Monitoring of Web Services Using UNICORE 6 as an Example
Die vorliegende Arbeit wurde in Zusammenarbeit mit der Forschungszentrum Jülich GmbH, im Jülich Supercomputing Centre (JSC), angefertigt.

Diese Masterarbeit wurde betreut von:

Referent: Prof. Dr. Rer. Nat. M. Reißel
Korreferent: Dipl. Inform. M. Romberg

Die Arbeit ist selbstständig angefertigt und verfasst worden. Keine anderen als die angegebenen Quellen und Hilfsmittel sind verwendet worden.

Jülich, Juli 2009
# Content

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Web Services</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Why Web Services?</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Service Oriented Architecture</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Business Process Execution Language</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Web Service Discovery Service</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>Web Service Resource Framework</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>Web Service Addressing</td>
<td>10</td>
</tr>
<tr>
<td>2.6</td>
<td>Web Service Description Language</td>
<td>10</td>
</tr>
<tr>
<td>2.7</td>
<td>Simple Object Access Protocol</td>
<td>12</td>
</tr>
<tr>
<td>2.8</td>
<td>HTTP SOAP binding</td>
<td>14</td>
</tr>
<tr>
<td>2.9</td>
<td>Extensible Markup Language</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>Existing Grid Monitors</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>System Level Grid Monitoring</td>
<td>17</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Ganglia</td>
<td>17</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Hawkeye</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>User Level Grid Monitoring</td>
<td>19</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Monitoring and Discovery Service</td>
<td>19</td>
</tr>
<tr>
<td>3.2.2</td>
<td>INCA</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>Combined monitors</td>
<td>21</td>
</tr>
<tr>
<td>3.3.1</td>
<td>LHC Era Monitoring</td>
<td>21</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Nagios</td>
<td>22</td>
</tr>
<tr>
<td>3.4</td>
<td>Commercial Business Process Monitors</td>
<td>24</td>
</tr>
<tr>
<td>3.4.1</td>
<td>IBM WebSphere Business Monitor</td>
<td>25</td>
</tr>
<tr>
<td>3.4.2</td>
<td>SEQUENCE</td>
<td>25</td>
</tr>
<tr>
<td>3.5</td>
<td>Assessment of Monitoring Systems</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>Design of a Web Service Monitor</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Grid Monitoring Architecture</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>Design Decisions</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>Requirements</td>
<td>33</td>
</tr>
</tbody>
</table>
5 Monitoring UNICORE 6 ................................................................. 35
5.1 UNICORE 6 .............................................................................. 35
5.2 SIMON 6 .................................................................................. 38
6 Summary ...................................................................................... 49
Appendix .......................................................................................... 51
Acronyms ......................................................................................... 53
List of Figures .................................................................................. 55
List of Tables .................................................................................... 56
References ......................................................................................... 57
1 Introduction

Web services have become more and more important in various computation related areas. Nowadays for example no weather online portal is possible without Web services anymore. These gather all the needed information from a lot of sensors and other measuring instruments about actual temperature, wind velocity, and rainfall to bring the weather forecast to the web page and so to the users.

Even if various online weather portals are available their approach is always the same: first the relevant Web service has to be detected which provides the weather data for the requested area. This will be done with the Universal Description, Discovery and Integration (UDDI) service [1], which plays a central role in dynamic Web services as a standardized directory service. UDDI is one component of the Service Oriented Architecture (SOA) [2] that represents a business related IT architecture approach.

Assuming that every Web service description differs from the other, the weather portal will be faced with different interfaces, once respective Web services have been found. Therefore the communication between the online portal and the Web service is based on the Web Service Description Language (WSDL) [3] and the Simple Object Access Protocol (SOAP) [4] to get the needed abstraction layer. These concepts support Web services’ basic principles which are flexibility and abstraction.

Web services can be categorized into two core application areas that are business and science. While in business use cases the classical Web services often are the first choice scientific applications sometimes require special enhancements. In this context the term of Grid services is often used. In contrast to the classical Web service association as a stateless service, a Grid service is used to control distributed and stateful resources.

Grid services are closely related to the modern vision of Grid computing, which aims to give users access to distributed data, applications and computing power. Nowadays also most Grid middleware, which provides access to the Grid, bases on Grid services or Web services respectively. There are many different definitions of Grid computing and various implementations of the so called Grid middleware as the interface between the Grid and the user.

The thesis concentrates on Grid middleware building on Grid services which are designed to access supercomputer resources. Supercomputers are deployed in many areas of commercial, but especially of scientific use to deal with very complex problems, which need to be computed in parallel to get the results in a reasonable time. To ease their work, scientists often use some Grid middleware like UNICORE [5] or Globus [6]. Integrated solutions are required which allow creating, submitting and monitoring their computing jobs.

Independently from business or scientific use of Web services, so called Business Processes play an essential role in both areas. With Business Processes, Workflows can be designed which are predefined sequences of processes. Naturally Business Processes are described with the Business Process Execution Language (BPEL) [7] that is an Extensible Markup Language (XML) [8] based language. With BPEL cycles of Business Processes can be set up, where each single activity is implemented through Web services.
In both business and scientific use cases each provider will have a massive interest that the Web or Grid services are really running and working as supposed. Failing Web services could mean a financial loss if customers turn towards competitors to get the requested functionality and for instance look for other weather portals.

Even if there may not be direct financial loss, when Web services fail in the scientific area, it could lead to bad credits for the supercomputer centre. Scientists are customers of the centre and they will look for other centres to achieve their results if the requested services are not reliable.

In a production environment, a 24 hours a day and 7 days a week availability of the underlying Grid infrastructure is essential and expected by the users. This is an ideal scenario of course because the health status of the Grid is dependent on various factors. All, broken hardware, processes that do not behave as expected, or network problems are risks that might disturb the Grid middleware functionality and so might danger the jobs submitted by the scientists. Even if it will not be possible to reach the 24/7 availability of all components, administrators are anxious to minimize the downtime of the provided services. The persons in charge have to be aware of problems before the user does so that one can react and fix the failure preferably in time. Therefore, it is essential to have automatic procedures to check the health status of the Grid middleware. In case of failures and problems the administrator is notified directly and can initiate the required steps. Those automatic procedures will get complex already if only one supercomputer is involved but one can imagine that it will get more and more complex in case of many centres with more than one supercomputer. Considering that each site runs UNICORE or other Grid middleware and all installations are connected, the need to have automatic monitoring tools even is a must.

Consequently the monitoring of Web respectively Grid services is the main topic of this thesis. Amongst others the thesis deals with the status analysis of existing Web service monitors. Up to now there is no monitor for the UNICORE 6 health status available. Therefore, a central approach is to design this UNICORE 6 monitor considering the results from the analysis of existing tools. Afterwards this design will lead to a concrete implementation.

At first an introduction into Web services is given in chapter 2. Amongst others, WSDL, SOA, SOAP, UDDI, Business Processes, Workflows, and BPEL are explained in detail.

Chapter 3 then analyses already available Grid monitors and classifies them. Basing on the results, a decision is deduced, which approach is considered as suitable to monitor Web services and to display the monitoring data.

Web service monitoring tools have to fulfil various requirements. In chapter 4 first a design study for a Web service monitor is performed. Afterwards the requirements are listed and described which lead to the concrete concept for the Web services monitor.

Chapter 5 provides a proof of concept implemented for the example of UNICORE 6. First an introduction into UNICORE 6 is given. Afterwards SIMON 6 is introduced as the services monitor for UNICORE 6.

Finally, chapter 6 summarizes the work. It describes the status quo and gives a perspective to possible further developments.
2 Web Services

The Web Service Architecture Group of the World Wide Web Consortium (W3C) [9] is leading the definition of Web service technology. W3C defines Web services as follows [10]:

“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.”

Following this definition, Web services are characterised by three main properties: A Web service is a

- software system or simply said a computer program that is
- interoperable with other distributed computer programs and that is designed for communication over a
- network. Thereby network stands for any kind of Inter-, or Intranet.

Reaching those properties, a couple of techniques are necessary, for instance Web Service Description Language (WSDL), Simple Object Access Protocol (SOAP), Hyper Text Transfer Protocol (HTTP) [33], and Extended Markup Language (XML), which will be described detailed in the next sections.

2.1 Why Web Services?

Even if the name Web service could lead to the assumption that this is something, only related to the HTML-based World Wide Web, as one realisation of the Internet [11], this is not the case at all. The above definition shows the universality of Web services, and the World Wide Web is just one application area out of many as shown later on.

At first glance, Web services could be seen as just another distributed computer technology [12] like CORBA [13], or Microsoft’s DCOM [14]. Both the Common Object Request Broker Architecture (CORBA) and the Distributed extension of the Component Object Model (DCOM) represent earlier developments in distributed computing. CORBA was developed in the early 1990s by the Object Management Group (OMG) [15] while the Microsoft approach followed in 1995. Many of CORBA’s and DCOM’s design principles have been reused by the Web service community. So, what was the reason that those distributive technologies have not succeeded? And why has there been the necessity to redesign already existing solutions? Dan Gisolfi who is an IT-architect at IBM and who has been strongly engaged in the Web service development from the beginning, finds a simple answer to both questions. In one of his online articles he [16] states that even if CORBA’s and DCOM’s concepts are reasonable in principle it emerged that the vendors behind those concepts were not really interested in interoperability at that time, although they officially had the intention. But in reality the vendors
Web Services

wanted to sell their own solutions on both parts of the distributive computing application: requestor and provider. This attitude changed a few years ago when the need and advantages of real interoperability was recognised. The vendors realised that they even could increase their e-Commerce if working with unified, simple, and widely accepted standards which are available for free. This was the turning point to leave the tightly coupled, language oriented, existing distributed computer technologies, like CORBA and DCOM, towards the loosely coupled message oriented Web services. Using few and simple widely accepted standards, results from lessons learned from the World Wide Web development. The World Wide Web success story bases essentially on using those standards. As a result, Web services are very flexible and deployable almost anywhere in a network.

Before characterising the main Web service properties, Figure 1 shows how a typical Web service is invoked using the example of the online weather forecast, which has been introduced in chapter 1. A Web service always is a client-server application. The client which is not a human being but a computer program needs the weather information of a city. Initially, the client does not know about which Web service to invoke and where to find it. Therefore, the first step is to contact a discovery service to discovers the Web services. The discovery service is described in detail in chapter 2.3. It returns the location of a Web service with the required functionality to the client.

