AGH – UNIVERSITY OF SCIENCE AND TECHNOLOGY
KRAKÓW, POLAND

FACULTY OF ELECTRICAL ENGINEERING, AUTOMATICS,
COMPUTER SCIENCE AND ELECTRONICS

PH.D. DISSERTATION

KAZIMIERZ BAŁOS, M.SC.

SELF-CONFIGURABLE SERVICE WITH ELEMENTS OF ADAPTABILITY FOR MONITORING OF INFRASTRUCTURE AND APPLICATIONS IN A GRID ENVIRONMENT

Supervisor:
Prof. Krzysztof Zieliński

Kraków, 2006

I would like to dedicate
this doctoral dissertation
to my wife,
Marta

Acknowledgements

The author would like to express his gratitude to his supervisor, prof. Krzysztof Zieliński, for support and assistance with research described in this dissertation. The research was partially carried out within the European CrossGrid project (IST-2001-32243 [37]) and partially within the Polish CLUSTERIX (National CLUSTER of LInuX Systems, [95]) project. It was also partially supported by a grant from the Polish Ministry of Education and Science (grant no. 1583/T11/2005/29), and by a grant from Sun Microsystems for the Department of Computer Science AGH-UST.

Abstract

Monitoring is key aspect of modern grid systems. Problems involving the design of a system which would be well suited to a wide spectrum of grid systems are currently the subject of dynamic research by OGF [116] and other standardization bodies (OASIS [117], DMTF [54]). The dissertation proposes the concept of a monitoring service built according to general architectural guidelines for building grid services, but additionally adhering to such criteria as component approach to resource representation and coherent interface for monitored resources. The author of this dissertation claims that, while preserving these assumptions, it is possible to build a monitoring service, which is self-configurable, scalable, and equipped with elements of adaptability. More specifically, the usage of the proposed system includes monitoring of hardware and software (middleware), grid infrastructure, as well as selected types of applications and servers.

The thesis is organised as follows. Chapter 1 contains the motivation for the dissertation, introduces key terms and contains the thesis statement. Chapter 2 is a critical survey of existing standards, technologies, and projects related to grid monitoring. Chapter 3 characterizes the grid as a whole, provides functional and non-functional requirements for the grid monitoring service and contains the thesis statement. The concept of the JIMS system, built according to the given requirements and assumptions, is described in Chapter 4. This chapter describes key solutions applied in the system, such as the component model and a uniform approach to resources representation, cluster- and grid-level self-configurability as well as elements of adaptability. The design and implementation details of this prototype are presented in Chapter 5. This chapter also introduces the concept of sensors for chosen elements of the grid infrastructure and middleware. Chapter 6 covers system evaluation and verification of results, where the author presents results of JIMS system usage in real grid installations and test results dealing with system intrusiveness, performance, and functionality. Chapter 7 presents results and summarizes the dissertation. The dissertation concludes with a list of common abbreviations, acronyms, definitions and indexed words, which should assist readers of this document.

Typographical Conventions

ABBR	The first use of an abbreviation is written in boldface, indicating that the full name of this abbreviation can be found in the list in Chapter 8 (Section 8.1 titled "Abbreviations", page 151). Some abbreviated terms are also explained in the text, in brackets or in footnotes.
Definition	All definitions written in boldface and using the Sans Serif font can be found in Section 8.2 ("Definitions", page 158).
`Source code`	All source code is printed using the monotype font, and placed in frames. Keywords and literals are highlighted according to the programming language used. Some additional source code fragments accompanying the thesis are placed in Appendix A.
Specification	Each italicized word carries a special meaning in terms of the related specification or context.
method()	Methods are denoted by empty brackets (regardless of their signature) and written in lowercase.

Table of Contents

List of Figures

List of Tables

Source Code Listings

1 Introduction

The chapter introduces the thesis and explains key terms and concepts included in the dissertation.

Building a geographically distributed system supporting the research of academic and commercial societies in the area of efficient parallel data processing requires a new approach to designing systems, which are based on networks consisting of hundreds or even thousands of computational nodes. An example of a subsystem, which is one of the building blocks of a geographically distributed computer **cluster**[1] installation, popularly called a **grid**, is a service for monitoring hardware and software infrastructure, which delivers information essential for operating a job management system, a network load prediction service, a performance evaluation service and many other services. Such a monitoring service is also a key element of grid management systems, providing feedback to administrators and controlling components[2].

