure with a similar name of the comment improves readability and maintainability.

Both smells, Code Duplication and Comment, suggest to use the refactoring Extract Method (which [De04] modified to Extract Procedure in Fortran). Therefore the next step is to make the subroutine. Because `main.F90` contains no module, the author chose to do a second refactoring — Move Method — in this step, too. The new subroutine has been placed in `main_drivers.F90` where already the subroutines `parallelAssembleFromElemInput` and `parallelDistributeAssembledInput` are located. Because the new subroutine chooses between the two algorithms based on a flag, it is a good fit. The name of the new subroutine `SelectInputType` has been chosen by using the comment that refered to the block of code. The final code is depicted in listing 4.6. After adding comments to the method and adding the method to the list of public procedures, the code is compiled and tested. Then the code snippets in listings 4.4 and 4.5 is replaced by the

```
! Select input type
select case (sctls%input_type)
    case (DCTL_INPUT_TYPE_ELEMENTAL)
        ! ELEMENTAL
        call parallelAssembleFromElemInput(M,A,b,nparts,&
            part_opts,A_interf)
    case (DCTL_INPUT_TYPE_ASSEMBLED)
        ! ASSEMBLED
        call parallelDistributeAssembledInput(M,A,b,A_interf)
    case default
        call DOUG_abort('[DOUG main] : Unrecognised input type.',&
             -1)
end select
```

Listing 4.4: Code snippet from `main.F90`

```
! Select input type
select case (sctls%input_type)
    case (DCTL_INPUT_TYPE_ELEMENTAL)
        ! ELEMENTAL
        call parallelAssembleFromElemInput(M, A, b, nparts, &
            part_opts, A_ghost)
    case (DCTL_INPUT_TYPE_ASSEMBLED)
        ! ASSEMBLED
        call parallelDistributeAssembledInput(M, A, b, A_ghost)
    case default
        call DOUG_abort('[DOUG main] : Unrecognised input type.',&
             -1)
end select
```

Listing 4.5: Code snippet from `aggr.F90`

new subroutine call (listing 4.7) one after another. After each replacement the code is compiled and tested to make sure no new errors have been introduced.

```
!--------------------------------------------------------------
!> Distributes data, chooses algorithm based on input type
!--------------------------------------------------------------
subroutine SelectInputType(input_type, M, A, b, nparts, &
    part_opts, A_interf)
  implicit none

  integer,          intent(in)      :: input_type !< Input Type
  type(Mesh),       intent(in out) :: M !< Mesh
  type(SpMtx),      intent(out) :: A !< System matrix
  float(kind=rk), dimension(:), pointer :: b !< local RHS
  ! Partitioning
  integer, intent(in) :: nparts !< number of parts
                              !! to partition a mesh
  integer, dimension(6), intent(in) :: part_opts !< partition
                              !! options (see METIS manual)
  type(SpMtx),intent(in out),optional :: A_interf !< matrix at
                              !! interface

  select case (input_type)
  case (DCTL_INPUT_TYPE_ELEMENTAL)
     ! ELEMENTAL
     call parallelAssembleFromElemInput(M,A,b,nparts, &
        part_opts,A_interf)
  case (DCTL_INPUT_TYPE_ASSEMBLED)
     ! ASSEMBLED
     call parallelDistributeAssembledInput(M,A,b,A_interf)
  case default
     call DOUG_abort('[DOUG main] : Unrecognised input type.', -1)
  end select
end subroutine SelectInputType
```

Listing 4.6: New extracted subroutine in `main_drivers.F90`

Now it becomes clear that the name for the subroutine as well as the previous comment is not very self descriptive for its functionality. Therefore, the refactoring Rename Method is used to clarify this. Both the subroutine definition in `main_-drivers.F90` as well as the calls in `main.F90` and `aggr.F90` are changed. After the changes, the code is compiled and tested again.

By looking in the list of public procedures of the module in `main_drivers.F90` one wonders, if now that `parallelAssembleFromElemInput` and `parallelDis-tributeAssembledInput` are not accessed from `main.F90` and `aggr.F90` anymore

```
call SelectInputType(sctls%input_type,M,A,b, &
                     nparts,part_opts,A_interf)
```

Listing 4.7: Code blocks in `main.F90` and `aggr.F90` have been replaced by subroutine call

the visibility of these routines can be reduced. The compiler complains if the subroutines are private but used externally so this mechanism can be used as a reporting tool. To be sure and find code that is obsolete and not called a full text search is done additionally in all source files. This is tried first with `parallelDistributeAssembledInput` and it surfaces this subroutine does not have to be public anymore. Afterwards, the same is done with `parallelAssembleFromElemInput`. While the code still compiles, the full text search returns unexpected results: A program `test_SpMtx_symmetry_at_pmvm` in a file with the same name exists in which `parallelAssembleFromElemInput` is called. This means that this file is not compiled. Because the program is not included in the build process, changes to the file cannot be tested yet. Therefore, it is decided that only `parallelDistributeAssembledInput` is removed from the list of public procedures. The source is compiled and tested one more time then the changes are committed to the version control system.

The refactoring of similar parts of the code will be crucial for ensuring quality and development productivity in the future. Therefore the author urges strongly to invest more effort into refactoring. Because of the short iteration cycles of coding, compiling and testing it is important that compiling and testing are fast enough. At the moment a typical compilation run takes about 40 seconds and a test run of a suit of reasonable size takes around 15 seconds.[2] If these times could be reduced, so could the time for refactoring.

4.2.4 Refactoring with Fake Polymorphism

An other particular problem in the maintainability of the DOUG source is the big number of conditional statements like `select case` and `if then else`. Nested conditionals make the code much harder to understand and maintain. Long procedures are more likely and the code is less self-expainatory. This became apparent in the adding of additional file formats. Note that in object oriented languages one would use polymorphism to solve the problem: An interface or an abstract class would have been created and implementations or inherited concrete subclasses would execute the different works. This is referred to as Strategy pat-

[2]Measurements have been taken on an AMD Athlon 64 3000+ at 1.8 GHz with 1 GB RAM.

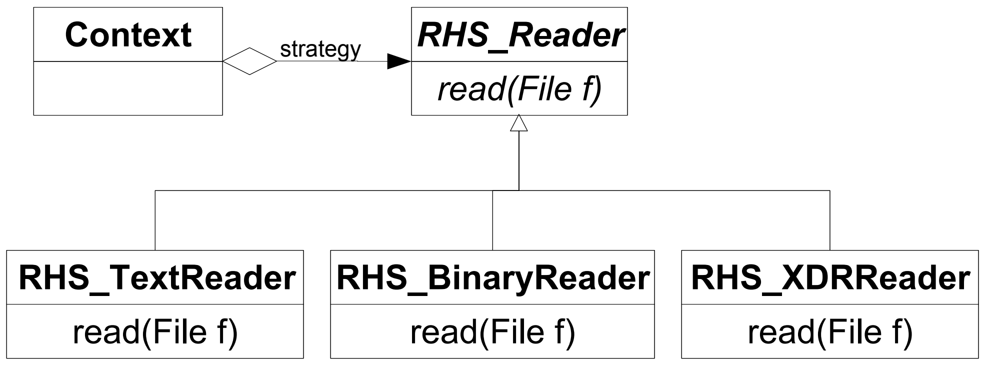


Figure 4.3: In OO a Strategy pattern would be used to reduce conditionals.

tern [GHJV95]. In figure 4.3 is an example UML class diagram with the entities used in DOUG's right hand side vector reader.

In a non object oriented language like Fortran this pattern is not applicable. Therefore an approach has been chosen that can be applied in the described circumstances. No research has been done if this approach is also used in other projects, so it cannot be said that it qualifies as a pattern. Nevertheless, it is abstract enough to be applied as if it was a pattern. Because it is a substitute for polymorphism, it is called *Fake Polymorpism.* Its idea is that the conditional statement is extracted into a procedure where only the conditional is situated. The code blocks within the conditionals get extracted again into one procedure each with the same name, but with an additional case marker at the end. In the end there are *number of conditions + 1* procedures more than before.

In listing 4.8 an shortened example is given. An additional data format must be added to the code. There are already two formats present and the difficult nesting is distracting the reader. First the call to `FindFreeIOUnit` as well as its error checking was moved into the blocks for text respectively binary version because the new format does not need to find a free IOUnit. Then each block of the conditional is moved to another subroutine with the same signature except the `fmt` parameter. Instead, the name of the procedure is extended with the specific format it handles. This means that `Vect_ReadFromFile` becomes `Vect_ReadFromFile_Text` and `Vect_ReadFromFile_Binary`. Then the last format can be added easily. Finally a few variables have been renamed for clarity. The final (shortend) code can be seen in listing 4.9.