![Figure 1: A typical Web service invocation](image)

Although the client now knows about the location of the Web service, it still does not know how to invoke the service. So, the client submits a request to the Web service and asks how to invoke the service. Because Web services are designed to be self-describing, everything the client has to know to invoke the service successfully, is returned in a so called Web Service Description Language (WSDL) response (see chapter 2.6). The client is able to interpret the WSDL response and therefore knows about the required data types, data, and message passing protocol the Web service requires. The client implementation usually is completely independent from the Web service implementation. So the
communication between both has to be formulated that abstract that both communication partners will understand each other. The SOAP protocol, which is described in detail in section 2.7, provides such an abstraction layer. The client submits a SOAP request to the Web service. The request bases on the interface description, contained in the Web service’s WSDL response and asks for the actual weather, by providing the city name or a postcode. The Web service answers with a SOAP response which contains the weather information the client requested.

The Web service invocation can be divided into four core layers specifying all involved protocols and abstraction languages:

- **Service Processes Layer:** aggregates and manages the Web services to be discovered by distributed clients.
- **Service Description Layer:** describes how to invoke the Web service. The message passing protocol and the interface description are formulated usually with WSDL which also contains the supported operations. Other languages are possible as well, but WSDL has become the first choice.
- **Service Messages Layer:** Web services only can be invoked by an abstract client-server communication with a protocol both communication partners understand. Usually, the SOAP protocol is used but other abstract descriptions are possible as well.
- **Service Transport and Communication Layer:** SOAP defines a message passing protocol, but it does not provide the transport protocol that allows the client-server communication. The most common protocol used in this context is HTTP, but other network protocols might be used instead.

Summarizing, Web services are a very flexible distributed client-server based computer technology and designed for any kind of network. They are completely platform and programming language independent and therefore message oriented by implementing popular standards, like WSDL and SOAP. Web services are self-describing and thus flexibly deployable. They build loosely coupled systems that scale better than strongly coupled systems, like those using CORBA. The Web service description has been and still is developed by major industrial vendors, but also by an important scientific community to provide real interoperability for distributed applications.

The following sections describe the basic Web service standards and access methods SOA, BPEL, Discovery Service, UDDI, Web service Addressing, WSDL, SOAP, XML and HTTP.

### 2.2 Service Oriented Architecture

The Service Oriented Architecture (SOA) describes a general concept to split software applications into logical building blocks [17] which leads to so called Business Processes. Business processes are described with the Business Process Execution Language (see 2.2.1).

The aim is to group those blocks together to form any new application. One can imagine the SOA concept with Lego bricks, the famous children’s toy, which can be combined to nearly any imaginable construct. Companies using SOA concepts hope to gain massive cost reduction because of efficiency increase. That is not only interesting for business applications, but also for scientific ones. Therefore,
nowadays also various Grid middleware detect the SOA concept as a reasonable solution for their requirements.

SOA characterises an architecture paradigm which completely passes on any underlying technique. The SOA is not Web services, but to realize the SOA concept often Web services are used. Web services deliver the required standard interfaces.

### 2.2.1 Business Process Execution Language

The Business Process Execution Language (BPEL) is a description language which combines IT with business [18]. BPEL was introduced by IBM, Microsoft, and BEA Systems in 2002. It is not a programming language to implement an application, but rather responsible to coordinate a couple of services into a special cycle order. This order is fixed through the application which is to be built from the Business Processes. The coordination of the services is called *orchestration* and consequently the *service orchestration* is the core idea of BPEL. As an orchestra is dependent on a good orchestration to be able to play nice music, it is the same with the set of services to achieve an efficient Business Process combination.

To stay with the orchestra example it is important to have a balanced orchestration for the good sound, but it is also essential to put those sounds together to a sequence which leads to one melody. In the IT one rather would speak of so called *workflows* (see chapter 5.1) instead of melodies. A workflow characterizes a special order of Business processes to define a (cost) optimal and automated cycle.

### 2.3 Web Service Discovery Service

In [10] the Discovery service is defined as follows:

"*A discovery service is a service that enables agents to retrieve Web service-related resource descriptions.*"

All Web service theory would not make sense if there were not mechanism to detect Web services on the Internet.

To detect a service, such a discovery service can act in three ways:

- **As a Registry service:** The Web service providers register their services with the Discovery Service. When asked for a service, it checks the local databases to detect the respective services and returns the Web service locations. The vendor is responsible to reregister the service in case of any change. The disadvantage is that the server hosting the Discovery service has to provide large databases to manage all service locations.

- **As an Index service:** This scenario expects the Discovery service to check actively for Web services when an agent request arrives. This is comparable with a search engine.

- **As a Peer-to-Peer service:** This is a combination of both the registry service and the index service. If more than one Discovery service is available, the first Discovery service checks if the requested service has been registered. If not, it asks some neighbour Discovery service.
If the service is not available there either, the neighbour service asks another neighbour service and so on until the service is found.

Universal Description, Discover and Integration (UDDI) is the most dominating standard protocol to discover Web services, which covers the three areas defined above. The UDDI registry service is a Web service itself managing information about Web services and the Web service providers. According to the requirements, UDDI registries are comparable with the White, Yellow, or Green pages [19]:

- **UDDI White pages**: The calling program knows about the required Web service already, but does not know where to find it. The White pages contain basic contact information about the Web service itself and the provider.
- **UDDI Yellow pages**: This represents a categorized discovery approach. The calling program only knows about the general functionality it wants to get from the Web service (for example looking for weather forecast Web service for Europe).
- **UDDI Green pages**: Contain extended information like business models and processes as well as technical details. This approach could mean an overhead of information. Furthermore it overlaps with the WSDL described in section 2.6 and is therefore not that common.

UDDI does not require WSDL to describe the Web services, but WSDL has a very structured description format. Therefore, it often is the first choice to allow other software to find the Web services.

Having one UDDI registry always means having a single point of failure. Therefore, some approaches focus on having multiple UDDI registries grouped into registry federations [20]. This approach is decentralized and thus has no single point of failure anymore, but rules have to be defined that the same service description is used across all UDDI registries. This can have an impact on the effectiveness of discovering a Web service.

A relatively new Web service discovery approach is described in [21]. The author describes the Web Service Crawler Engine (WSCE) [22]. Again, multiple UDDI registries are available, but with connection to each other. WSCE is capable of crawling all information from those registries and stores them centrally in a uniform way. This centralized Web services repository then can be used for large-scale discovery of Web services.

### 2.4 Web Service Resource Framework

The *Web Service Resource Framework* (WSRF) belongs to the OASIS [23] specifications for Web services. Main contributors include the Globus Alliance [24] and IBM. The WSRF is the interface between Web services and Grid services. That means that WSRF allows accessing, handling and manipulating stateful Grid services instead of stateless Web services. In the whitepaper “*Modeling Stateful Resources with Web Services*” [25] Ian Foster, Steve Tuecke (both from the Globus Alliance), Jeffrey Frey and Steve Graham (both IBM) categorize stateful resources which are defined to:

- have a specific set of state data expressible as an XML document,
- have a well-defined lifecycle, and
be known to, and acted upon, by one or more Web services.

One specification of the WSRF is the Web service Addressing which is described in the next chapter.

2.5 Web Service Addressing

As mentioned in section 2.3 the Discovery service returns the location of the Web service to the calling program. Again this is done by standard mechanism known already from the World Wide Web. The Web services are addressed just like Web pages with plain URIs (Uniform Resource Identifiers). But it is important to remember that Web services are not designed to be human readable but machine readable. That means that if the Web service’s URI is typed in any Web browser one only would get error messages because the Web browsers are not able to handle the Web services code because they are not implemented in HTML. So the URI can only be a parameter in a command line call from any other program.

2.6 Web Service Description Language

The Web Service Description Language (WSDL) is an essential part of the Web service architecture and belongs to the so called Interface Definition Languages (IDL) [26], which have a long history in distributive computing and build the first interoperability approach. WSDL is a result of compromises between IBM’s Network Accessible Service Specification Language (NASSL) and Microsoft’s SOAP Contract Language (SCL). Once more IBM and Microsoft realized that Web services could only be a success story with a single IDL. As a consequence the first WSDL version was submitted to the W3C consortium by IBM, Microsoft and a couple of other companies in 2001.

WSDL covers the functional description of a Web service containing the service’s properties and functionality in an XML document (see chapter 2.9). As mentioned already in section 2.1, Web services are self describing. WSDL cares for returning all required information for invoking the Web service in a machine readable form, once the service is discovered and addressed by the calling program. A WSDL element contains three essential properties of a Web service [11]:

- What does the service? The functionality the service provides, and the interface (arguments and returns) needed to invoke the service.
- How is the service accessed? This contains the network protocol as well as details about the data type and the data required to access the service. The network protocol will usually be HTTP (see chapter…) but could be also any other protocol.
- Where is the service located? Details on the protocol specific network address, such as an URI.