1.1 Motivation

The most popular contemporary platforms used for building grid infrastructures, such as. **GT** (Globus Toolkit) [65] and **UNICORE** (UNiform Interface to COmputing REsources) [62, 89], can be characterized by such components as a job submission and execution system, resource brokering, security, and infrastructure monitoring. Job management is typically left to a dedicated batch system, allowing execution of jobs in the scheduled time and using predefined computational resources. Resource brokering is commonly designed as a specialized module integrated into the middleware. Its job is to choose the best computational environment for job execution, according to requirements included in the job description.

[1] Key terms used in the thesis are expressed using the Sans Serif font, in boldface, and explained at the end of the dissertation.

[2] Monitoring is also gaining attention in the context of autonomic computing, where it acts as a *sensor* in a controlling loop along with an *effector* as an executor of controlling operations.

Security infrastructure provides authentication and authorization services for users from different virtual organizations[3] leveraging joint grid computational resources. Last but not least, the monitoring service, delivers information about available hardware and software resources and is an essential part of the infrastructure, supporting the grid management system, the resource broker, network and performance prediction components, performance evaluation systems, and others.

The concept of a monitoring system, which would be suitable for such a distributed system with virtualized resources, is currently the focus of dynamic research by many standardization bodies with the OGF (Open Grid Forum [116]) at the forefront. For example, OGF Performance-WG[4] has specified a general architecture for grid monitoring systems – the so-called grid Monitoring Architecture (GMA, [161]), which provides an overview of the major components of a grid monitoring architecture and specifies their essential interactions. A key element of this architecture is a publish-subscribe pattern where producers and consumers communicate indirectly with one another using a directory service (typically LDAP [88]). Existing monitoring systems built in accordance with this specification cover different areas of grid infrastructure, including computational clusters, virtual organizations, user applications, etc. For example, the **EDG** (European IST DataGrid) project [74] integrated many monitoring and performance tools (see Table 1.1) for measurement of different aspects of the grid. Altogether, these tools cover the majority of grid monitoring aspects and provide most of the functionalities and features that are required by grid users and other grid components. Nevertheless, the problem, which prevents users from using these monitoring frameworks, is the fact that the common platform on which these tools can cooperate is very limited in scope[5] and therefore requires a homogeneous grid. This very strong limitation significantly restricts the usage of this toolset family. Moreover, the frameworks and monitoring tools developed in different parts of the globe either do not qualify as real grid solutions[6] or do not provide a sufficient level of online detailed information about the

[3] Virtual organizations have a special meaning in GT-based grids and are typically referred to as **VO**s (Virtual Organization). However, the author would like to avoid relating to any particular type of grid middleware (i.e. GT).

[4] Working Group

[5] Testbeds created under the European IST DataGrid and CrossGrid projects were very strictly versioned using **CVS** (Concurrent Versions System) and maintained over the years without significant upgrades due to problems with compatibility between numerous grid components. Thus, at the beginning of 2005 (end of the CrossGrid project) the testbed still used an old version of the Linux system (Red Hat 7.x) and all its grid components were very strongly interconnected, i.e. the dependences between different modules were very strong. For example, they all relied on an old and early version of the WS framework - AXIS v. 1.2 alpha, and an old Linux package manager, RPM (Red Hat Package Manager) v. 3.x, what limited the overall flexibility of the system and prevented developers from introducing newer and more efficient software solutions. Nevertheless, such restrictive software management in CrossGrid enabled all components to cooperate for a relatively long time, which was important in a near-production environment.

[6] Like Lemon, which is a tool for monitoring clusters in a single, **autonomous** system.

monitored infrastructure. The information is rather static and represents only a subset of common parameters available on different platforms[7].

Table 1.1 Monitoring systems developed within the DataGrid project.