The benefit is of Fake Polymorphism is that

- the procedures become shorter,
- the code less is nested,

```fortran
!-----------------------------
!> Read vector of floats from file
!-----------------------------
subroutine Vect_ReadFromFile(x, fnVect, fmt)
  implicit none

  float(kind=rk), dimension(:), pointer :: x
  character*(*), intent(in)             :: fnVect
  integer, intent(in)                   :: fmt

  logical :: found
  integer :: iounit

  call FindFreeIOUnit(found, iounit)
  if (found) then
    if (fmt == D_RHS_TEXT) then
      ! some
      ! lines
      ! of code
    elseif (fmt == D_RHS_BINARY) then
      ! some
      ! lines
      ! of code
    else
      call DOUG_abort('[Vect_ReadFromFile] : Wrong format', -1)
    endif
  else
    call DOUG_abort('[Vect_ReadFromFile] : No free IO-Unit', -1)
  endif
  return
end subroutine Vect_ReadFromFile
```

Listing 4.8: An example where Fake Polymorphism should be used.

```
!-----------------------------
!> Read vector of floats from file
!-----------------------------
subroutine Vect_ReadFromFile(x, filename, format)
  implicit none

  float(kind=rk), dimension(:), pointer :: x
  character*(*), intent(in)             :: filename
  integer, intent(in), optional         :: format

  if (fmt == D_RHS_TEXT) then
    call Vect_ReadFromFile_Text(filename, x)
  elseif (fmt == D_RHS_BINARY) then
    call Vect_ReadFromFile_Binary(filename, x)
  elseif (fmt == D_RHS_XDR) then
    call Vect_ReadFromFile_XDR(filename, x)
  else
    call DOUG_abort('[Vect_ReadFromFile] Data format wrong.', -1)
  endif
end subroutine Vect_ReadFromFile

subroutine Vect_ReadFromFile_Text(filename, x)
  implicit none

  float(kind=rk), dimension(:), pointer :: x
  character*(*), intent(in)             :: filename
  logical :: found
  integer :: iounit, n, i

  call FindFreeIOUnit(found, iounit)
  if (.NOT.found) &
    call DOUG_abort('[Vect_ReadFromFile_Text] : No free IO', -1)
  ! open
  ! sanity check
  ! read
  ! error handling
end subroutine Vect_ReadFromFile_Text

subroutine Vect_ReadFromFile_Binary(filename, x)
  ! some code
end subroutine Vect_ReadFromFile_Binary

subroutine Vect_ReadFromFile_XDR(filename, x)
  ! some code
end subroutine Vect_ReadFromFile_XDR
```

Listing 4.9: The same code after application of Fake Polymorphism.

- reusability is encouraged because of a higher number of procedures,
- understandability is increased because of clear names of the procedures and
- future adding and removing of conditions can be done without harming the structure of the code.

Unfortunately, the additional indirections in the code make it harder for a human to follow the flow of the program although clear names and minimal sets of parameters will often reduce the need to look at the code itself.

4.2.5 Streaming vs. Attachments

From the very beginning it was clear that data has to be copied from one machine to another. In the current solution the data is in objects wherever it makes sense but there is also data which is a file by its very nature. For example it makes no sense to represent binary data files by a special class. The same applies for the control file within the limits described in chapter 4.2.1.

Of course there are several ways to copy a file on a network. The most promising for this application are *HTTP streaming* and *WS-Attachment*. While HTTP streaming is very easy to implement, e.g. by using Jakarta Commons HTTP Client [Apaf], it has also some drawbacks. The sequence of data exchange with HTTP streaming would be like this:

- The client copies the files to a location in the file system where they are availible via HTTP.
- The client invokes the service on the server and passes the URL as parameter.
- The server starts the download of the file from the client via HTTP.

Notice, that two transfers are necessary. This will most likely slow down the total transfer. Because the client needs to supply a HTTP server, memory usage will increase. This solution will be less stable since there are more security issues involved (e.g. to restrict access to the transfered file) and atomic requests are safer with heavy load of the service (”all or nothing”). With WS-Attachments there is a better solution since the file is encoded with MIME or DIME and attached to the SOAP message so that only one message s sent. Therefore, WS-Attachments is often also called SOAP with Attachments or MIME for Web Services. There is just one single remote procedure call that contains all the data. It also has an additional memory footprint because of a few more classes which have to be loaded but compared with the costly XML parsing it is still very reasonable. No additional security issues are introduced since the SOAP message contains all the needed data. Standard Web Service security mechanisms can be used. There is also no need for the programmer to handle incomplete data transfers where the remote procedure call (RPC) succeeds but the HTTP streaming fails. A RPC with WS-Attachments is either a success or a failure.

Another problem of HTTP streaming is limited flexibility in reaction to system environment changes. One of the advantages of Web Services is that the wire transport protocol can be changed independently from message content. In case, the provider of the service decides that it cannot offer HTTP anymore but limits external communication to SMTP. Web Service frameworks can offer different transport protocols as communication means and still be standard compliant. If HTTP streaming would have been implemented into the application this flexibility is severely limited. This is especially problematic in GRID environments, which at the moment suffer from the necessity to have too many open TCP and UDP ports. This is considered to make security on the GRID resources more difficult.

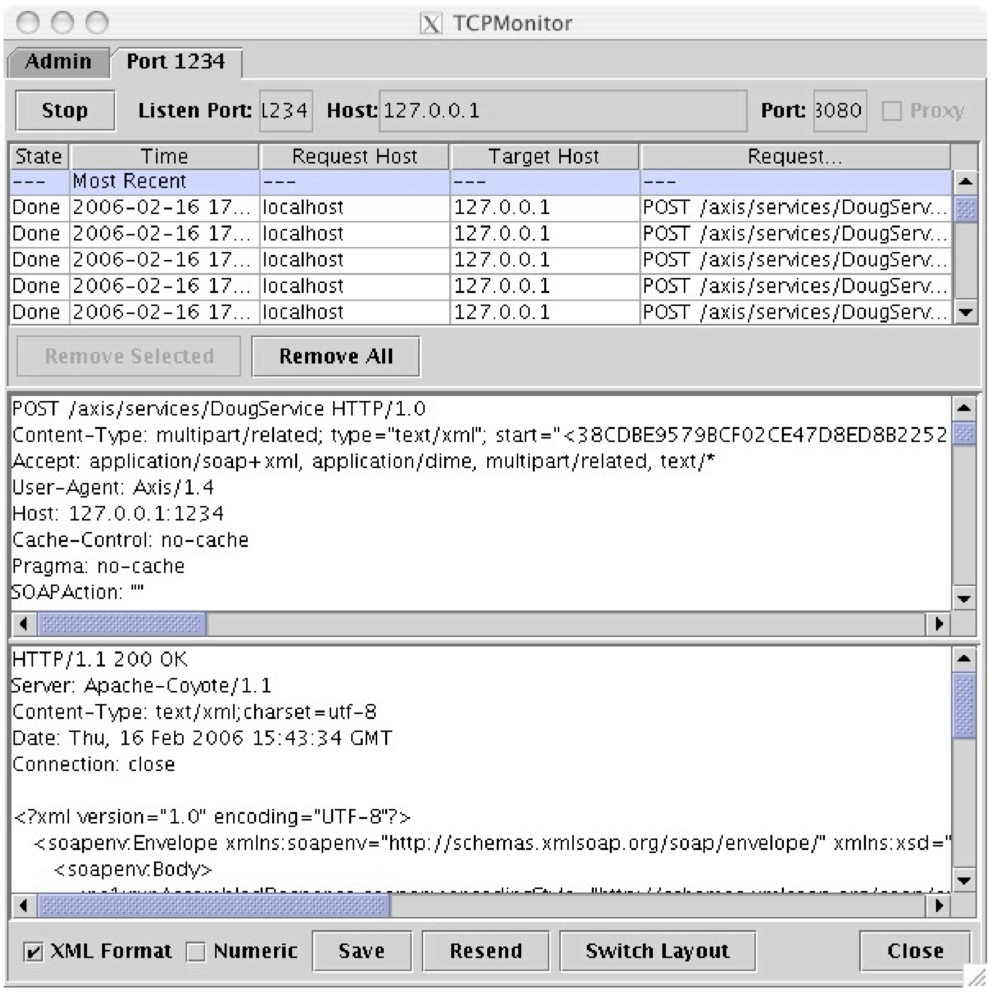
4.2.6 Debugging the RPC

It is fairly easy to log the SOAP messages that are transfered between the two peers. For this purpose, a proxy called TCPMonitor has been used that is situated between the Web Service and the client. It is shipped with Apache Axis and can be started with `java org.apache.axis.utils.tcpmon [listenPort targetHost targetPort]`. You can see a screenshot of the tool in figure 4.4. When the client is started, it has to change the port number in the endpoint address. By configuring the used port in the endpoint address, the programmer can easily use TCP monitoring or bypass it.

A usability problem occurred if the attachments are too big: then TCPMonitor can not handle the message anymore, throws an exception and does not pass the message to the Web Service. This was problematic since the exception was printed on the shell from which TCPMonitor was started. Therefore it was not directly visible and it took some time to figure out, why the service did not work anymore. The programmers of TCPMonitor should consider to open a message box explaining what went wrong or enable messages of unlimited length possible.

A similar tool is NetTool [O'T]. Although it lacks XML pretty printing, it can handle messages of virtually any size. It also measures time between call and response and the sizes of the messages. TCPMonitor was preferred because of the better readability of the messages but in the case of big messages NetTool is the tool of choice.