Figure 2 shows an example of a WSDL document.
Figure 2: Example of a WSDL document requesting an offer from a travel agency

```xml
<wsdl:definitions targetNamespace="urn:DefaultNamespace"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:message name="GETPREISRequest">
    <wsdl:part name="PREISUNTER" type="xsd:string"/>
  </wsdl:message>
  <wsdl:message name="GETPREISResponse">
    <wsdl:part name="GETPREISReturn" type="xsd:double"/>
  </wsdl:message>
  <wsdl:portType name="PreisInfo">
    <wsdl:operation name="GETPREIS" parameterOrder="PREISUNTER">
      <wsdl:input message="impl:GETPREISRequest" name="GETPREISRequest"/>
      <wsdl:output message="impl:GETPREISResponse" name="GETPREISResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="DominoSoapBinding" type="impl:PreisInfo">
    <wsdl:soap:binding style="rpc"
 transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:soap:operation soapAction=""/>
    <wsdl:input name="GETPREISRequest">
      <wsdl:soap:body
 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
 namespace="urn:DefaultNamespace" use="encoded"/>
    </wsdl:input>
    <wsdl:output name="GETPREISResponse">
      <wsdl:soap:body
 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
 namespace="urn:DefaultNamespace" use="encoded"/>
    </wsdl:output>
  </wsdl:binding>
  <wsdl:service name="PreisInfoService">
    <wsdl:port binding="impl:DominoSoapBinding" name="Domino">
      <wsdl:address Location="http://localhost"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```
### 2.7 Simple Object Access Protocol

The Simple Object Access Protocol (SOAP) again is a development resulting from competing approaches by Microsoft and IBM but also other big companies like SUN. But in this case not the companies themselves formulated a common SOAP approach but W3C did for over two years and presented their recommendation as SOAP 1.2 in 2003. SOAP is a relatively simple protocol and again it follows the principle of simplicity as one of the key features of successful computer technologies.

**Figure 3:** Example of a SOAP request, which invokes the translation service BabelFish [27]

Generally speaking the client program is completely independent from the Web service. It is very likely that different programming languages and platforms are used and both do not care about each interface. SOAP offers that both partners can communicate with each other anyway by providing an abstract message format of what the program wants and the Web service delivers. Thereby SOAP does not specify the underlying message transport protocol. This is arranged by the WSDL as described in chapter 2.6. But the HTTP protocol will always be the first choice as seen in chapter 2.8.

SOAP provides a message format where data exchanges messages conform to an XML data type. A SOAP message contains the following four elements:

- An Envelope element to identify the XML document as a SOAP message.
- A Header element that contains header information.
- A Body element that contains the SOAP request or the SOAP response. Figure 3 shows an example for a SOAP request and Figure 4 the corresponding SOAP response.
- A Fault element that contains error codes and status information.

Web service providers and consumers want to concentrate on the real functionality without having to formulate complex SOAP messages all the time. Therefore a little helper program exists, called a stub.
These stubs come with the WSDL description and generate automatically SOAP requests and interpret SOAP responses. That means that the program calling the Web service does not have to care about any SOAP specification and the programmer does not have to implement lines of complex code to invoke the SOAP messages. The principle is very simple [12]:

- The client program has implemented a concrete Web service call.
- This is translated into an abstract SOAP message by the client stub.
- The SOAP message is submitted from the client to the server via the network by using for instance the HTTP protocol. This is called SOAP request.
- The server receives the SOAP message and delegates it to the server stub which again translates the SOAP request into what the Web service requires.
- The Web service now does the expected work and delegates the results to the server stub which again produces an abstract SOAP message from it.
- This is sent over the network back to the client. The client stub receives the SOAP response and translates it into something concrete the application understands.

**Figure 4:** Example of a SOAP response, coming from the translation service BabelFish [31]

One can imagine that it is not very convenient to invoke an own server stub for every Web service call. Therefore Web services are embedded into so called SOAP engines. In principle SOAP engines are server stubs as well but serving all SOAP translations to more than one Web service. The Apache Axis [28] is a popular example for a SOAP engine. While SOAP engines only provide SOAP message generation functionality the SOAP engine itself normally runs in an Application server which makes SOAP to function as a service and which provides the possibility to receive different client requests. One example of an application server is the Jakarta Tomcat [29] Even if some application servers
already provide HTTP functionality the application server has to be embedded into Web servers like the Apache HTTP server [30] to get full HTTP functionality for message submission.

SOAP is not the only solution for abstract formulation which is essential for service interoperability. Even if nowadays SOAP is the most common approach also other protocols are still in use. This is for instance the XML-RPC [32] developed as one of the SOAP predecessors by the Userland Company in 1998. XML-RPC describes a much slimmer implementation. In spite of less functionality compared to SOAP it is the first choice for instance in case of some sensors. Those often have very limited memory and limited network bandwidth so that they have to use a less complex alternative to SOAP.

2.8 Hypertext Transport Protocol SOAP binding

The Hypertext Transport Protocol (HTTP) is one of the standard network protocols to transport data over a network via TCP/IP. Originally designed to load Web pages from the World Wide Web into a Web browser nowadays increasingly HTTP is used to transport any kind of data and is not limited to hypertext anymore. By easy extension of request methods, header information, and status codes HTTP is a very flexible but still reliable protocol that became interesting for SOAP message transport as well. In the meantime HTTP became the most common transport protocol in the Web service’s world. Next to the flexibility it has another big advantage: HTTP traffic is supported by the network infrastructure of most organizations and so it is guaranteed that the message will pass the firewalls for instance