Monitoring System	**Function**
EDG Data Access Prediction [59]	Cost estimation of access to distributed data replicas stored in different locations on the grid.
EDG Network Monitoring System [121]	Network monitoring system built according to GMA[8] specification.
EDG Network Cost Estimation Service (**NCES**) [30]	Provides grid schedulers with an aggregate estimate of network performance in terms of achievable throughput.
GridICE [8]	**GIS**[9]-based infrastructure monitoring system.
Lemon [41]	Monitoring system for LCG[10] clusters.
MapCenter [31]	Grid status visualization tool (services, protocols).
EDG Logging and Bookkeeping Service [97]	Collects resource usage and job status.
R-GMA [46]	A realization of GMA that exploits the relational data model and the SQL query language.
Mercury [78]	A realization of GMA developed in DataGrid and gridLab projects.

1.2 Aim of the Investigation

The results of the investigation presented in this thesis and the corresponding prototype implementation of the proposed solution stem from experience gained during the author's

[7] See for example **CIM** (Common Information Model) described briefly in section 2.1.3, MDS – section 2.2.1, or Ganglia – section 2.2.2.

[8] The GMA architecture is a common approach for building grid-monitoring systems in European projects and is typically used as a high-level architecture of the components and interfaces providing a means for interoperability between heterogeneous monitoring systems. More information on GMA can be found in [161].

[9] GIS – Globus Toolkit grid Information Service and the Globus MDS [129] is a common interface for publishing monitoring information in grids based on Globus Toolkit [76].

[10] The LCG (LHC Computing grid) project is a distributed production environment for LHC data processing. The LHC (Large Hadron Collider), currently under construction at CERN near Geneva, is the largest scientific instrument on Earth.

participation in the development of a grid **testbed**[11] for interactive applications in the European IST CrossGrid project [37]. Such an environment required an online tool for monitoring and management of infrastructure and applications. At the same time, the monitoring system had to enable very specific functionality to cope with a large, distributed environment, with a changing topology, different hardware platforms, varied OS and software package configurations and divergent network configurations.

The monitoring system, which is the subject of this dissertation, constitutes a practical proof of concept, showing the possibility of building a scalable and self-configurable system, with uniform representation and monitoring of grid resources, according to the following design assumptions:

- uniform monitoring of hardware, network and software infrastructure and applications assuming structured and coherent resource representation in the form of software components built according to well-defined rules for deployment, installation and operation,
- dynamic discovery and automatic topological[12] self-configuration, according to the current state of the grid,
- adaptation of its functionality to different aspects of grid monitoring[13],
- dynamic loading, installation and start-up of software components during system runtime[14],
- platform-independent architecture and design,
- lightweight implementation, easy packaging and maintenance,
- low intrusiveness and high performance,

[11] A platform for experimentation for large development projects. In the context of grids, it implies a hardware and software infrastructure which allows execution of computational jobs across many geographically distributed nodes and clusters.

[12] The topological aspect of self-configurability of the described system should be clarified. This topological self-configuration ability concerns the geographical locations of different clusters and is based only on their network IP addresses. Indeed, the topological map of the grid contains no information about the cluster's location itself, but only its IP address. On the other hand, all clusters in WAN have their names registered in the domain name system (DNS), which allows simple and fast identification of every cluster. For example, in the European CrossGrid project, there were clusters spread across different countries, all easily identified within the grid (the following clusters were available in the production testbed, not including development sites): zeus24.cyf-kr.edu.pl, ce.grid.cesga.es, cgce.ifca.org.es, cedar.crossgrid.man.poznan.pl, ce02.lip.pt, cms.fuw.edu.pl, cgnode00.di.uoa.gr, loki01.ific.uv.es, grid01.physics.auth.gr, xgrid.icm.edu.pl, ce2.das2.nikhef.nl, cluster.ui.sav.sk.

[13] Includes adaptability to different versions of the operating system kernel, type of network protocol, type of queuing system installed, etc.

[14] In spite of different modules loaded, the interface designed for monitored data access remain stay the same, providing support for other modules that can be designed in the future. The interface should also make use of a protocol which will be simple enough for implementation in languages other than the one used to implement the system itself

- conformance to majority of monitoring and interoperability protocols[15] for collaboration with other systems.

The aim of the investigation is to solve the problem of building such an automatically configurable and adaptable monitoring system for a dynamically changing grid. Strict resource and applications management is of lesser interest in this thesis, unless it concerns the functions of the monitoring system and issues related to its configuration and installation. Furthermore, the thesis does not attempt to reinvent or replace systems for large-scale grid software installation such as **LCFG** [7] or **Quattor** [120]. It also does not aim to replace modern monitoring systems used in grids, but rather focuses on showing the advantages of grid monitoring and management, using a component architecture for representation of hardware and software resources and applications in large-scale systems, organized as a hierarchical network of computational clusters. Section 1.4 formulates the thesis and consecutive chapters present its proof of concept through evaluation and verification of the system implemented according to the above criteria.