A different problem occurred because of typographical errors in the creation of the SOAP message and in the Web Service deployment descriptor. The first of those errors occurred after copy and pasting some source code, which registers how objects are mapped to the XML SOAP message. One typical (correct) piece of



Listen Port: 1234 Host: 127.0.0.1 Port: 8080 Proxy

State	Time	Request Host	Target Host	Request...
---	Most Recent	---	---	---
Done	2006-02-16 17...	localhost	127.0.0.1	POST /axis/services/DougServ...
Done	2006-02-16 17...	localhost	127.0.0.1	POST /axis/services/DougServ...
Done	2006-02-16 17...	localhost	127.0.0.1	POST /axis/services/DougServ...
Done	2006-02-16 17...	localhost	127.0.0.1	POST /axis/services/DougServ...
Done	2006-02-16 17...	localhost	127.0.0.1	POST /axis/services/DougServ...

```
POST /axis/services/DougService HTTP/1.0
Content-Type: multipart/related; type="text/xml"; start="<38CDBE9579BCF02CE47D8ED8B2252
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/1.4
Host: 127.0.0.1:1234
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: ""
```

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/xml;charset=utf-8
Date: Thu, 16 Feb 2006 15:43:34 GMT
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="
    <soapenv:Body>
```

Figure 4.4: The TCPMonitor in use while debugging SOAP messages.

```java
// Add (de-)serializer for AssebledMatrix.
QName qnameAssembledMatrix =
        new QName(Settings.NAMESPACE_ID, "AssembledMatrix");
call.registerTypeMapping(AssembledMatrix.class,
        qnameAssembledMatrix,
        new BeanSerializerFactory(AssembledMatrix.class,
                qnameAssembledMatrix),
        new BeanDeserializerFactory(AssembledMatrix.class,
                qnameAssembledMatrix));
```

Listing 4.10: Type mapping registration

code where a type mapping is registered is given in listing 4.10. If the programmer forgets to change some of the identifiers, then the code still compiles and runs but produces errors when serializing or deserializing the passed objects. Axis could be less error prone if it would assume a default qualified name (`QName`). Still, there has to be a mechanism to set the `QName` by hand in case of name clashes. Also for `Call#registerTypeMapping`, there should be an easier solution while the shown, very flexible solution can be retained. There are only four different cases for type mappings in Axis:

- primitive type mappings (`int`, `double`, `String`, etc.)
- array type mappings
- JavaBeans [Gra97] type mappings
- custom type mappings

Primitive types are mapped automatically by Axis. There is no need to register a primitive type. The mappings are specified in JAX-RPC [Rob03]. Arrays have to be registered only in the deploy descriptor[3]. Other objects have to be registered both in the deploy descriptor and the client source code. The necessary source code for registering a bean is shown in listing 4.10. To make the register process less error prone, there could be an additional method in `org.apache.axis.client.Call` with the name `registerBeanMapping`. See listing 4.11 for usage and listing 4.12 for a possible implementation. Custom type mappings have to be done like in listing 4.10 except that other serializer and deserializer objects than instances of the factory must be given.

```
// Add (de-)serializer for AssebledMatrix.
call.registerBeanMapping(AssembledMatrix.class);
```

Listing 4.11: Possible usage of JavaBean type mapping registration

4.2.7 Build process

The build process is typical compared to most Java projects. An ant [Apaa] build file has been supplied with the usual targets (*init*, *clean*, *compile* and *deploy*). For the sake of clarity the target *deploy* has been split up to two targets

[3]A deploy descriptor is a small XML file that configures the Axis runtime environment. It specifies which methods to expose and which special type mapping should be done.