```xml
<xml version="1.0" encoding="UTF-8">
  <SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://...">
    <SOAP-ENV:Body>
      <m:getLastTradePrice xmlns:m="trading-url">
        <ticker>SNW</ticker>
      </m:getLastTradePrice>
    </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
</xml>
```

**Figure 5:** Example for a SOAP message embedded into a HTTP request.
Analogues to the SOAP the communication units between client and server are called messages in HTTP. The client formulates a HTTP request while the server answers with a HTTP response. Every message consists of a HTTP header and HTTP body.

shows an example for a SOAP message embedded into a HTTP request. The relation between HTTP and SOAP is shown in the simple formula [34]: \( \text{HTTP} + \text{XML} = \text{SOAP} \).

### 2.9 Extensible Markup Language

The Extensible Markup Language (XML) is a W3C recommendation as a complement to HTML. It is a markup language much like HTML but also with essential differences. While HTML is designed to display data, XML is designed to transport and store data independent from software and hardware. In contrast to HTML there are no predefined tags in XML. XML allows the author to define his own tags and his individual document structure. As the only standard a XML document must have one root element from that so called children and sub children are derived. The structure of a XML document is comparable to a directory structure or a tree structure. A XML document consists of elements and attributes, the respective value assignments and the content of the elements like text or child elements.

With XML an author is able to define his own markup languages. This is comparable to the object oriented programming approach: The classes in those programming languages are the tags in XML. To guarantee universal unambiguousness of elements and attributes so called namespaces can be addressed in the XML document. This is especially relevant if tags with the same name are used in the document. Only by addressing the respective namespace the context the element has to be interpreted in is defined. For this reason W3C defines the qualified names [35]. Qualified names always have a prefix describing the namespace and a local name part which defines the name of the element or attribute within the namespace.

Since XML is just plain text it can be read by any program which is able to handle plain text. XML is the basis for all Web service components. Without XML there would be no WSDL, SOAP and UDDI. Therefore often the literature speaks of XML Web services instead of Web services [36].
3 Existing Grid Monitors

Having learned about the Web services theory in the previous chapter, this chapter concentrates on the monitoring of the Web services and Grid services. As mentioned already in chapter 1, Grid services are an enhancement of Web services as they are stateful. To ease the notation, in the following only the term Web service will be used which then also means Grid service.

When searching for Grid monitors one will find a couple of already existing tools which can be divided into two general classes, the system level Grid monitors and the user level Grid monitors.

Based on both views Grid monitoring can be categorized into four areas:

- System level Grid monitoring.
- User level Grid monitoring.
- Combined user level and system level monitors.
- Commercial Business Process monitors.

The following chapters describe a selection of the most common monitoring tools separated into the above categorizations.

3.1 System Level Grid Monitoring

System level Grid monitoring means to monitor the aggregation of single compute nodes or clusters to one computing unit. The system level Grid monitor providers then offer a complex control of system information like CPU load, memory status, performance, and network information (traffic, flows, connectivity, and topology).

Two representative examples of hardware Grid monitors will be listed more in detail in chapters 3.1.1 and 3.1.2. This is important in view of the fact that one possible extension to the system developed in this thesis could be the integration of hardware information monitors into the Web service monitor to get an overall view of the system.

3.1.1 Ganglia

Ganglia [37] is an open-source software that grew out of the University of California, Berkeley, in the Millennium Project [38]. Ganglia is a scalable monitoring system for distributed computing systems such as clusters. It has been used to link clusters amongst others across university campuses especially in the USA. [39]
Existing Grid Monitors

Figure 6: Ganglia screenshot showing cluster statistics [40]

Ganglia monitors the overall health status of the respective nodes. This includes detailed checks of the hardware like CPU status, memory, network etc. It deals with a client/server approach. One central entity collects information from components which have to be installed on every node to monitor its state. This approach leads to an increasing administrative overhead the more nodes will be added. The monitoring components have to be configured manually on every node. Ganglia scales to handle clusters with up to 2000 nodes. It also could get problematic if nodes with different security policies have to be monitored [41].

Figure 6 shows an example of the Ganglia Web interface with cluster statistics.

3.1.2 Hawkeye

Hawkeye [42] is a monitoring tool designed in the Condor project [43] which developed the Condor software especially for high throughput computing. Condor aims to provide traditional batch system functionality like job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. But additionally Condor can be used to manage a cluster of dedicated compute nodes and is designed to find the fastest CPUs or nodes with the least load, which is essential for high throughput computing.

Hawkeye has been developed to monitor Condor nodes concerning CPU power, system and memory load, network troubles, user information, and overall node status. This is done through periodically executed tasks on the target nodes which adopt sensor functionality. That means that those sensors have to be installed and configured on every node which is to be monitored. The Hawkeye functionality bases on the so called Condor ClassAds (Classified Advertisements). A ClassAd is a
mapping from attribute names to expressions. The expressions can be simple constants but also flexible conditions and comparisons to other ClassAds.

3.2 User Level Grid Monitoring

The user level Grid monitoring is for checking the underlying Grid middleware for availability, correct functionality, status of running jobs, etc from the user’s point of view. Since most Grid middleware have switched to a Web services based approach in the meantime the Grid monitors are Web service monitors in this context as well.

It is assumed that the necessary hardware and network status tests are done by a hardware monitoring tool, and both then send their data to a central entity. Examples of user level Grid middleware monitors are MDS (section 3.2.1) and INCA (3.2.2).

3.2.1 Monitoring and Discovery Service

The Monitoring and Discovery System (MDS) originates from the Globus project and is part of the Globus Toolkit [6]. It implements a standard Web service interface to local monitoring tools and other sources. The latest MDS4 bases on the Web Service Resource Framework (see section 2.4) and builds query, subscription and notification protocols on it. MDS consists of three layers. With the Index Service and the Trigger Service two higher level services are built on the so called Aggregation Framework. The third level consists of the Information Providers. The Index Service is an OGSA-compliant Web service that serves as a collection point for status information from other Web services [44]. While the Index Service is a high-level service the Trigger Service collects concrete resource information and performs actions as sending an email in case of certain conditions. The Information Provider interfaces to different monitoring sources. At the time of writing four Information Providers have been implemented:

- **WSGRAM Information Provider**: the Web service based Grid Resource Allocation and Management (WSGRAM) [45] is the job submission service component of the Globus Toolkit. It provides information about the local batch scheduler like number of running and waiting jobs, used and available CPUs, job count information, etc. This information provider does not monitor the status and functionality of the WSGRAM component itself.
- **RFT Information Provider**: the Reliable File Transfer (RFT) [46] is the Globus file transfer service. It provides information about the status of the transferred data, the status of the data source and target server, and of the number of running transfers. Again the RFT component itself is not monitored.
- **Ganglia Information Provider**: handles all the information coming from a Ganglia monitoring system (see section 3.1.1)
- **Hawkeye Information Provider**: gathers data about the Condor pool resources described in 3.1.2.

MDS delivers a web interface (WebMDS) for administrators and users to check all collected information in a central point.
3.2.2 INCA

INCA [47] has been developed as a user level Grid monitoring tool to test and measure Grid infrastructure from an impartial user perspective and detect problems before the users do [48]. The INCA development started in 2002 as a partnership between the TeraGrid project [49] and the San Diego Supercomputing Centre (SDSC). INCA1 was deployed in 2003 and since February 2007 the production version of INCA2 is available.

INCA does not come with any predefined functionality. All functionality is reached with the INCA reporters which are Perl modules. But a number of prewritten reporters are offered already. Reporter APIs make it easy to create new INCA tests. The reporters produce standard XML-output which then is interpreted by INCA. Together with required packages, and libraries the reporters are collected in so called reporter repositories which are accessible using a URL. With the repositories INCA enables the reporters to be shared with other Grid deployments.

INCA runs from a standard user account. Running as an administrator could distort the results because of potentially different privileges and standard settings. If the tests run with a “normal” user account it is very likely that the results can be generalized to any account. The tests which come already with INCA are based on user documentation rather than on system administrators’ knowledge.

A wide variety of user-level monitoring results can be collected that reach from simple test data to complex performance benchmark output. The monitoring results have to be available in XML-format to be stored and archived. Statistics are created from the archives which is important to analyse how reliable a service has been in the past. INCA has a comfortable web interface to display all results and statistics. The administrator can select between different views starting at general overview up to the concrete service results. All tests are executed periodically and the timeframe is configured by the administrator.

The security is also an important INCA feature. The communication between all INCA components is secured with SSL, and sensitive data is encrypted before written to disk. INCA can be configured to execute monitoring jobs by using short lived certificates, so called proxies, which base on standard X509 certificates.

If new resources have to be monitored or the monitoring requirements change this can be easily adapted in order to facilitate maintenance of a running INCA deployment.

The INCA architecture is shown in Figure 7. The server consists of the agent, the reporter manager, and the depot. The agent implements the configuration defined by the INCA administrator. It also stages and coordinates the reporter managers that execute the reporters. The depot stores and archives the results and provides full archiving of the reports. Web services query the data from the depot and turn it back in XML format for further processing by the data consumer.

INCA provides an administrator configuration tool, called INCAT [48], to set up the tests and configure the reporters to execute on a set of resources. The configuration also is stored in a XML file and sent to the agent.
3.3 Combined monitors

Combined monitors do both system level and user level monitoring and check the status and availability of the computer hardware and the Grid middleware functionality at the same time. With Lemon and Nagios two of those monitoring systems are introduced below.

3.3.1 LHC Era Monitoring

The *LHC Era Monitoring* (Lemon) [50] development started in 2004 at CERN to monitor amongst others the EGEE [51] compute nodes. It is a client-server based monitor system containing tools for monitoring status and performance of computers and so follows a similar approach as Ganglia. But the crucial difference is the ability to monitor Web services as well. The central entity is the so called Measurement Repository which collects data from a monitoring agent being installed on every compute node. Different sensors report about the hardware and software status of the respective node. The monitoring agent launches and communicates using a push/pull protocol with sensors which are responsible for retrieving monitoring information.

In Lemon a distinction is drawn between three types of data [52]:
**Performance metrics** including CPU usage, load averages, memory use, disc use and performance, sockets, and network availability

**Exceptions** including high load, swap use of over a certain percentage, unavailable services

**Status information** including uptime, boot time, kernel version, etc.

Because the Lemon components have to be installed on every compute node which is monitored, the deployment gets more and more complex, the more sites are involved. According to developer information, Lemon is available on some specific versions of Scientific Linux, Solaris operating systems and AIX [53]. Currently Lemon is only fully supported on the 2.4 series of the Linux kernel.

Lemon does enhanced hardware checks and monitors also the status of running services as Web services. It is possible to integrate own reporters to enhance the functionality. It also allows performing defined actions in case of any failure. This could be an automatic restart for instance without the manual engagement of one administrator. To get the full functionality, one has to install the Lemon version which is based on Oracle databases.

### 3.3.2 Nagios

The first Nagios [54] version was released in 1999. Since mid of 2008 Nagios 3.0 has been available. Nagios is a flexible and powerful host and service monitoring system which is available as open source software. One essential difference to most other monitoring systems is that Nagios does not include any internal monitoring mechanism. The Nagios functionality completely bases on external plugins. This is similar to the INCA reporter approach as described in 3.2.2. Plugins providing basic functionality come with the Nagios releases already. This functionality is described on the official Nagios web page [54] as follows:

- Monitoring of network services (SMTP, POP3, HTTP, NNTP, PING, etc.).
- Monitoring of host resources (processor load, disk usage, etc.).
- Simple plugin design that allows users to easily develop their own service checks.
- Parallelized service checks.
- Ability to define network host hierarchy using "parent" hosts, allowing detection of and distinction between hosts that are down and those that are unreachable.
- Contact notifications when service or host problems occur and get resolved (via email, pager, or user-defined method).