```
public void registerBeanMapping(Class c) {
    call.registerTypeMapping(c,
            new BeanSerializerFactory(c),
            new BeanDeserializerFactory(c));
```

Listing 4.12: Possible implementation of `registerBeanMapping`

named *deployServer* and *deployClient*. Two addional targets have been given to*register/unregister* a service including its non-standard type mappings to the Axis server configuration using the deploy/undeploy descriptors.

Although it is not related with software engineeing in GRID computing in particular, one problem and its solution should be noted. When using ant as a build tool, it is always difficult to decide what to do with required libraries. The project described in this thesis needs eleven jar files in the classpath. They are distributed with different licenses. While one would think that they can be stored in the version control system along the source code, there are a few issues. First, version information of used libraries is lost unless it is coded into the filename of the library (which introduces other problems). Second, there could be a large number of projects in the repository which all have the same libraries stored. This uses more storage than necessary and makes upgrades of libraries much more difficult. Therefore it seems promising to download missing libraries automatically from a website. A few of the used libraries cannot be downloaded easily by an ant task because there are licenses which have to be read and accepted manually. Therefore a local repository for libraries should be used for automatic downloading. This could be a webserver (with password protection to satisfy the paragraphs about redistribution in the licenses). Maven [Apab], a project management framework, which can be considered a super-set of ant, can help solving these issues and also much more but was considered to be too complex for the setting in the Chair of Distributed Systems. The usage of Maven pays off in bigger organizations where repeatable processes need to be standardized.

Regardless of the problems, the libraries have been added to the version control system to have a less elegant but very simple solution.

4.2.8 Limitations of the prototype

For this implementation a prototypical approach has been chosen so that this work can focus on the important software engineering issues. See [Som07] for advantages and characteristics of prototyping in software development projects.

The main characteristics of prototyping are that some topics will not be handled by the final software and an agile process to enable exploration into a limited number of unknown topics. In this work the following topics have been considered not so relevant:

- Useage of custom serializers instead of bean serializers.
- Concurrency
- Keeping data not longer than really necessary in memory.

To improve performance or structure these topics could be tackled. In other words, the software is not designed as a throw-away prototype but rather an evolving one in the spirit of agile development processes [Som07].

As the reader can see in the code excerpts (listing 4.10), JavaBeans are serialized before constructing the SOAP message. This is very easy to implement which is why it has been chosen in this implementation. For better structure the data classes (`ee.ut.math.doug.AssembledMatrix`, `DoubleVector` and `Eigenspace`) should not be beans. Methods that have been added just for the purpose to satisfy the contract for JavaBeans have been marked by a comment so it should be easy to revert them to a better, non-JavaBean class. In this case the `BeanSerializerFactory` is no longer able to serialize the objects. Therefore a custom Serializer has to be implemented. See [Apad] for information how to do it.

4.3 Performance

4.3.1 Size of SOAP message

Different approaches for constructing the SOAP message have been examined. So far, the performance of these approaches has not been considered. Because there is no existing solution to compare the Web Service with, they can only compared to each other. Here the sizes of the SOAP messages are compared since they determine the speed of the system as a whole. This thesis focuses on comparing the XDR/attachment (having `DataHandler` parameters) vs. bean version of `runAssembled` (having `AssembledMatrix` and `DoubleVector` as parameter) because XDR input/output formats are so far available only for the assembled version of DOUG.

The two approaches represent extremes: While the XDR approach is a fast, if not the fastest possible way to do a Web Service call, the SOAP message lacks clarity, expressiveness and flexibility that can help the user to debug the call. The message of the beans version has a maximum of clarity and flexibility but the textual representation of the data as well as additional XML elements produced by Axis increase the size very much.

To make an educated decision how to balance clarity and size of the SOAP messages one needs to have numbers to compare the different approaches. Therefore several examples have been run and the size of the messages have been recorded. This was done as suggested by [YHW05]: The proxy NetTool was used to set up a TCP tunnel and the Web Service call was redirected to a different port. NetTool automatically keeps track of the size of the messages that pass through the tunnel, therefore this is a convenient choice.

File	Unknowns	Sent (bytes)	Recieved (bytes)	Total (bytes)
Lap4x4	9	2,764	1,203	3,967
Lap8x8	49	6,028	1,527	7,555
Lap16x16	225	21,005	2,931	23,936
Lap32x32	961	84,749	8,827	93,576

Table 4.1: Sizes of different SOAP messages using XDR version.

File	Unknowns	Sent (bytes)	Recieved (bytes)	Total (bytes)
Lap4x4	9	26,804	1,395	28,199
Lap8x8	49	155,633	3,555	159,188
Lap16x16	225	753,445	13,033	766,478
Lap32x32	961	3,313,813	52,685	3,366,498

Table 4.2: Sizes of different SOAP messages using beans version.

File	Unknowns	Sent (%)	Recieved (%)	Total (%)
Lap4x4	9	10.3	86.2	14.1
Lap8x8	49	3.9	43.0	4.7
Lap16x16	225	2.8	22.5	3.1
Lap32x32	961	2.6	16.8	2.8

Table 4.3: Comparing sizes of different SOAP messages. (XDR/beans)

The examples use matrix data that was created for testing purposes and belongs to the standard DOUG example set. The right hand side was created by a small program the writes out 64 bit double precision floats between 0 and 1. In tables 4.1 and 4.2 the name of the data set, the number of unknown in the problem, the sizes of the input files and both the sent as well as the received message sizes are shown both for the XDR and the bean solution. In table 4.3 the relative message sizes of the XDR version compared to the bean version are given.

It is obvious that there is some protocol overhead — the header of the SOAP message — which has an impact on the measurements on the smaller examples. The larger the input problems get, the larger the message body gets and the smaller its impact is. Looking at the table with the comparisons it becomes clear, how big the performance gain by the usage of XDR really is. The message sizes using problems of reasonable size are around 35 to 40 times smaller as with beans. In the opinion of the author, the loss of clarity when using XDR is acceptable considering the vastly reduced size of the SOAP messages.

For the sake of completeness it should be remarked that there are different other approaches how to balance the properties of the SOAP message. With message style Web Services the programmer has direct access to the XML rather than using an abstracted access through an API. Automatic mapping of objects to XML is then turn off and the user's responsibility. As an example look at the SOAP message in table 2.1 with contains a `DoubleVector`. With message style Web Services the line `<vector xsi:type="xsd:double"> -1.3248258958028</vector>` could be replaced by `<v="-1.3248258958028"/>` which is considerable smaller. While this is a valid XML element, the problem with this approach is that because no standard for the format of the message is used the client needs special knowledge how to extract the vector from the message. This knowledge would have to be given to client programmers in a textual document which is prone to errors. Automatic usage by other programs would be considerable harder. Therefore, while size and clarity of the SOAP message is very good, interoperability becomes a problem. Since interoperability in GRID environments is a must-have, the approach with XDR attachments is more favorable.