- Ability to define event handlers to be run during service or host events for proactive problem resolution.
- Automatic log file rotation.
- Support for implementing redundant monitoring hosts.
- Optional web interface for viewing current network status, notification and problem history, log file, etc.

A big open source developer’s community has implemented a lot of additional plugins for monitoring various services. All Nagios plugins are available on [55]. Nagios provides the system administrators with a lot of detailed documentation and also a guide to develop own plugins is available. Nagios has
no requirements concerning the plugin programming language but if the plugin is written in Perl an
optional embedded Perl interpreter is available to speed up the execution of the Perl plugins. Because
Nagios does not know what exactly it monitors it just interprets the return codes from the plugins.
Table 1 shows a list of valid return codes together with the respective service or host states.

With version 3 Nagios offers the possibility to introduce Nagios independent user variables to store
additional information that are useful for the interoperation between Nagios and other application such
as trouble ticket systems. Nagios 3 allows handling flexible time periods for scheduled maintenance,
holiday times, or on-call duty for instance. Also broad notification logic is available to notify
individual persons or groups by email, SMS message or pager. It is possible to configure delayed
notification. This might be useful if temporary and cursory problems occur on less critical systems. Of
course if critical services fail the notification is sent out directly. If desired Nagios can also be
configured to alert the administrators in case of repeated status changes which mostly indicates that
the service begins to flutter.

<table>
<thead>
<tr>
<th>Plugin Return Code</th>
<th>Service State</th>
<th>Host State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OK</td>
<td>UP</td>
</tr>
<tr>
<td>1</td>
<td>WARNING</td>
<td>UP or DOWN/UNREACHABLE</td>
</tr>
<tr>
<td>2</td>
<td>CRITICAL</td>
<td>DOWN/UNREACHABLE</td>
</tr>
<tr>
<td>3</td>
<td>UNKNOWN</td>
<td>DOWN/UNREACHABLE</td>
</tr>
</tbody>
</table>

**Table 1:** NAGIOS return codes and the respective service or host states. If the plugin return code is 1,
the host state depends to the administrators’ configuration if this is interpreted as up or down [56].

Compared to previous versions, Nagios 3 has improved the host check logic. While this used to be a
possible bottleneck now a new caching feature has been introduced to delay status requests. This
might become necessary, if for instance network components fail, and a lot of non successful contact
tries increase the load of the monitoring components and disable the monitoring functionality in the
worst case. The caching feature allows the administrator to define the time span before a new request
is started. So a compromise between accuracy and performance has been reached: a bigger time period
reduces the reliability of the status information but increases the performance [57]. As a matter of fact
it is the administrators work to balance this. To increase the monitoring performance Nagios 3 allows
executing parallel host checks.

Nagios comes with a comfortable web interface for administration which shows the status of every
monitored service. While it was developed for running on most UNIX derivatives as Linux for
instance machines with all kind of operating systems can be monitored. Also Nagios requires running
a C-compiler while it is written in C. It has to be installed only once and all machines which have to be
monitored are configured in this installation.
Except for the flexibility and easy installation and configuration the production status of Nagios is another big advantage and it will be enhanced continuously. Nagios is used in production in both commercial and scientific infrastructures worldwide. Amongst others the German Bundesverwaltungsamt, which is the central service provider for the German government, is using Nagios to monitor all their machines and services.

Figure 1 shows the Nagios starting page which navigates the visitor through critical and enabled services and warnings.

**3.4 Commercial Business Process Monitors**

Various products are available for monitoring Business Processes in general and Web services based Business processes in special. Representative two products are introduced in the following chapters. All of those have in common that they optimize business-management aspects of sequences of operations.
3.4.1 IBM WebSphere Business Monitor

The IBM WebSphere Business Monitor 6.0 is a part of the IBM WebSphere suite to develop, administrate and monitor Business Processes. The IBM WebSphere Highlights are:

- Efficient management of unexpected business situations.
- Understanding the optimization of Business Processes.
- Tracing of essential performance indicators for the orchestrated services which are to be monitored.
- Analysis of the real time monitoring data.

IBM offers the WebSphere Business Monitor to their customers for them to receive the acknowledgement and analysis information of the Business Processes to be able to enhance and to optimize them permanently. The Business Processes are monitored in real-time. WebSphere Business Monitor answers the following questions:

- Are the Business Process performances acceptable compared to the goals of the company?
- What arrangements have to be done in case of business critical situations?
- Does the Business Processes behave as expected or is the Business Process cycle disturbed?

The cycle of Business Processes has to be defined within the IBM WebSphere Business Modeler [60] and so it does not allow to monitor any Business Process coming from outside IBM.

3.4.2 SEQUENCE

SEQUENCE [61] is a product of the PNMSOFT Company which creates solutions to enhance Microsoft Business Process Management (BPM) [62]. It is a software package that allows building, running and managing Workflows on the one hand but also to monitor them on the other hand. It is customized for the full Microsoft suite including the Microsoft Office [63] and the BizTalk Server [64].

The approach is the same as the IBM WebSphere one: to monitor the Business Process cycles which are created by the BPM. Also real-time operational dashboards are provided including monitoring and alerting applications.

3.5 Assessment of Monitoring Systems

The Grid monitors introduced in the previous sections provide various functionalities to each target community. The two examples of Business Process monitors (see chapter 3.4.1 and chapter 3.4.2) are quoted here for the sake of completeness. Even if they monitor (Web services based) Business Processes they are not relevant for the scope of this work. Both products are designed for big companies that have to administrate and monitor thousands of different Business Processes. The business-management approach is the main focus. The introduced tools by IBM and Microsoft/PNMSOFT allow only monitoring Business Processes which are generated with in-house developed applications. So the monitors are not relevant to monitor foreign services. Another disadvantage is that those products are not available for free. It is difficult to explain system
Existing Grid Monitors

administrators to pay for a monitor software to monitor freely available Grid middleware. Therefore the thesis only concentrates on the system level and user level Grid monitors in the following and not on the commercial Business Process monitors any more.

When dealing with system level Grid monitors like Ganglia and Hawkeye only hardware component specific monitoring is done. This could be reached by Web services sending out the information to the monitor. But the system Grid monitors are not designed to check the Web services functionality itself. At best the system level Grid monitors could be taken as an add-on to the monitor which will be designed in this thesis to get an overall overview about the system status. Therefore those monitors will not be examined further in the following chapters as well.

Much more relevant in the context of this thesis are the user level Grid monitors like MDS and INCA. MDS delivers various functionality for monitoring Grid middleware and Web services in general. It bases on Web services standards and thus is flexible to add new monitors while they are WSRF compliant and the provided data matches the XML schema. The WebMDS provides comfortable opportunities for checking all available information. A disadvantage is that MDS is delivered only with the Globus toolkit software and all Globus components have to be installed to use MDS for Grid monitoring. This means quite a big overhead. Every time the Globus version changes the MDS has to be adapted as well. Another concern is that at the time of writing only four information providers are available, two Globus specific ones and the other two to handle the information coming from Ganglia and Hawkeye.

INCA in contrast delivers the more promising approach to integrate a Web service monitor. It is not dependent on any Grid middleware and so not dependent on version changes. Since INCA provides a flexible framework to add new reporters all kinds of services can be monitored and so all kinds of Web and Grid services as well. INCA delivers various ways of displaying the data and of generating statistics. Especially the long-term statistics are valuable information for the administrator concerning the stability of each service. INCA is publicly available open source software which is maintained and permanently enhanced by the developers. An important criterion is also that INCA is used already in big infrastructures such as DEISA [68](see chapter 5.2).

The third category of Grid monitors, the combined user level and system level Grid monitors are promising for the intention of this thesis as well. The big advantage compared to the user level or system level Grid monitors is that not only software or hardware is monitored but both within one tool. Web services can only do what they are expected to do if the underlying hardware is available. So it is essential to get knowledge about the overall health status of each system. Examples for the combined monitors are Lemon (see chapter 3.3.1) and Nagios (see chapter 3.3.2).

Even if Lemon allows adding new functionality in form of reporters and does both enhanced hardware checks and (Web) services checks it turned out to not to be that reasonable as basis for this thesis for following reasons:

- Lemon has to be installed on each compute node that has to be monitored. So the complexity increases with every additional resource.
- Lemon is supported only for some special Linux derivates.
Lemon reaches the full functionality only with an additional Oracle database which leads to additional costs and installation effort.

In contrast to Lemon Nagios follows a centralised approach and has to be installed only once to monitor any number of additional resources. Nagios does not deliver Web services monitors in the default suite but the functionality can be enhanced as flexible as in INCA by simply adding new reporters. As INCA, Nagios provides a flexible framework which produces various kinds of statistics for and notifications to the administrators. Nagios delivers many hardware monitors by default so that the administrator does not have to care about those any more when developing the Web service monitor. Implementing new reporters and using the Nagios Web interface is comfortable. Nagios is deployed by many scientific and commercial computing centers to monitor various resources. Just like INCA Nagios is permanently enhanced, well documented and the licenses are available for free.

Summarizing the strong arguments for using INCA and Nagios are:

- Both are flexible to adapt to the local requirements.
- It is easy to add new reporters or to change existing ones.
- The respective developers maintain and enhance the software continuously.
- Developed by a professional community.
- Production status is reached.
- Both monitors are widely spread and used in very different environments for very different approaches.
- Used in many supercomputer centres already for other monitoring issues.
- Detailed and actual online documentation is available including FAQs and support pages.
- Both products are for free and each license allows them to be installed anywhere.

Due to all those pros both tools are promising as part of integrating the Web service monitor to be developed. For this reason only INCA and Nagios will be considered in the following.
4 Design of a Web Service Monitor

The monitor to be designed has to fulfil various aspects. The first question is, if reasonable general implementation concepts and ideas are available already. If so, the next question will be, if these ideas can be adapted to the requirements of the concrete implementation of the monitor. Is the concept flexible enough that it can be used to monitor different Grid middleware for instance and thus provides with interoperability?

The next sections will first contemplate a basic monitoring concept (GMA), followed by design decisions basing on this concept. Finally concrete requirements will be deducted.

4.1 Grid Monitoring Architecture

When developing a Web service monitor in general or a monitor for Grid middleware in specific the work which has already been done by the Open Grid Forum (OGF) [65] is relevant. The OGF is a consortium of various international experts and targets the development of Grid standards. While at the beginning of the Grid middleware development in the late 1990s nobody thought about using standards nowadays the use of standards is an essential criterion for the Grid middleware quality.

In 2002 the performance working group of OGF suggested a Grid Monitoring Architecture (GMA) [66] as a minimal specification to support required functionality and as a basis for later implementations. As the performance working group developed the proposal the focus of this architecture is on performance as well.

The GMA considers following design aspects:

- **Low Latency**: the lifetime of the gathered information might be very short and information might be updated frequently. With high latency the information could already be out of date when receiving them.

- **High data rate**: if the applications to be monitored produce a lot of data in a short time this data must not lead to damage in the monitor components as well as on the compute node itself.

- **Minimal measurement overhead**: the monitor activities must not have any negative impact on the system load and thereby on the “real” applications.

- **Security**: a Grid may include many distributed resources with many different access rules. The monitor has to be as flexible as to handle those security issues like firewall settings. The monitoring tool itself has to provide authentication and authorisation mechanisms for secure connections to the distributed resources and vice versa.

- **Scalability**: potentially many sites, services or applications are measured and the data is required by many entities, therefore scalability with respect to measurement, transmission, and security is essential.
Figure 9: Grid Monitoring Architecture components

The GMA architecture consists of three layers characterizing three different components: the Directory Service, the Producer and the Consumer as shown in Figure 9. The Producer selects all available distributed monitor information and offers them to the Consumer. The Directory Service is a central registry for all Producers and Consumers and contains information where to find and how to contact them. If the Consumers need information the Directory Service is contacted to send back the respective location. On the other hand the Producer uses the Directory Service to get Consumers of interest. The GMA allows both the Producer and the Consumer to do the initial request at the Directory Service.

Once the connection between Consumer and Producer is established the communication is done between both directly without the detour via the Directory Service. The GMA defines three Producer/Consumer interactions:

- **Subscribe/Publish:** The subscription might be done either by the Producer or by the Consumer. Once contacted control messages are exchanged. After the subscription is established the Producer publishes one or more performance events or monitor events to the Consumer.

- **Query/Response:** this is deduced from the HTTP request/answer protocol. The first contact the *query* always is initiated by the Consumer and the *response* comes always from the

---

**Figure 9:** Grid Monitoring Architecture components

- **Consumer**
- **Producer**
- **Directory Service**
- **Transfer events**
- **lookup location**
- **store location**
- **database**
- **Monitoring System**
- **Application**

---

**Directory Service**

A central registry for all Producers and Consumers, it contains information on how to find and contact them. It acts as a go-between for initial requests and can send back locations or services of interest.

**Producer**

Selects available monitor information and offers it to the Consumer. It uses the Directory Service to get Consumers of interest.

**Consumer**

Requests information from the Directory Service and from the Producer. It can subscribe to monitor events and receive notifications.

**Transfer events**

The communication between the Producer and Consumer is established through transfer events, establishing a direct connection without the detour of the Directory Service.

---

**Subscribe/Publish**

- **Subscription** can be initiated by either the Producer or the Consumer.
- After subscription is established, the Producer publishes monitor events.

**Query/Response**

- Initiated by the Consumer, it relies on the HTTP request/answer protocol.
- The response always comes from the Producer.
Producer. The difference to the Subscribe/Publish interaction is that all monitor information is summarized in one response.

- **Notification:** the notification is not really a Producer/Consumer interaction because the Producer sends all monitor events centralized in one notification to the Consumer. This approach is similar to the Query/Response interaction but the Notification is only a one-stage interaction.

For performance and security reasons GMA distinguishes between two communication channels, one for the control connection and one for the event communication. Only a concept is suggested but no concrete implementation. As a consequence no protocols for control and event data channel are defined. Also GMA does not deal with delivery guarantees. It is left to the implementation if the monitoring data are delivered at-least-once, that means that at least one package reaches the target but duplicates are possible, at-most-once, that means that a package could be lost, or exactly-once, which is an idealised at-most-once and means that exactly one package will reach the target. An example of a concrete implementation can be found in [67]

### 4.2 Design Decisions

With the Grid Monitoring Architecture the Open Grid Forum offers a reasonable concept for a Grid monitoring tool. Even if designed to give a basis for Grid middleware monitors it can be expanded from the Grid middleware to monitor Web services in general. The GMA concept is consciously kept very general. It does not regulate a special realisation and implementation but provides only the core components.

As already mentioned, GMA bases on three fundamental components: the Consumer, the Producer, and the Directory Service. To stay with the Web services terms, the Consumer is the calling program that wants to communicate with a Web Service which is represented by the Producer and which registers at the Directory Service. The Directory Service is comparable with the UDDI (see chapter 2.3).

In this thesis the GMA concept will be taken as a basis to deduce concrete requirements for an implementation of a Web or Grid service monitoring tool. The GMA provides a concept for user level grid monitoring and is therefore a natural candidate as well, since a monitor always should work from the users’ point of view.

According to the GMA concept three Producer/Consumer actions, Subscribe/Publish, Query/Response and Notification, are possible when monitoring Web services. While the first two actions only have a slightly different approach they will be abstracted to one in the following. Hence only two Producer/Consumer approaches have to be considered.

The notification approach means that the Consumer plays a passive role in the communication with the Producer since the Producer pushes all data to the Consumer without any initial request from the Consumer. Assigned to Web services this approach means that the monitor just plays a listening role and waits for the Web service to get in touch. If the Web service does not contact the monitor after a defined period of time, the monitor has to assume that something went wrong. This solution seems not to be practical for monitoring Grid middleware for two reasons:
Even if the monitor would get aware that the according Grid middleware Web service is not available any more, it would not know about the reason of the failure.

The Web service itself has to be implemented to report the status actively and it has to be configured to send the data to the listening Consumer.

In contrast the Query/Response action follows the more promising approach. Here the Consumer does the initial request to the Producer and polls for the information actively. The Producer uses the same connection to send back the answer. So it is guaranteed that the Consumer will get an answer in any case which then can be finished by the Consumer.

The Query/Response action is a first design idea which will be taken for the concrete implementation of the Web service monitor.

Thereupon the target group for such a monitoring tool has to be defined. Primarily the system administrators will work with this tool and benefit from it. They have to react in case of any problems and inform the users if required. Only a subset of the information might be published to common information systems the users can access. The scientist is not interested in specific reasons why a service fails, but only that a service is available and if not that someone is working on the problem.

Another scenario where a Web service monitor could be helpful is resource scheduling. The monitor could be used to propagate the results to Grid schedulers. Some Grid middleware systems provide schedulers that allow the user to submit a job without knowing where it will run. The Scheduler automatically looks for available resources that fit the request and ideally the best resource is chosen then. If all components are available and behave as expected this is an important decision basis to select the resource for job execution.

To guarantee a high degree of reliability the Web service monitors could run on different machines but monitor the same components. Especially if the test jobs produce a lot of data this could have critical impact on the load. Another possibility is to install a monitor which then checks the availability of the Web service monitor on another machine.

There are three essential monitoring steps:

- Analysis
- Diagnosis
- Problem Handling

The analysis checks the Web service status and diagnoses any failure. On the way from the analysis to the problem handling the following conditions has to be covered:

- Is any maintenance scheduled?
- Is the underlying network available?
- Are the involved servers available or has any other hardware failed?
- Is the Web service available?
- Does the Web service answer as expected?

If a failure occurs the problem has to be handled anyway. A must is to inform the administrator. This could be done in different ways and depends on how important the Web service indeed is. If it is
necessary that the administrator has to react immediately an automatic message to the mobile phone should be sent. If not the standard way to inform the administrator should be at least via email. An important question is whether the monitor just informs the administrator and is passive apart from that or if active actions are taken. For instance one could think of implementing the Web service in that way that it restarts automatically the failed Web component in case of any problem. Even if this is an option this never should be the default. The administrators should at least have a look at what happens. Also because of security reasons this option is critical. To restart components often privileged access is necessary and the monitor should run under a non privileged account. This is additionally suitable because a successful monitoring test with a privileged root account does not mean automatically that the test job would behave in the same way for a normal user account.

Once the monitor has got the results from the requested Web service, it has to be capable of recognizing certain text patterns and to translating them into concrete error messages which are human readable. The results should be universally usable, meaning that the format should not be customized for special processing software which deals with the monitoring results. As a flexible way of displaying data the XML language (see chapter 2.9) has excelled within the last years.

4.3 Requirements

This section formulates requirements to a concrete Grid middleware monitoring tool. Basic functionalities of Grid middleware are distributed job submission and results retrieval that are realised via Grid services. The following requirements reference to these concrete functionalities of Grid services rather than to Web services in general and so prepare the basis for the concrete implementation described in chapter 5.2.

As for all tools a basic requirement for the monitor is to be intuitive and easy to use. A common known and platform independent programming language should be used for the implementation. So new settings and modifications can be adopted easily by other administrators. For instance Java, C, or Perl could be a choice. Living Grid middleware will consistently enhance its functionality. So the monitor has to be extensible as well. It has to be easy to install and to configure. At least a command line tool should be available for this but a graphical user interface would be nice to have as well. In large dynamic Grid infrastructures like DEISA and PRACE [69] permanently new resources are added or older ones removed. If additional sites should be there to be monitored, these have to be integrated easily.

Web services have a quite complex structure and functionality. It is not sufficient just to check for the service availability by testing for its existence. But also the correct functionality is fundamental. Services can still be visible in the process table even if the service itself is not available any more or is in an undefined status. As mentioned above various very different Grid middleware implementations are available as e.g. UNICORE and Globus. The function can only be tested by submitting jobs into the respective infrastructures. That means that the monitoring tool has to be flexible be able to exchange the various client commands for each grid middleware.

As the monitor should be capable to checking different sites it also should be possible to run different kinds of jobs on those sites. An asynchronous job submission mechanism is essential. That means that
the monitor would not wait for one job to finish before submitting the job to another site. It could be
that one job is staying in the batch system queue of one super computer but during this the
administrator wants to be informed about the status of other systems. The monitor will be deployed in
many different scenarios and the results have to be universally usable. Computer centres use different
tools for monitoring all their services independently of Web services like INCA, or Nagios. The Web
service monitor has to present the results in a way that all those tools can continue processing them
flexibly. A question in this context is what should be extracted from the results. The results have to be
unfiltered so that different tools can extract different information if necessary.

The interaction with other monitoring and information tools is essential. For instance before the Grid
middleware is checked other reasons that might cause failures have to be excluded, e.g. an announced
maintenance. So the Web service monitor should be able to talk to (Grid) information systems.

If already available software products are used to realise the Web service monitor that software should
be open source under the BSD GNU license [70] to guarantee an individual operational area without
having to buy expensive licenses.

Another essential aspect is security. All well established Grid middleware systems provide user
authorization and authentication. For this purpose standard X.509 certificates or short life certificates,
the so called proxies, are used for both the user and the Grid components. Only if the SSL-Handshake
is established a job submission is possible.

The Grid and resources provider has to guarantee that only authorized requests come from the Web
service monitor as well. So the Web service monitor also has to comply with all security requirements.
While the security policies differ from Grid middleware/infrastructure to Grid middleware/infrastructure the monitoring tool has to be flexible that different policies can be introduced.

At the time of writing all relevant available Grid middleware use standard X509 certificates for user
authorisation and authentication and this is essential for the monitor as well. Even if a monitor will
only submit test jobs the data has to be protected from unauthorized access. In sensitive cases no one
but the administrator has to get knowledge of special projects which are using the resource. So the
monitor results have to be treated absolutely confidential.

Another essential point concerning the security of the monitoring tool is the service reliability. On the
one hand the administrators would not be aware of any problems if the monitor fails and would be
notified by the users in the worst case. On the other hand other components like schedulers might be
dependent on the information from the Web service monitor. But if some failure would occur no jobs
would be submitted to the respective resource anymore even if the resource itself is working well [71].
5 Monitoring UNICORE 6 Services

Nowadays more and more scientists require a lot of high performance computing power to run their complex parallel applications. Even if the available systems get increasingly powerful this often is not enough. As a consequence many applications are designed to run not only on one supercomputer but on several in parallel. Next to a powerful underlying network for instance this implies Grid middleware is also an essential offer to the scientists to hide the complexity and the heterogeneity of the underlying systems and architectures. But the more (super-) computers are connected the more complex gets the middleware infrastructure itself. Therefore it is necessary to have automatic monitors for the Grid middleware as for instance UNICORE 6, to be always aware of any problems.

5.1 UNICORE 6

UNICORE 6 [83] is a Grid middleware which provides a seamless interface for preparing and submitting jobs to a wide variety of heterogeneous distributed computing resources and data storages. It supports users to generate scientific and engineering applications, to submit them and to monitor the results.