4.3.2 Compressing XML

An additional techique is compression in the protocol (either HTTP or TCP/IP). The most powerful compression can be achieved by using common compression algorithms like zip, gzip or bzip2. Tests conducted for [WBF04] have shown that network traffic can be reduced by 26% by using gzip compression for HTTP requests. Nevertheless, this might be not very good for debugging. The same authors explain that differential encoding for the compression of small files achieves an average size reduction of random text up to 86%. It should be noted that the files used for DOUG might yield bigger possible reductions because the files are larger and because there is usually more redundancy in the matrices. To the knowledge of the author, there are no Web Service frameworks available that implement differential encoding.

One software package that implements a differential encoding compression is

XMill [LS00]. An interesting and rewarding future work would be to make XMill available for Axis or an other Web Service framework.

The W3C has a working group dedicated to the Efficient XML Interchange (EXI) [Eff]. Their work also includes BinaryXML which is supposed to be smaller at the cost of clarity. Their first candidate recommendation is due for July 2007. Adoption into usable frameworks will take longer so it is unclear when the fruits of their work can be used by Web Service programmers.

4.3.3 Memory cosiderations

In case of the operations that do not use attachments the data is kept longer in memory then necessary. The data classes could be modeled in a variation of the Proxy pattern [GHJV95]. The interface of the classes could be maintained but internally they rely on the file stored on the hard disk. In the case of `AssembledMatrix` arithmetic operations have to be stored in private members of the class as well because the file on the disk should not be altered so it could be used again later by the user. This has to be implemented carefully but can reduce memory usage drastically if the files are large.

4.4 Assessment

In this chapter an approach was given to GRID-enable a legacy application using the example of DOUG. A wrapper was used to implement the Web Service and necessary preparations for the execution of the legacy application's binary. The wrapper is independent of the implementation language of the legacy application so any language and Web Service framework can be used to create the wrapper.

It was proven that the naive approach to Web Services, namely object serialization to XML in particular, can render the Web Service unusable due to performance issues — although its benefits. If this is the case, then WS-Attachments should be used to attach files to the operation call. Data files are almost certainly smaller than serialized data objects especially if binary files are used.

In the case of DOUG interoperability was not yet given. With the usage of XDR a binary format has been created which files are very small although not compressed and enable interoperability. This solution will also be beneficial for other legacy applications, if the XDR library is reachable from the implementation language. With Fortran this is possible with FXDR.

Compression of Web Services is still immature and except for gzip not available in the frameworks. Differential encoding needs to be integrated into Web Service Frameworks, because it's compression ratings are higher and it is faster than traditional compression. Soon new impulses can be expected from the W3C

EXI working group. If in future there were major advances in this area, then the conclusion to use a solution with XDR binaries might be relativated by an object serialization which result is reasonable small. Differential encoding and compression would have to be part of that solution, too. For now although, XDR attachments are part of an optimal solution. It should be noted, that it is more beneficial to minimize the message in the first place, than to add time-intensive compression to wasteful communications.

Chapter 5

Extraction of the preconditioner

While the implementation of a Web Service based Eigenvalue solver used the whole DOUG as a component, this project took a different direction. Here the task was to extract one part of DOUG — the preconditioner — and make it available as a component itself. As we will see after carefully analyzing the task, it was realized that two additional components are needed within the preconditioner. In figure 5.1 a component diagram of the resulting components is shown.

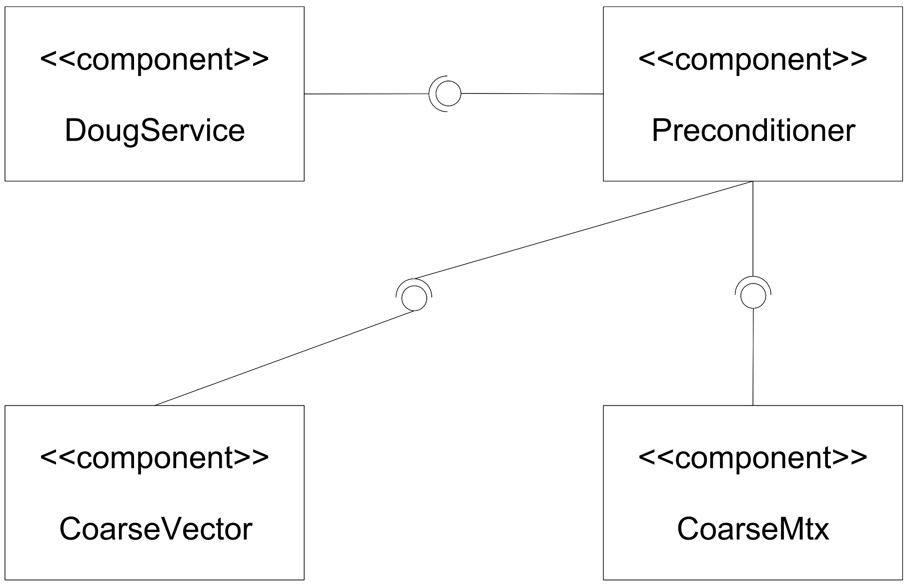


Figure 5.1: Component diagram of the preconditioner

The preconditioner code tends to be more volatile than other code. There is no best way to do a preconditioning since there are conflicting optimization goals.

Therefore, researchers can benefit if they are able to exchange different algorithms fast and test their implementations. The extraction of the DOUG preconditioner will not only benefit the development team to try out new algorithms but it might also lead to other people using this or any other preconditioner component that will be developed. These could be offered permanently on the GRID as service. Then no compilation and configuration is necessary except the ones to find the service.

5.1 Requirements

Different architectural designs have been made which have different implications. The designs were required to fulfill the following requirements:

1. The preconditioner must be a separate component from `DougService`.
2. All necessary data must be passed via a defined interface.
3. The preconditoner component must be able to be used by third parties.
4. `DougService` must still be reasonable fast.[1]
5. The communication between the services must be realized as Web Service communication.
6. The used Web Service framework must be Axis 1.4.
7. The platform as well as the number of CPUs running the services must be flexible.

5.2 First Architecture

The first architecture is depicted in figure 5.2. `DougService` behaves externally exactly as described in chapter 4. Internally, it delegates the calculation of the precondition matrix to the service with the name `Preconditioner`.

The number of nodes on which both components run does not need to be equal. This is due to the nature of the GRID: It is the resource broker's choice which resource is chosen for the job. If the user's job is executed on a specific host, then it is usually reasonable to use all processors that are available to speed up the calculation. In figure 5.2 this is exemplarily shown as `DougService` uses five and the `Preconditioner` uses four nodes.

[1]It was not specified what ”resonable fast” means. Due to the nature of DOUG, an increase in execution time of 10% is probably not worth the flexibility gain anymore.

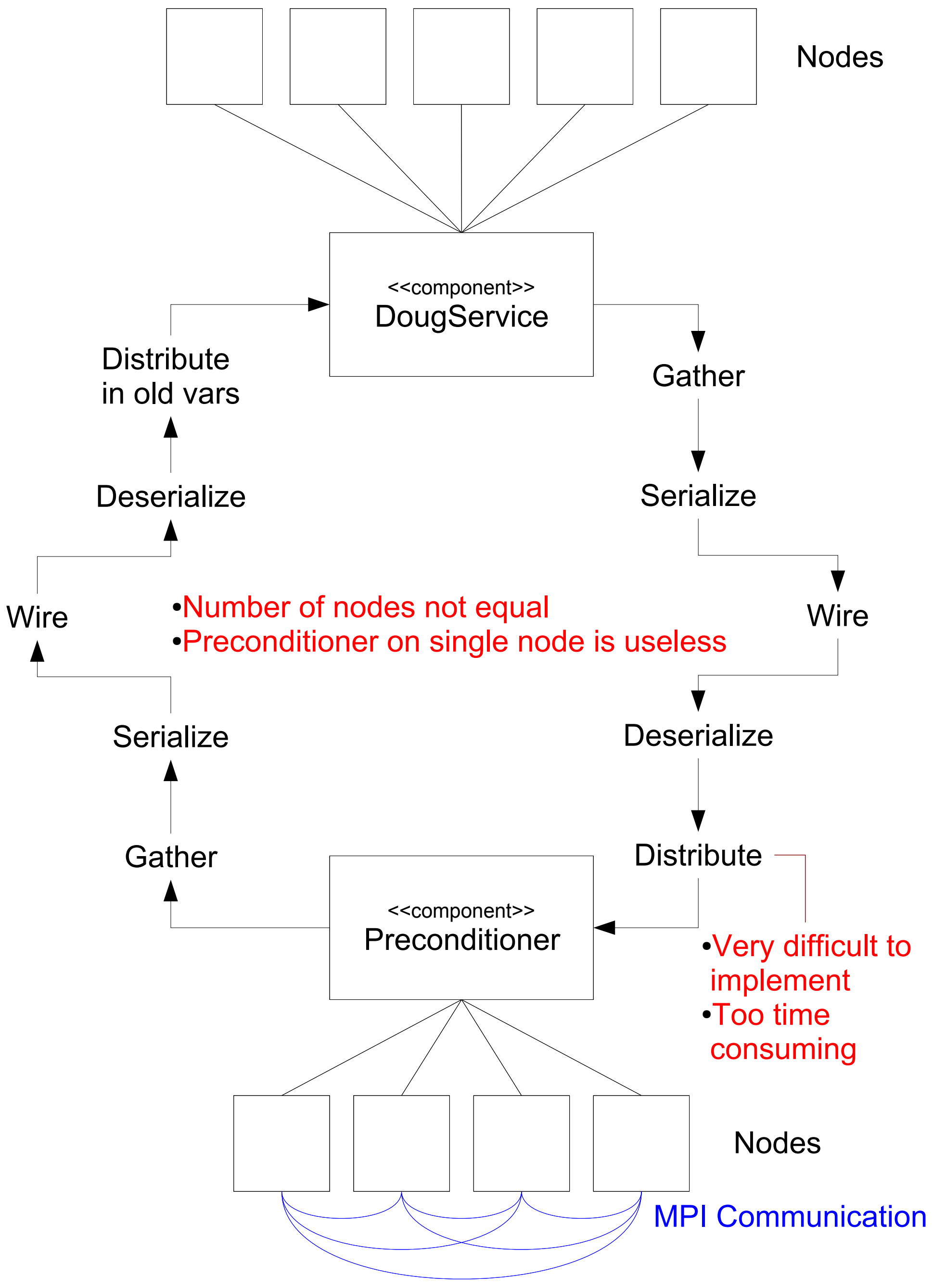


Figure 5.2: The first architecture for the Preconditioner Service.

The communication has to be prepared in several steps. First the data which is distributed in the nodes has to be gathered in the root node using `MPI_Gather` (see figure 2.2). Then it will be serialized to a file. This is necessary because of requirement 6. Data exchange between DOUG which code base is Fortran and the wrapper written in Java is done through a file as in chapter 4.1.3.

Since Axis is a Java framework, it cannot be used to add Web Service client code to `DougService`. There was not enough time for the author to familiarize himself with a different framework so a workaround was developed to overcome this problem. From the Fortran code an executable would be called either blockingly or non-blockingly that would initiate the Web Service call. This executable would read the data of the application which has been stored in a file before, prepare and execute the Web Service call, wait for the result, write the result in a file and finally terminate. The result is then ready for the application to pick up. In the architectural diagram the step Wire includes all necessary code for the thin Web Service client layer. It is obvious that the usage of an executable implementing the Web Service layer is not a good substitute for the use of a native Web Service framework.