UNICORE stands for Uniform Interface to Computing Resources. The UNICORE software was initially developed in the UNICORE [72] and UNICORE Plus projects [73], funded by the German Ministry of Education and Research (BMBF) since 1997 until the end of 2002. After that, its functionalities and its robustness were enhanced within several EU-funded projects, for example, EUROGRID [74] and OpenMolGRID [75]. All these work resulted in the UNICORE 5 middleware which several infrastructures respectively computing centres are employing in production since 2004. UNICORE is an open source activity and implemented by a wide-ranging international developer community.

The development direction is specified by the technical board of the UNICORE forum [77] which was founded in December 1999 by developers, leading European HPC centres, and supporting hardware vendors as a non-profit association to foster the distribution and use of UNICORE, to publish and maintain the specifications, and to coordinate further development. [77] Also since 2004 the standardization efforts of the Open Grid Forum have resulted in the first standard proposals which were already based on Web services. Even if the first proposals were not mature the direction of the Grid community was clearly towards Web services to standardise their approaches and so to move forward to interoperability.

In 2005 the UNICORE developers started to implement a complete redesign of UNICORE 5 which resulted in a new product called UNICORE 6. The first introduction of UNICORE 6 into production was in 2008. In contrast to UNICORE 5, UNICORE 6 now is fully based on standard Grid services which have been elaborated by the Open Grid Forum in the last couple of years. To keep the platform independency the components still are implemented in Java and Perl.
As seen in Figure 10 the vertically integrated UNICORE 6 concept bases on three layers:

- **The client layer**: UNICORE 6 provides a variety of clients, in particular the *UNICORE command line client* (ucc) which is essential for this thesis as described below.

- **The service layer**: contains all the UNICORE 6 core functionality like user authentication and authorisation, service registration, file transfer services and job management services. The components of the service layer are the *gateway* as an entry point to the UNICORE Grid, the *registry* where all services register, the *unicorex* as the central component containing the job management and file transfer services, the workflow components *workflow manager* and *service orchestrator* which are essential for creating Workflows, and the *XUUDB* which contains a database for user authorisation.
- **The system layer**: the *Target System Interface (TSI)* component which is installed directly on the target system, the front end of the supercomputer for instance. The TSI is the interface between the service layer and the system layer. Together with unicorex’s *Incarnation Database (IDB)* it translates the abstract job formulation coming from the user to concrete system specific commands.

Figure 11 shows a typical distributed UNICORE 6 installation at the example of the DEISA infrastructure. To keep the figure clear the configuration of only three out of thirteen DEISA core partners is shown. The configuration is the same at all other sites.

![Figure 11: DEISA UNICORE 6 setup with a central gateway at FZJ also hosting the global registry, and local gateways at CINECA and EPCC](image)

From the beginning of the UNICORE development to the actual UNICORE 6 the central aim has always been to hide the supercomputer, or generally said target system, and technical idiosyncrasies from the user. This requires a high level of abstraction which has been realized with the so called
Monitoring UNICORE 6 Services

Abstract Job Object (AJO) [73] in UNICORE 5 and with Job Submission Description Language (JSDL) [78] documents in UNICORE 6. The AJO is a UNICORE 5 internal abstract job description which bases on Java classes while the JSDL is an extensible XML specification for a standard job description notation developed by the Global Grid Forum. UNICORE 6 bases on the SOA concept (see chapter 2.2) and deals with service orchestration of Business Processes

5.2 SIMON 6

A service called SIMON (SIte MONitoring) is going be the proof of concept implementation for the example of UNICORE 6 of the Web service monitoring concept introduced in chapter 1. Before starting to develop it, it is reasonable to check which monitoring tool is available already for UNICORE 5. Even if UNICORE 5 is not Grid services based the basic concepts may be useful for monitoring UNICORE 6.

In 2006 SIMON 5 [79] was developed by the DEISA project. DEISA stands for Distributed European Infrastructure for Supercomputer Applications and started in 2004 as a European project, which was funded by the FP6 programme of the European Union. In DEISA leading national supercomputer centers are connected, to deploy and operate a persistent, production quality supercomputing environment to the European scientists’ community. The members of the consortium wish to improve the level of exploitation of their systems and, at the same time, to provide a higher quality of service to the users, by being able to offer them a larger joint resource pool [80]. The 13 DEISA partners considered several applications and middleware technologies that are providing the functionalities necessary to integrate their high-performance computing systems. From the beginning UNICORE was the choice of middleware which all partners installed on their supercomputers.

With SIMON 5 DEISA has a powerful monitoring software to check the health status of UNICORE 5 at all sites. This is essential to guarantee high availability and support production status for the users in such a complex environment.

SIMON 5 is based on implementations of DESHL, the DEISA Services for Heterogeneous management Layer [81] that gives a standards based access to a heterogeneous supercomputing architecture. DESHL is a software package also developed in DEISA, which provides a command line interface to UNICORE 5 resources.

SIMON 5 does not only aim to monitor the availability of UNICORE 5 components but the functionality as well. Therefore the UNICORE 5 internal job description (AJO) is used for monitoring jobs. As a consequence SIMON 5 monitoring jobs have to be available in the Java based AJO format as well. In case of any problems with one of the UNICORE 5 components the administrators get informed for instance by email. SIMON 5 is available as an INCA reporter and monitors all DEISA UNICORE 5 installations.

Even if UNICORE 5 differs essentially from the new UNICORE 6 the basic concepts of SIMON 5 (submitting a UNICORE job via a command line interface to monitor the UNICORE servers) have been proven as a reasonable approach within the last years and have consequently been adopted for the
UNICORE 6 (SIMON 6) monitor which has been developed in this thesis. The adopted functionalities are:

- Checking for functionality of each component and not only for availability. A service which is available in the process list does not have to deliver the expected functionality necessarily. For instance if the system load is critically high the service could be available nevertheless even if not providing any functionality any more.
- Notification of administrators in case of any error.
- Displaying the results in some high level monitoring software like INCA or Nagios.

In SIMON 6 these functionalities are reached through a much more flexible implementation than in SIMON 5 which depends on the quite complex AJO structure. The higher flexibility and the lightweight implementation of SIMON 6 now is achieved with the UNICORE command line client (ucc). UNICORE 6 provides the ucc as an integrated command line interface. Using this as an administration interface from SIMON 6 to the UNICORE 6 Grids has essential advantages:

- SIMON 6 does not depend on DESHL any more. Although DESHL is available for UNICORE 6 in the meantime as well and so could be used as the monitoring basis for SIMON 6 the decision has been taken to use the ucc instead to avoid additional software layers.
- SIMON 6 does not depend on the AJO anymore. While developing a test job for SIMON 5 required knowledge of the UNICORE internal job description and Java capabilities the creation of SIMON 6 jobs is much easier because of creating an XML document and using simple ucc commands is sufficient.
- The monitoring is kept as simple and efficient as possible.
- No installation overhead.
- The intuitive ucc-syntax allows performing every action necessary for monitoring UNICORE 6 Grids.

An essential requirement formulated in chapter 4.3 is the security aspect. As mentioned this should comply with the security requirements of the respective Grid middleware. At the time of writing the general Grid security standard is based on X.509 standard certificates which are also used by UNICORE 6. Accordingly SIMON 6 has to be X.509 certificate compliant as well. As ucc needs a user certificate to submit jobs into UNICORE 6, this also is assured for SIMON 6 monitoring jobs. SIMON 6 can use every non privileged user certificate which satisfies the respective requirement from chapter 4.2.

With ucc different jobs can be set up if necessary for instance to monitor different sites. The SIMON 6 implementation allows adding all kind of jobs for different sites just by integrating another ucc command.

Surely the most essential advantage is that using ucc one does not have to care about the Web service definition and the standardized Web service interfaces anymore. While ucc has already implemented all the interfaces to access the UNICORE 6 Grid services this comes for free for the SIMON 6 implementation as well.
SIMON 6 follows the Grid Monitoring Architecture concept (see chapter 4.1). The realisation is seamless as UNICORE 6 already provides exactly the GMA design. Comparing Figure 9 and Figure 10 the identical structure gets obvious. The UNICORE 6 clients are in accordance with the GMA Consumer, the UNICORE 6 Service Registry is the GMA Directory Service and the combination of Gateway, unicorex, and TSI is in accordance with the GMA Producer. As the ucc submits the monitoring jobs this is consistent with the GMA query/response action which is described in chapter 4.1.

**Figure 12: Simon 6 concept design**

Figure 12 shows the SIMON 6 concept design: SIMON 6 submits availability and functionality tests into the UNICORE infrastructure using the gateway as the entry point. All UNICORE 6 components are monitored one after the other, and afterwards the results come back to SIMON 6 again. If any problems with the UNICORE 6 components occur, the administrator will be informed. In the current implementation SIMON 6 can be configured to send an email in this case. But since both INCA and Nagios provide already different notification modes (like sending an SMS to a mobile phone) it is reasonable to use those mechanisms. To revert to those existing solutions keeps SIMON 6 as lightweight as possible. A detailed description of the SIMON 6 functionality follows in the next paragraphs around Figure 13.

As seen in Figure 13, SIMON 6 follows an iterative approach to monitor the UNICORE 6 components. That means that the components are checked one after the other. In principal one could think about monitoring every component at once independently from the others or to monitor only separate components within different timeframes. The Business Process concept of UNICORE 6 would allow this since all components have well defined standardised interfaces. So for instance more stressed UNICORE 6 components like the unicorex could be monitored more frequently than less stressed components like the Gateway. Even if this approach promises a very flexible way of monitoring and is easy to realise through Web services this is not feasible in practice for security reasons. Since the most important operational area for UNICORE Grids are supercomputer centres the
connected computers have to be especially protected. Each centre operates at least a central firewall, often the computers are in a Demilitarized Zone (DMZ) for additional protection. If the monitoring concept would follow the flexible approach the supercomputer centres would have to open their central or internal firewalls for every monitored UNICORE component. This would mean a couple of opened ports, but each open port with a listening service behind is a danger for the site security. Consequently the respective security officers would block this approach.

Following the iterative approach SIMON 6 will use only the Gateways ports as entry point to the UNICORE 6 Grid. At each site this port is opened in its firewall anyway because otherwise it would not be possible to run a UNICORE 6 Grid. SIMON 6 monitors the TSI, the unicorex and xuudb, the workflow components composed of the workflow manager and the service orchestrator, the (central) registry, and the (central) gateway. With the iterative approach SIMON 6 will stop monitoring as soon as the first component fails. If components which are passed through by the job would fail later, this would only be recognized in later monitoring steps.

When speaking of the gateway one has to distinguish between the central and a local gateway. The central gateway in front of a central registry does not necessarily have to be the same as the local gateway in front of any unicorex. Even if this would not be an advised UNICORE 6 installation a third gateway could be used for both central workflow services. In the same context more than one registry, the central one and a local one for every unicorex, are imaginable. In case of any gateway or registry problem SIMON 6 has to distinguish clearly if the error occurred on the central or on a local component.

The Nassi-Shneiderman diagram in Figure 13 shows the working of SIMON 6: After submitting an ucc workflow job into the UNICORE 6 Grid first the central Gateway availability is proved. If the central Gateway is up and running, the job gets the target system information from the central Registry. If this is successful the Registry is up and running as well.

With the workflow manager and service orchestrator the UNICORE 6 provides two separate workflow components. If the workflow is configured to use its own local gateway this has to be checked next. If not the job is forwarded through the workflow components to the unicorex. In case of any workflow component failure no error but just a warning message will be written.

In contrast to the workflow components it is recommended to install local gateways at every site for the unicorex and xuudb services, especially in large distributed environments. As Figure 11 shows this is the case in DEISA for instance.

When checking the unicorex functionality, there could be the same situation as with the workflow components, in case another local gateway is in front of the unicorex. This for instance is the typical DEISA configuration and recommended for distributed installations. If unicorex is also available the xuudb will be addressed next and last the TSI which is located on the target system (on the supercomputer for instance) itself.

Actually SIMON 6 distinguishes three return codes:

- **SUCCESSFUL**: the job submission was successful and the components behave as expected.
- **WARNING**: Either the Workflow Manager or the Service Orchestrator is not available. This is just a warning because other non-Workflow jobs could pass successfully nevertheless.
- **CRITICAL**: One of the components (central) Gateway and registry, unicorex, xuudb, or TSI are not available.