The `Preconditioner` service is wrapped similar as the `DougService` in chapter 4. The wrapper receives the SOAP message, writes the data to a file and then starts the Fortran code of the preconditioner. The root process deserializes the data from the file and sets up its data structures. Since there are several nodes which run the preconditioning in parallel, the data has now to be distributed. This is very hard though: The distribution algorithm represents the most important knowledge of DOUG and would have to be applied to this problem again. Distributing the data again would also cost more CPU and communication time than necessary.

Another problem is that the nodes involved with the preconditioner algorithm have to exchange data three times. They have to gather the coarse matrix and a coarse vector in all nodes once. This is done by a variant of the MPI function `MPI_AllGather` (see chapter 2.3). This MPI communication must be possible on the host. With inter-node MPI communication within the component an additional constraint is given to the resource broker namely to pick only hosts where MPI is possible at all and where the network ports used by MPI are open. It would also violate the requirement of a component (see chapter 1.3.1) to be as independent as possible. A possible solution would be to run the preconditioner only sequential that is on only one node. Then there is no need for MPI communication and also the problem of redistribution explained in the previous paragraph would no longer exist. Of course this would contradict the philosophy of DOUG to do as much as possible in parallel and slow down the calculation too much.

When the `Preconditioner` is done, the data has to be gathered again, serialized and can then be sent back to `DougService`. This is done via the return value of the RPC. After the deserialization it has to be distributed back to the right nodes so that `DougService` can resume its work.

Web Services are rather slow (cf. chapter 2.1). On the other hand, Web Services give more flexibility, the ability to split off components and gain a standardized external interface. Therefore, it makes sense to minimize the amount of Web Service communication (size and number of calls) while maximizing its benefits to DOUG. By splitting the system in subsystems in a way so that not much communication is necessary, we can get the most out of componentization with Web Service. A simple estimate to compare the performance implication of different architecture designs is to look into the number of necessary blocking Web Service calls. Non-blocking calls can be ignored if between the call and the next synchronization enough computation is done. If this is the case, one can assume that communication is not the bottleneck.

With this architectural design, a call would be executed *for every iteration*. While this will very likely be notable, it would not lead to seriously worse communication times.

To summarize there are three main problems in this approach. The first is that redistribution of data is necessary because of the different number of nodes that the services use. The second is that the usage of a non-native Web Service framework requires extensive workarounds. The last problem is that the MPI communication between the nodes of the `Preconditioner` does not fit the philosophy of modern GRIDs and componentization. Because of this problems another approach was considered.

5.3 Second Architecture

The second architecture overcomes the cost-intensive redistribution of the data. This is achieved by using the same distribution scheme for both the `DougService` and the `Preconditioner`. The number of nodes of `DougService` matches the number of instances of the `Preconditioner`. The difference is that no gathering is necessary because each of the nodes of `DougService` initiate a Web Service call (via an external executable as before) and pass just their local data to their matching `Preconditioner` service instance.

The architecture diagram in figure 5.3 shows an example with three nodes which make up `DougService`. The communication is done similar to the previous architectural design except that the gather and distribution is no longer necessary and that several — in this case three — communications happen in parallel.

This approach solves the most severe problem of the first architecture but also comes at a price: Because every node makes its own call, the number of Web

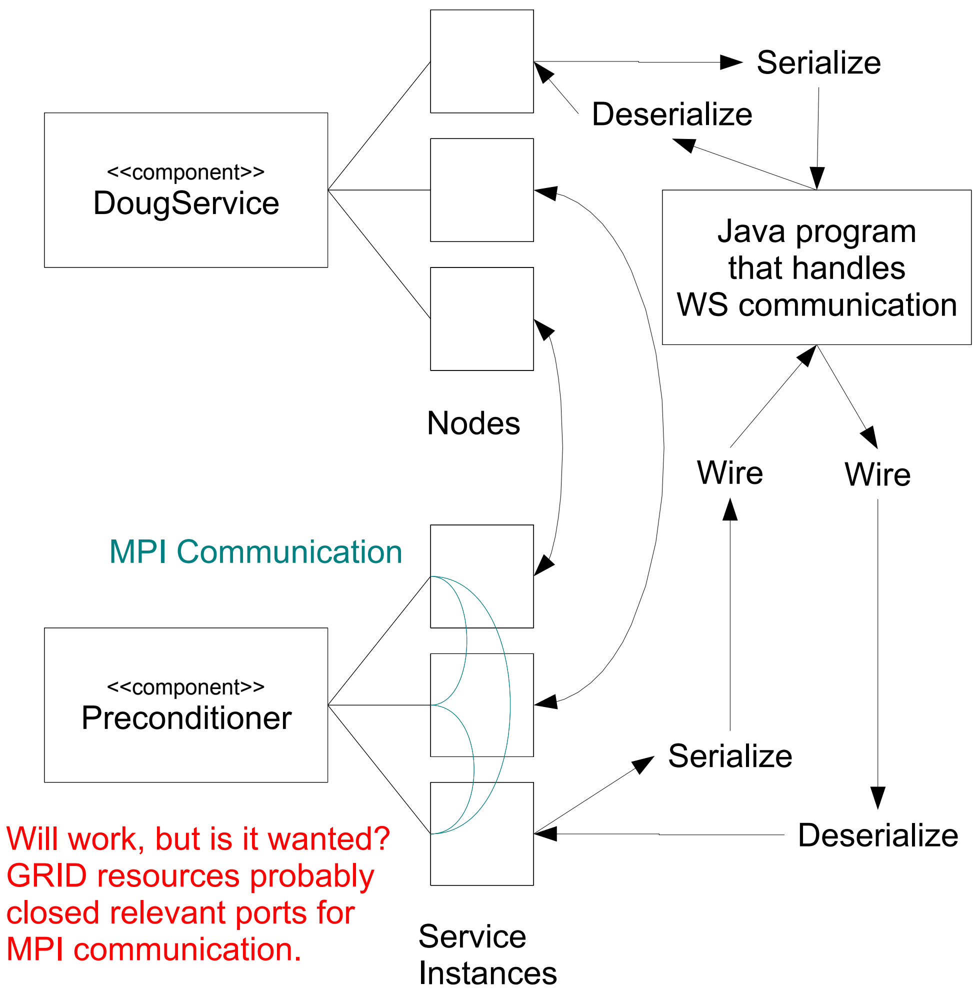

<<component>>
DougService
Serialize
Deserialize
Java program that handles WS communication
Nodes
Wire
Wire
MPI Communication
<<component>>
Preconditioner
Serialize
Deserialize
Will work, but is it wanted? GRID resources probably closed relevant ports for MPI communication.
Service Instances

Figure 5.3: The second architecture for the `Preconditioner` service.

Service calls increases to *number of iterations times number of nodes.* Since the payload of the calls is also shared over the calls for every iteration, it will not effect the communication time very strongly. Nevertheless, the overhead involved will still be greater. Therefore the communication part of this architecture will take more time than the communication part of the previous architecture.

The author believes that this scheme — local data in one node gets passed to one "dedicated" instance of a service — is useful for many parallel programs in GRIDs. If this approach prevails, it could evolve into a best practice.

5.4 Replacing MPI communication

There are still two problems in this architecture. The MPI communication between the instances of the `Preconditioner` is not desirable. In the second architectural design there is no single point anymore where a MPI boot is possible. Booting MPI means to start a runtime environment (e.g. a demon on Linux) and supply it with a list of hosts that are involved in the computation as well as the number of nodes that will be created used on that host. This is necessary so that the nodes can "find" each other. With this architecture, there is no central authority that can boot MPI or give a list of hosts. Actually, there is a central authority, that knows, which host run the `Preconditioner` — the resource broker — but to the author's knowledge there is no standard way to query it.

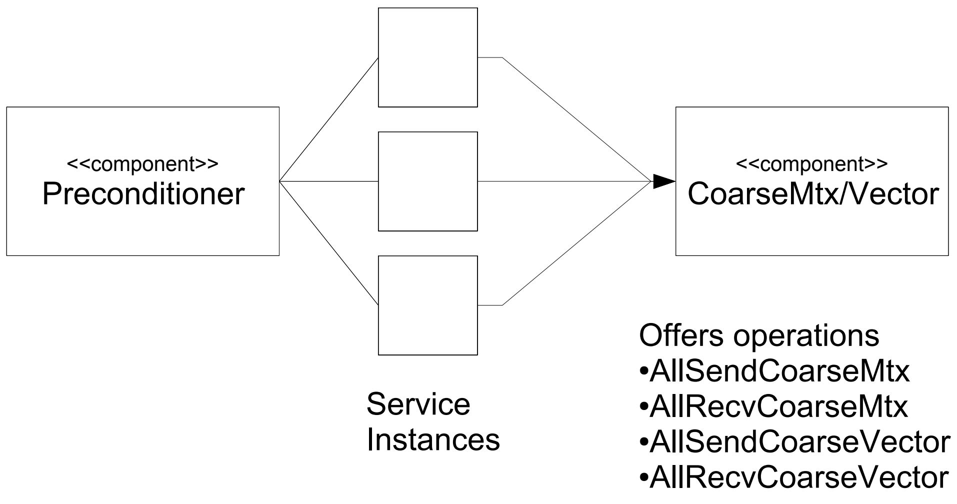


Figure 5.4: Replacing MPI with Web Service by introducing new components.

The easiest way to solve this dilemma is to replace the MPI communication in `Preconditioner` with communication via Web Services. Each run of the `Pre-`

conditioner there are three MPI_AllGather communications.The communication is necessary to exchange the coarse matrix as well as an accompanying coarse vector. One of these MPI calls is blocking, the others two are non-blocking with synchronization points relatively far from the initialization of the call. Note that non-blocking variants of MPI_AllGather are not specified by the MPI-2 specification [For]. It is rather an unofficial extension which opens a new thread and does a normal, blocking MPI_AllGather from this thread. The MPI_AllGather operations would have to be simulated with Web Services. Therefore one or two components have to be created as shown in figure 5.4. These components have the same operation names as the matching Fortran subroutines in the DOUG code. For identification purposes the services are called CoarseMtx respectively CoarseVector resembling naming conventions in the DOUG code, too. Because the operations of the new services have the same external behavior as the MPI-based solution they can are easily replaced and the new code can be compared to the old code in tests.

Axis 1.4 does not support non-blocking calls natively but a call can be easily made non-blocking by using the same mechanism as described for the MPI_AllGather variant: Start a new thread and do the call from there. See listing 5.1 on how to do it in Java.

```java
new Thread() {
    public void run() {
        // do Axis call here
    }
}.start();
```

Listing 5.1: Non-blocking Web Service call

Several runs of DougService might run in parallel on one GRID. Or the components CoarseMtx and CoarseVector might be used by other parties on the GRID. Either way the data will very likely be corrupted, if more than one application uses the components at the same time. Therefore more data storage instances must be used. When an application calls either of the two components some kind of identifier must be passed along the data, that groups the participants of one gather operation. This must be unique in the GRID as long as the operation is executed. For DougService this means that even before it is parallelized in several nodes the identified must be agreed upon. This can for example be done by using a universally unique identifier (UUID) [ITU04]. Implementations of this standard exist for most popular languages.

Analyzing the number of Web Service calls that are needed it is found that it has increased by another factor. Every instance of the Preconditioner in every iteration of `DougService` there are three additional Web Service calls summing up to *number of iterations * number of nodes * 4* while half of those are non-blocking. While it seems like a fitting architecture, after discussions within the chair it was decided that the loss of performance through the introduction of a high number of Web Service calls would be too large. Although it seemed a good idea to benefit from the increased flexibility and community support by the separation of the preconditioner it is nested too deep in the code. The internally occurring MPI communication is an additional obstacle for componentization. Therefore, it was not proceeded further with the separation of the precondtioner. Still, the author believes that componentization of legacy code is an interesting and rewarding topic for GRID users.

If more people need to replace MPI communication with Web Service communication, then it would make sense to create a toolkit with expert knowledge. It is not a trivial task to imitate MPI with Web Services and near-optimal performance is important. A naive ad-hoc solution would not include performance optimizations when even the current implementations of MPI do not get optimal performance [CHPvdG04]. This might be a topic for further research, but first a cost/benefit analysis is needed.

Although the usage of Web Services is clearly dominated by using it for RPC as it is done in this thesis, too, Web Services can be viewed as a messaging technique because they share a common architectural view [Vog03]. Since MPI is also a messaging techique, it shares characteristics with Web Services. A detailed comparison of MPI and Web Services can result in a bridge between those techniques, which would help componentization efforts of other GRID users very much.

Chapter 6

Related work

There has already lots of work been done, which used a wrapper approach to make legacy application available as a Web Service or on service oriented GRIDs.

In their article [KE02] two IBM employees explain an architecture for wrapping a legacy application as Web Service using an adapter. They don't explain how that adapter would be implemented. On the other hand, they identify the problem of concurrency which has been factored out for the described prototype. As a solution they propose the usage of a full fledged transaction, guaranteeing ACID (Atomicity, Consistency, Isolation and Durability) as well as rollbacks. To gain the functionality of transactions they suggest usage of EJB session beans.

Through the creation of a so called Generic Application Serive, [SMZ+05] accessed legacy code by executing binaries as it has been done in this thesis. The legacy application has not been modified. They also use an approach similar to a RPC. Interessting is, that they decided to use a different application description document, than WSDL. It was not elaborated, why WSDL was not chosen, but their the description document is still based on standards like XML and XML-Schema. An interface to the Generic Application Service allows the same operations on all wrapped legacy applications. Their approach has only been tested on examples that do not need big amounts of data so that the problems described in this thesis did not occur. Also interoperability was not identified as a problem.

The Grid Execution Management for Legacy Code Architecture (GEMCLA) [GKT+05] is another tool to deploy legacy code on the GRID. It is integrated into the GRID portal P-GRADE [KDK+03] and based on Globus Toolkit 3. Here the legacy application's code is not changed as well, just a Legacy Code Interface Description (LCID) has to be created which includes the interface to the application.

A step towards componentization was done by [HIWD03] using the Java-C Automatic Wrapper (JACAW) and the Mediation of Data and Legacy Code Interface tool (MEDLI). It wraps code that is available as libraries. It can only wrap C libraries and monolithic legacy applications like DOUG do not benefit from it. It offers no support to split off components. When using JACAW wrapped components, they can be used in a visual programming language called Triana to model workflow. It is planned to extend JACAW to make wrapped libraries available as a Web Service to use it on a GRID but it has not been evaluated so far if libraries have similar properties as components.

These solutions focus on more or less automatic transformation of legacy applications into components. While this approach is good for some applications, other applications do not fulfill the degree of independence required for components. Those would have to be refactored or would have to gain interoperability through a wrapper that also handles data conversion. Also, none of those approaches gives a solution for large data transfers.

Reworking the architecture of a legacy application to a component based one has not been covered much yet. The authors of [PMB$^+$06] presented a case study where a Java legacy application using the distributed object middleware ProActive was componentized and made available on the GRID. They present a software engineering process how legacy applications can be componentized. Since obejct oriented languages encourage modular programming more than non object oriented languages (like Fortran) and since an distributed object framework was already used, their effort is not directly comparable to this work. Distributed object calls are natural cut points for componentization, while in shared memory environments like MPI the communication should be contained in the components.

While in this work Web Services where used to form components and the SOAP messages in RPC style, Web Services could also be understood as a messaging technique [Vog03]. The authors of [PTL04b] did tests with small applications that used first MPI and then Web Service to compare the performance. While they believe that replacing MPI with Web Services is not generally beneficial, they recognize that there are some cases when it can be useful. In this thesis, this is the case. The author of this thesis agrees that researching the similarities between MPI and Web Services is needed.

Chapter 7

Conclusion and outlook

Using DOUG as a case study, the preparation of a legacy application for modern GRIDs has been examined. Two complete different approaches have been considered: The first approach is the wrapping of the whole application as a component for service oriented GRIDs and the second is the componentization of the application into several parts.

In the first part, Web Services have been used to make the component. A thin wrapper which handles the communication was created. External use of the component has been shown by the creation of another component, an Eigenvalue solver, which can be used by other components as well. This has been proven by the implementation of a graphical user interface.

A focus has been placed on the exchanged data. Three different kinds of messages have been examined. The data formats used by DOUG before did not fulfill the requirement of interoperability over different platforms — an imperative for GRID environments. Therefore, a new data format has been implemented based on binaries written by the XDR library. Those files were sent within the SOAP message using WS-Attachments. Usage of those in external applications is not very convenient because each application has to parse the data by its own. Since XDR is standard-aware, this is not a big problem. The alternative to use objects that are serialized to XML according to JAX-RPC is more elegant for external use. Unfortunately, the huge size of the resulting message forbids the usage of it in a production environment.

Handling of the new data format based on XDR has been implemented into the legacy application itself and not in the wrapper so that the users of the application will benefit from the gained interoperability even without the usage of the Web Service this way. The solution describes a process which is also applicable for other legacy applications that should be made available on the GRID. Starting with modeling the components with UML diagrams, then focusing on the interfaces and the data exchanges, the programmer should take a close look if the messages are small enough and interoperable. If the messages are too large, WS-Attachments with a standardized data format, e.g. XDR, are a very good choice. The SOAP messages can be profiled and debugged with a proxy like TCPMonitor or NetTool.

Some changes have been suggested or implemented for DOUG itself, too. For example, it has been shown how refactoring, a technique made for object oriented systems, can also be used with the Fortran code. A pattern called Fake Polymorphism has been identified and applied to the code prior to adding the new data format. This improved the structure of the source code drastically.

In the second major part of the work, it was researched how DOUG could be componentized. The target was the preconditioner a part of DOUG which is volatile and therefore would benefit if it was a component. It was found that applying a different architecture to a legacy system is hard especially if it uses a non-object oriented implementation language and distributed memory. If the data was gathered in one node before it was sent to a different component this would equal a synchronization barrier. Even worse, the data would have to be redistributed to a potentially different number of processors again which is not feasible for DOUG. Therefore, it makes sense to let each node make a call to a preconditioner service instance by itself without prior data gathering.

A component written in a specific language that is calling another component and therefore acts as a consumer can not use the wrapper approach anymore. It must use a Web Service framework in its implementation language or the design will suffer severely.

An interesting problem is the remaining MPI communication between the preconditioner component instances. For increased independence, the MPI communication must be replaced with Web Service communication. This results in more components to act as a helper to gather data. MPI and Web Services share surprising similar characteristics.

Several topics have been identified that require further work. Although DOUG was prepared as a Web Service for this thesis and comparisons between different styles of messages have been done, the XDR version of the operations is yet only available for assembled input. Using the implementation for assembled input as an example, also elemental input should be migrated to XDR binaries. The work for that can be done along the lines of the previous implementation:

- Build test cases for the old data format using the testing framework.

- Implement a Fortran program to convert the old style, unformatted binary format to XDR binary format for all six different file types.

- Refactor reader and writer code within DOUG. Extract methods and/or use the Fake Polymorphism pattern to improve the structure of the code.

- Add necessary configuration parameters to the control file to switch between XDR and old style formats.
- Add reader and writer code for the XDR format.
- Run the prior made tests with converted input files and check the solutions for equality, for example with a hash function like SHA1.
- Add the necessary method signatures to `I_DougService`.
- Add the necessary service implementation to `DougService`.
- Build a simple client to test the service.

After that, the Eigenvalue Solver could be rewritten to use the XDR Web Services as a base. Since the scientific gain of this is limited, it should only be done if it is used regularly and considered to be too slow.

Also within DOUG an automatic recognition of the input should be done to simplify the interface of `DougService` and improve the usability of standalone DOUG. The user should not need to configure if old Fortran binary, XDR binary or formatted text is used as input. DOUG could also distinguish between elemental or assembled input automatically thus reducing the interface of the Web Service.

At least in the open-source community are surprisingly few products for tool supported refactoring of non-object oriented languages. The author hopes that the Eclipse IDE will serve as an incubator for development environments and refactoring tool suits for non-object oriented languages. For Fortran in particular the work around Photran [OXJF05] is very promising and should soon be ready for production use. Design as well as refactoring patterns should be established to be used in modern IDEs.