When the job submission should fail at any point on the way through UNICORE 6 all Java implemented UNICORE components throw a unique exception which is sent to standard error of the ucc. SIMON 6 interprets each Java exception with Perl regular expressions and translates them into something INCA and Nagios understand. Even if the TSI component is the only one implemented in Perl it either returns error messages to the unicorex or the unicorex throws an exception directly if the TSI is not available.

![SIMON6 functional structogram](image)

**Figure 13**: SIMON 6 functional structogram

Table 2 shows the possible exceptions from and the meanings for each UNICORE 6 component.
<table>
<thead>
<tr>
<th><strong>SIMON 6 interpretation</strong></th>
<th><strong>UNICORE 6 Java exception</strong></th>
</tr>
</thead>
</table>
| Central Gateway is not available | Can't access any target system factories.  
The root error was: `java.net.ConnectException: Connection refused`  
Re-run in verbose mode (-v) to see the full error stack trace.  
No matching target system available, try 'ucc connect'. |
| Local Gateway is not available | No matching target system available, try 'ucc connect'.  
Run in verbose mode ('-v') for more information |
| Registry is not available | The root error was: `org.codehaus.xfire.fault.XFireFault: java.net.ConnectException: Connection refused` |
| Unicorex or xuudb is not available | No matching target system available |
| Workflow manager is not available | The root error was: `org.codehaus.xfire.fault.XFireFault: java.net.ConnectException: Connection refused`  
`org.codehaus.xfire.fault.XFireFault: java.net.ConnectException: Connection refused` |
| Service Orchestrator is not available | internal server error 500 |
| TSI is not available | Failure reading reply data from a (Classic) TSI  

**Table 2:** UNICORE 6 exceptions and their meaning for SIMON 6
As seen in Table 2 the exceptions are unique for all UNICORE 6 components but for the unicorex and xuudb, which deliver the same error message. Although SIMON 6 becomes aware of any unicorex or xuudb failure at the time of writing it is not possible to distinguish which of both components actually is affected. Therefore a request has been formulated to the UNICORE 6 developers to work on unique error return codes for the unicorex and xuudb. As soon as this will be available it will be easily adopted into SIMON 6 as well.

Within this thesis SIMON 6 reporters have been implemented for both Nagios and INCA and integrated in both tools. Figure 14 - Figure 19 show the outcome of the SIMON 6 tests in Nagios and INCA. For displaying the UNICORE 6 monitoring results the respective standard test suite of both INCA and Nagios has been enhanced with the SIMON 6 reporter. In Figure 14 SIMON 6 reports a UNICORE failure with the unicorex and/or xuudb. The respective row turns to red. At the same time an email is sent automatically to the administrator. The detailed information belonging to this failure is shown in Figure 15. The administrator also has the option to resubmit the Nagios UNICORE 6 test suite manually. If UNICORE 6 is up and running the status is as shown in Figure 16.

Nagios provides various possibilities for summarizing events. One is shown in Figure 17 where the SIMON 6 alert history is displayed. Nagios also offers to generate alert histograms and trends on special services. The Nagios administrator can configure an overview report either of the complete host or of special service groups and single services.

As already stated before it usually is not enough just to monitor the status of the Grid middleware only, also system health issues have to be considered in a production environment. Together with the information coming from SIMON 6, Nagios gives a general overview which even would allow forecasting the status of the running Grid services. Important additional parameters are for instance the current system load, the available disc space or the swap usage. If one of those is alarmingly high this could have negative impact on any running application including the Grid middleware to the point of crashing. Nagios is powerful in this context because all those additional reporters already come for free with the Nagios default package.
5.2 SIMON 6

Figure 14: Nagios warning in case of any UNICORE 6 failure

Figure 15: Nagios detailed information to the detected failure
46 Monitoring UNICORE 6 Services

Figure 16: Nagios view in case of successful tests

Figure 17: Nagios UNICORE 6 monitoring history
Figure 18 shows the SIMON 6 results formatted for INCA. INCA also detects the UNICORE 6 error and generates a link to the detailed error information in Figure 19. INCA informs the administrator directly via email in case of any UNICORE 6 failure. Similar to Nagios INCA provides different analysis views of the respective service or of the whole system.

**Figure 18**: INCA general view showing a UNICORE 6 failure
Summarizing the above the implementation of SIMON 6 follows the design principles and requirements formulated in chapter 1 which are:

- Basing on GMA following the query/response action.
- Checking for functionality of each component and not only for availability.
- Administrator notification.
- Implemented to display the monitoring results in INCA and Nagios.
- Using Grid middleware internal interfaces (as ucc).
- X.509 certificate security.
- Intuitive and easy use through ucc.
- Executing different jobs for different sites.
- Following an iterative approach.
- All used software (ucc, INCA, Nagios) is open source.
6 Summary

The scope of this thesis is to develop a Web service monitor at the example of UNICORE 6. A respective concept design and a concrete implementation of SIMON 6 have been achieved. With SIMON 6 a simple but efficient and reliable tool is available now which has not been available in the UNICORE 6 software bundle so far. Two reporters have been developed, one for Nagios and one for INCA. After a first analysis of already available Web and Grid service monitors the decision was taken to use both Nagios and INCA to display the monitoring results. It has been shown that both Nagios and INCA are suitable to do so. Compared to the other Grid monitors introduced in chapter 3 there are strong arguments to use these tools. First it is very flexible to adapt to the local requirements, second it is easy to add new (own) reporters or to change existing ones and third both INCA and Nagios have reached production quality.

The result of this thesis is the development and implementation of SIMON 6 reporters for both Nagios and INCA that alert in case of any component failure.

An interesting aspect for administrators is how frequent the monitoring tests should be performed. This thesis follows the approach that real jobs have to be submitted to check the health status of the Grid middleware. Even if the submitted test jobs should only require minimal resources and minimal execution time on the target system they block resources for the time running. As a consequence the supercomputer administrators would argue for executing the Grid middleware test suite as infrequent as possible. But on the other hand the Grid middleware administrators want to have the smallest intervals possible between all the tests to get a coherent view on the Grid middleware health status. Resulting from intense discussions with site- and system administrators, a five minutes interval is a reasonable compromise. If a failure is reported the administrators could react for the short term.

In this thesis SIMON 6 has been implemented to monitor UNICORE 6 Grid services by using the UNICORE command line client (ucc). But even if all tests base on UNICORE 6 the previous results can be adapted to any Grid services based Grid middleware. It has been shown that each Grid middleware can be monitored with the respective Grid middleware commands and jobs. Under the assumption that the Grid middleware offers a command line interface, in SIMON 6 the existing UNICORE 6 commands only have to be replaced by the respective commands of the other Grid middleware. The respective job description formats have to be considered as well. Additionally the error codes have to be adapted.

Some requirements remain still open for future developments: As announced by the UNICORE 6 developers unified error messages of all components will be available in one of the next releases. Currently one can not distinguish between a unicore x and a xuudb error. The new messages will lead to changes in SIMON 6 as well so that those messages will be interpreted correctly.

To get the monitoring results SIMON 6 actually checks for the UNICORE 6 health status. It would be interesting in future to integrate other information services as well into SIMON 6 or vice versa as requested in chapter 4.2. This is not available in the implementation of this thesis because at the time of writing the UNICORE 6 Common Information Service (CIS) [82] was under construction and so
not fully available. The CIS will display the status of submitted jobs, the queue status on the (super-)computer, but also provides general information such as scheduled maintenance etc. That information is valuable for SIMON 6 as well, because monitoring should not be restricted to informing the administrator only in case of failures. Once CIS released SIMON 6 will be extended to present the overall status view of the Grid middleware and the underlying systems which will help the administrators to administer the resources more efficiently.
Appendix
### Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AJO</td>
<td>Abstract Job Object</td>
</tr>
<tr>
<td>BPEL</td>
<td>Business Process Execution Language</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>DEISA</td>
<td>Distributed European Infrastructure for Supercomputing Applications</td>
</tr>
<tr>
<td>DESHL</td>
<td>DEISA Services for Heterogeneous management Layer</td>
</tr>
<tr>
<td>DMZ</td>
<td>Demilitarized Zone</td>
</tr>
<tr>
<td>GMA</td>
<td>Grid Monitoring Architecture</td>
</tr>
<tr>
<td>HPC</td>
<td>High Performance Computing</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IDB</td>
<td>Incarnation Database</td>
</tr>
<tr>
<td>IDL</td>
<td>Interface Definition Languages</td>
</tr>
<tr>
<td>JSDL</td>
<td>Job Submission Description Language</td>
</tr>
<tr>
<td>Lemon</td>
<td>LHC Era Monitoring</td>
</tr>
<tr>
<td>MDS</td>
<td>Monitoring and Discovery System</td>
</tr>
<tr>
<td>NASSL</td>
<td>Network Accessible Service Specification Language</td>
</tr>
<tr>
<td>OGF</td>
<td>Open Grid Forum</td>
</tr>
<tr>
<td>OGSA</td>
<td>Open Grid Services Architecture</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OpenMoGRID</td>
<td>Open Computing GRID for Molecular Science and Engineering</td>
</tr>
<tr>
<td>PRACE</td>
<td>Partnership for Advanced Computing in Europe</td>
</tr>
<tr>
<td>RFT</td>
<td>Reliable File Transfer</td>
</tr>
<tr>
<td>SIMON</td>
<td>Site Monitoring</td>
</tr>
<tr>
<td>SOA</td>
<td>Service Oriented Architecture</td>
</tr>
<tr>
<td>SCL</td>
<td>SOAP Contract Language</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>TSI</td>
<td>Target System Interface</td>
</tr>
<tr>
<td>ucc</td>
<td>UNICORE Commandline Client</td>
</tr>
<tr>
<td>UDDI</td>
<td>Universal Description, Discovery and Integration</td>
</tr>
<tr>
<td>UNICORE</td>
<td>Uniform Interface to Computing Resources</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifiers</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Service Description Language</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>WSGRAM</td>
<td>Web Service Grid Resource Allocation and Management</td>
</tr>
<tr>
<td>WSRF</td>
<td>Web Service Resource Framework</td>
</tr>
<tr>
<td>W3C</td>
<td>World Wide Web Consortium</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>Xuudb</td>
<td>UNICORE User Database</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1: A typical Web service invocation [11] ................................................................. 6
Figure 2: Example of a WSDL document requesting an offer from a travel agency .............. 11
Figure 3: Example of a SOAP request, which invokes the translation service BabelFish [27] ...... 12
Figure 4: Example of a SOAP response, coming from the translation service BabelFish [31] ....... 13
Figure 5: Example for a SOAP message embedded into a HTTP request. .................................. 14
Figure 6: Ganglia screenshot showing cluster statistics [40] ...................................................... 18
Figure 7: INCA architecture. Architecture picture taken from [48] ............................................ 21
Figure 8: Nagios tactical monitoring overview [58] ................................................................. 24
Figure 9: Grid Monitoring Architecture components ............................................................ 30
Figure 10: UNICORE 6 architecture taken from [76] ............................................................... 36
Figure 11: DEISA UNICORE 6 setup with a central gateway at FZJ also hosting the global registry, and local gateways at CINECA and EPCC ......................................................... 37
Figure 12: Simon 6 concept design ......................................................................................... 40
Figure 13: SIMON 6 functional structogram ........................................................................... 42
Figure 14: Nagios warning in case of any UNICORE 6 failure ................................................ 45
Figure 15: Nagios detailed information to the detected failure .................................................. 45
Figure 16: Nagios view in case of successful tests ................................................................. 46
Figure 17: Nagios UNICORE 6 monitoring history ................................................................. 46
Figure 18: INCA general view showing a UNICORE 6 failure ............................................... 47
Figure 19: INCA detailed view ............................................................................................... 48
List of Tables

Table 1: NAGIOS return codes and the respective service or host states. ................................................. 23
Table 2: UNICORE 6 exceptions and means for SIMON 6................................................................. 43
References

[3] WSDL: http://www.w3.org/TR/wsd1
[4] SOAP: http://www.w3.org/TR/soap
[9] W3C: http://www.w3.org
[18] Geschäftsprozesse: BPEL - die unumstrittene SOA-Königin (See http://www.silicon.de/software/business/0,39039006,39180638,00/bpel+die+unumstrittene+soa_koenigin.htm)


[27] SOAP request: http://www.torsten-horn.de/techdocs/soap.htm#Beispiel-UDDI-WSDL-SOAP


[31] SOAP answer: http://www.torsten-horn.de/techdocs/soap.htm#Beispiel-UDDI-WSDL-SOAP


[33] HTTP: http://www.w3.org/Protocols

[34] SOAP-HTTP binding: http://www.w3schools.com/soap/soap_httpbinding.asp0


[38] Millenium Project: https://www.millennium.berkeley.edu


[40] Ganglia screenshot: https://www.gridpp.ac.uk/w/images/2/2f/Ganglia-admin-3-load-overview-26042006.png


[47] INCA: http://INCA.sdsc.edu


[52] Lemon Web Monitoring, M. Siket, CERN IT/FIO (See http://cern.ch/lemon-status)


[54] Nagios: www.nagios.org


[56] Nagios return codes: http://nagios.sourceforge.net/docs/3_0/pluginapi.html

[57] Volle Kontrolle – Nagios überwacht Rechner, Netze und Prozesse, Linux-Magazin 03/08, pp29-48 (in German)


[59] IBM WebSphere Business Monitor, Version 6.0 (See http://www.websphere.org)


[61] SEQUENCE: http://www.pnmsoft.com


[64] BizTalk server: http://www.microsoft.com/germany/biztalk/default.mspx


[68] DEISA: http://www.deisa.org

[69] PRACE: http://www.prace-project.eu

[71] Monitoring, B. Hagemeier, 2005 (See: http://wwwcs.uni-paderborn.de/StaffWeb/streit/pgsc/Site/media/seminar/sem_bjoernh.pdf)


[76] UNICORE architecture: http://www.unicore.eu

[77] UNICORE Forum: http://www.unicore.eu/forum