Another very surprising fact is that to the knowledge of the author there is no Web Service framework that includes useful compression of the SOAP message. GZip compression exists for Axis but its characteristics make it not useful for compressing SOAP messages. Integration of differential encoding into Web Service frameworks is in the opinion of the author a very rewarding task. The Axis community would appreciate the integration of a software like XMill [LS00] into the framework. This will not be very difficult, because Axis is already prepared for the integration of modules.

Not much work has been done in coponentizing legacy software for the GRID although this is an important topic. The authors of [PMB$^+$06] did research using already object oriented applications, but for non-object oriented programs — which are very important, considering the typical user of GRIDs — there is to the knowledge of the author no research at all, except the architecture design in this thesis. The research in this area should be intensified. It should also take into account distributed memory techniques commonly used in high performance computing. The findings of this thesis can be a starting point for future work.

Considering the characteristics that Web Services have in common with messaging techniques, it is an interesting task to compare MPI with Web Services on a functional level. Existing MPI applications can be used to replace MPI communication with SOAP messages. Starting with simple one-to-one communication and then moving up to group communication similarities and problems can be identified. In the end it might be possible to create a bridge between MPI and Web Services which would make componentization of MPI applications easier as well as help preparation for a deploy on modern GRIDs.

The approach presented in this thesis is different than in other publications. DOUG has been transformed into a component which can be run on the GRID not only by wrapping it, but also by changing its properties before. Previous approaches would have led to a service that lacks independence and is therefore not suited for the GRID. A repeatable process has been developed to apply the solution on other legacy software. Also, a first step to legacy application componentization for the case of distributed memory architectures with non object oriented implementation languages has been presented, a field which has not been covered before. Therefore, this work is a significant step to migrate applications from old GRIDs to modern, service oriented ones.

Bibliography

[Apaa] Apache Software Foundation. Apache Ant Homepage. http://ant.apache.org/ (25.04.2007).

[Apab] Apache Software Foundation. Apache Maven Homepage. http://maven.apache.org/ (25.04.2007).

[Apac] Apache Software Foundation. Axis Homepage. http://ws.apache.org/axis/ (25.04.2007).

[Apad] Apache Software Foundation. Axis User's Guide. http://ws.apache.org/axis/java/user-guide.html (25.04.2007).

[Apae] Apache Software Foundation. Axis2 Homepage. http://ws.apache.org/axis2/ (25.04.2007).

[Apaf] Apache Software Foundation. Jakarta Commons HTTPClient. http://jakarta.apache.org/commons/httpclient/ (25.04.2007).

[Ars04] Ali Arsanjani. Service-oriented modeling and architecture. http://www-128.ibm.com/developerworks/webservices/library/ws-soa-design1/ (25.04.2007), November 2004.

[Bat] Oleg Batrashev. Instructions for the DOUG testing framework. http://www.dougdevel.org/dougwiki/index.php/Testing (24.04.2007).

[BDV94] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[Bel04] Donald Bell. UML basics: The component diagram. http://www-128.ibm.com/developerworks/rational/library/dec04/bell/ (25.04.2007), December 2004.

[Ber96] Philip A. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996.

[Bun] Julian J. Bunn. Floppy. `http://www.netlib.org/floppy/` (25.04.2007).

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl` (25.04.2007), March 2001.

[CFF$^+$04] Karl Czajkowski, Don Ferguson, Ian Foster, Jeff Frey, Steve Graham, Tom Maguire, David Snelling, and Steve Tuecke. *From OGSI to WS-Resource Framework: Refactoring and Evolution, Version 1.1*, May 2004.

[CH01] W.T. Councill and G.T. Heinemann. *Component Based Software Engineering — Putting the pieces together*, chapter 1. Addison Wesley, Boston, 2001.

[Cha96] D. Chappell. *Understanding ActiveX and OLE: a guide for developers and managers*. Microsoft Press, Redmond, WA, USA, 1996.

[CHPvdG04] E. W. Chan, M. F. Heimlich, A. Purkayastha, and R. A. van de Geijn. On optimizing collective communication. In *Proceedings of the International Conference on Cluster Computing*, pages 145 – 155. IEEE, September 2004.

[Coh03] Bram Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, 2003.

[De04] Vaishali De. A foundation for refactoring Fortran 90 in Eclipse. Master's thesis, University of Illinois, Urbana-Champaign, 2004.

[Dub05] Manek Dubash. Moore's Law is dead, says Gordon Moore. `http://www.techworld.com/opsys/news/index.cfm?NewsID=3477` (25.04.2007), April 2005.

[Eff] Efficient XML Interchange Working Group. EXI Homepage. `http://www.w3.org/XML/EXI/` (25.04.2007).

[ES01] Dietmar W. Erwin and David F. Snelling. Unicore: A grid computing environment. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 825–834, London, UK, 2001. Springer.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000.

[For] MPI Forum. MPI specifications. `http://www.mpi-forum.org/docs/docs.html` (25.04.2007).

[Fos05] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. In Hai Jin, Daniel Reed, and Wenbin Jiang, editors, *IFIP International Conference on Network and Parallel Computing*, number 3779 in Lecture Notes in Computer Science, pages 2–13. Springer, December 2005.

[Fow99a] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[Fow99b] Martin Fowler. *Refactoring: Improving the Design of Existing Code*, page 76. Addison Wesley, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GKT$^+$05] A. Goyeneche, T. Kiss, G. Terstyanszky, G. Kecskemeti, T.Delaitre, P.Kacsuk, and S.C. Winter. Experiences with deploying legacy code applications as grid services using gemlca. In P.M.A. Sloot, A.G. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, editors, *Advances in Grid Computing - EGC 2005*, volume 3470/2005 of *Lecture Notes in Computer Science*, pages 851–860, Berlin/Heidelberg, July 2005. Springer.

[Gra97] Graham Hamilton (Editor). JavaBeans Version 1.01-A. Technical report, Sun Microsystems, August 1997.

[Gu00] M. Gu. Single- and multiple-vector iterations. In Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst, editors, *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide.* SIAM, Philadelphia, 2000.

[HIWD03] Yan Huang, IanTaylor, David W. Walker, and Robert Davies. Wrapping legacy codes for grid-based applications. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03).* IEEE, April 2003.

[ITU04] ITU-T Study Group 17. Information technology – Open Systems Interconnection – Procedures for the operation of OSI Registration Authorities: Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 object identifier components. Technical Report X.667, International Telecommunication Union, September 2004.

[Kar] Mai-Liis Karring. Performance analysis tools and methods for MPI programs (working title). master thesis in progress.

[KB05] Y. Kulbak and D. Bickson. The emule protocol specification. Technical Report TR-2005-03, Hebrew University of Jerusalem, 2005.

[KDK$^+$03] P. Kacsuk, G. Dózsa, J. Kovács, R. Lovas, N. Podhorszki, Z. Balaton, and G. Gombás. P-grade: A grid programming environment. *Journal of Grid Computing*, 1(2):171–197, June 2003.

[KE02] Dietmar Kübler and Wolfgang Eibach. Adapting legacy applications as Web services. `http://www-128.ibm.com/developerworks/library/ws-legacy/` (27.04.2007), Januar 2002.

[Ker04] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, August 2004.

[KNN03] Michael Kirchhof, Manfred Nagl, and Ulrich Norbisrath. EAI heißt insbesondere Integration: Probleme und die Rolle technischer Hilfsmittel (Enterprise Application Integration is in Particular Integration: Problems and the Role of Technical Utilities). In *Proceedings of ONLINE 2003*. Online Verlag, 2003.

[LCG05] LCG TDR Editorial Board. LHC Computing Grid - technical design report version 1.04. Technical report, CERN, June 2005.

[LS00] Hartmut Liefke and Dan Suciu. Xmill: an efficient compressor for xml data. *SIGMOD Rec.*, 29(2):153–164, 2000.

[Met96] Michael Metcalf. A program to convert FORTRAN 77 source form to Fortran 90 source form. `ftp://ftp.numerical.rl.ac.uk/pub/MandR/convert.f90` (25.04.2007), December 1996.

[Nat] National Center for Supercomputing Applications at the University of Illinois (NCSA) and University of New Mexico (UNM) and Boston University (BU) and University of Kentucky (UKy) and Ohio Supercomputing Center (OSC). Introduction to MPI. `http://webct.ncsa.uiuc.edu:8900/public/MPI/` (25.04.2007).

[OAS07] OASIS Web Services Notification (WSN) TC. Web services notification (WSN) specifiations. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn#technical` (25.04.2007), March 2007.

[O'T] Neil O'Toole. NetTool Homepage. `http://nettool.sourceforge.net/` (25.04.2007).

[OXJF05] Jeffrey Overbey, Spiros Xanthos, Ralph Johnson, and Brian Foote. Refactorings for Fortran and high-performance computing. In *SE-HPCS '05: Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 37–39, New York, NY, USA, 2005. ACM Press.

[Pie] David W. Pierce. FXDR Homepage. `http://meteora.ucsd.edu/~pierce/fxdr_home_page.html` (25.04.2007).

[PL03] Randall Perrey and Mark Lycett. Service-oriented architecture. In *Symposium on Applications and the Internet Workshops*, pages 116–119. IEEE, Januar 2003.

[PMB$^+$06] Nikos Parlavantzas, Matthieu Morel, Françoise Baude, Fabrice Huet, Denis Caromel, and Vladimir Getov. Componentising a scientific application for the grid. Technical Report 31, CoreGRID, Institute on Grid Systems, Tools and Environments, April 2006.

[PTL04a] D. Puppin, N. Tonellotto, and D. Laforenza. Using web services to run distributed numerical applications. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241/2004 of *Lecture Notes in Computer Science*, pages 207–214, Berlin/Heidelberg, September 2004. Springer.

[PTL04b] Diego Puppin, Nicola Tonellotto, and Domenico Laforenza. *Using Web Services to Run Distributed Numerical Applications*, volume 3241/2004 of *Lecture Notes in Computer Science*, pages 207–214. Springer, Berlin/Heidelberg, November 2004.

[Ras00] Jef Raskin. *The Humane Interface: New Directions for Designing Interactive Systems.* Addison-Wesley, 1st edition, March 2000.

[Rip02] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proceedings of the First International Conference on Peer-to-Peer Computing (P2P'01)*, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[Rob03] Roberto Chinnici et al. Java API for XML-based RPC — JAX-RPC1.1. Technical Report JSR-101, Java Community Process, October 2003.

[Saa96] Yousef Saad. *Iterative methods for sparse linear systems.* PWS, 1st edition, 1996.

[SBG04] B.F. Smith, P.E. Bjorstad, and W.D. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, 2004.

[SC06] Borja Sotomayor and Lisa Childers. *Globus Toolkit 4: Programming Java Services.* Morgan Kaufmann, 2006.

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming.* Addison Wesley, Harlow, 2nd edition, 2002.

[SKM+02] David Stainforth, Jamie Kettleborough, Andrew Martin, Andrew Simpson, Richard Gillis, Ali Akkas, Richard Gault, Mat Collins, David Gavaghan, and Myles Allen. Climateprediction.net: Design principles for public-resource modeling research. In S. G. Akl and T. Gonzalez, editors, *Proceedings of the 14th IASTED International Conference on parallel and distributed computing systems*, pages 32–38, Camebridge, USA, November 2002.

[SMZ+05] Vivekananthan Sanjeepan, Andréa Matsunaga, Liping Zhu, Herman Lam, and José A.B. Fortes. A service-oriented, scalable approach to grid-enabling of legacy scientific applications. In *Proceedings of the IEEE International Conference on Web Services (ICWS'05).* IEEE, July 2005.

[Som07] Ian Sommerville. *Software Engineering.* Pearson Education, Harlow, 8th edition, 2007.

[SV06a] R. Scheichl and E. Vainikko. Additive schwarz and aggregation-based coarsening for elliptic problems with highly variable coefficients. Bath Institute For Complex Systems Preprint 9/06, 2006.

[SV06b] R. Scheichl and E. Vainikko. Robust aggregation-based coarsening for additive schwarz in the case of highly variable coefficients. In P. Wesseling, E. ONate, and J. Periaux, editors, *Proceddings of the European Conference on Computational Fluid Dynamics, ECCOMAS CFD 2006*, TU Delft, 2006.

[VB] Anke Visser and Oleg Batrashev. Fortran Extension for Doxygen. `http://dougdevel.org/index.php?page=doxygen` (25.04.2007).

[Vog03] W. Vogels. Web services are not distributed objects. *IEEE Internet Computing*, 7(6):59–66, December 2003.

[W3C06] W3C Web Services Addressing Working Group. Web Services Addressing 1.0 - Core. `http://www.w3.org/TR/2006/REC-ws-addr-core-20060509/` (25.04.2007), May 2006.

[WBF04] C. Werner, C. Buschmann, and S. Fischer. Compressing SOAP messages by using differential encoding. In *Proceedings of the IEEE International Conference on Web Services*, pages 540–547. IEEE, July 2004.

[Wik05] Wikipedia user Kku. Virtual organisation diagram. `http://en.wikipedia.org/wiki/Image:VirtOrg.png` (25.04.2007), August 2005.

[YHW05] Ying Ying, Yan Huang, and David W. Walker. A performance evaluation of using SOAP with attachments for e-science. In *Proceedings of the UK e-Science Meeting*, September 2005.

List of Figures

Listings