1.3 Basic Concepts

Before the thesis is stated, some terms concerning the grid network should be precisely defined. These terms include *object*, *component*, *node*, *service*, and grid.

1.3.1 Objects, Components, Nodes, Clusters and grids

The concept of the *object* is the basis for the **OOP** (Object Oriented Programming) paradigm and also for defining terms used in *component* architectures.

Object

OMG (Object Management Group) **CORBA** (Common Object Request Broker Architecture) Core Specification [113] says that "an *object* is an identifiable and encapsulated entity". This very terse definition is enhanced in [171], where it is said that "the object is an entity that has a set of *operations* and a *state* that remembers the effect of operations. Objects may be contrasted with functions, which have no memory. Function values are completely determined by their arguments, being identical for each invocation. In contrast, the value returned by an operation on an object may depend on its state as well as its arguments. An object may learn from experience and its reaction to an operation being determined by its invocation history". An extension of the *object* is the *component*. In literature, we may find many definitions of the *component*, focusing on various aspects thereof – among other things on its functions or possible means of deployment. However, for the purposes of this

[15] As major monitoring protocols we should consider at least SNMP and JMX. Web Services should also be considered as an interoperability interface and protocol for monitoring information exchange (as in the case of GT-4 [76]).

dissertation, a suitable definition of the component can be derived from the Java world, the **J2EE** (Java 2 Enterprise Edition) platform and the **UML** modelling language.

Component

A. Thomas, author of the white paper "Enterprise JavaBeans Technology – Server Component Model for the Java Platform" [160], says that "component is a reusable software building block: a pre-built piece of encapsulated application code that can be combined with other components and with handwritten code to rapidly produce a custom application". The UML-centric view of the component, presented by authors of the **SCEA** (Sun Certified Enterprise Architect) guide [4], assumes another definition, which is more closely related to the meaning of the term *component* used throughout this thesis. In this nomenclature, "*component* represents a modular and deployable system part. It encapsulates an implementation and exposes a set of interfaces. These interfaces represent services provided by elements that reside on the component".

Node

Grid semantics strongly depend on the term *node*, which is a base element of any grid infrastructure. It is also a key for specifying the monitoring service that spans many grid nodes. Such a node can be defined as follows: "node is a physical element object that represents a processing resource, generally having memory and processing capability – such as a server. Obviously, nodes include computers and other devices, but they can also be human resources or any processing resources" [4] (or any running applications, A/N[16]). In the grid terminology, nodes play the specific roles of **CE**s[17] (Computing Elements, sometimes called access nodes[18]), **WN**s (Worker Nodes)[19], **UI** (User Interface) [20], and **SE** (Storage Element) nodes[21].

Cluster

Cluster is a composition of many computer systems, connected into one, logical unit. Clusters are characterized by a set of nodes that communicate over a high-bandwidth, low-latency and

[16] Author's note.

[17] According to GT/MDS terminology, CE is a computing cluster with separated computational nodes, seen by the end user as a single software unit with certain computational resources. The main function of a CE is job submission, execution, control and termination management. However, CEs are sometimes referred to as single access nodes, with additional functionality for job management, etc.

[18] Special nodes, designated for use by users and applications external to the cluster.

[19] Nodes, which are actually performing grid computations.

[20] Among other things, UI nodes offer job management tools used by grid users.

[21] User interface nodes are special nodes, designated for use by users, for starting grid jobs. User interface nodes typically do not perform computational tasks. (Note that this terminology may differ in various projects and user interface nodes are commonly referred to as access nodes.)

reliable interconnect such as Myrinet [29] or Gigabit Ethernet. In clusters, nodes are frequently homogeneous in terms of hardware and operating system, and, almost universally, the system is managed by a single administrative unit [106]. Clusters, as sets of computational nodes, are typically equipped with specialized software for job management (like PBS[167] or SGE[152]) and are specialized to perform parallel computations (typically using **MPI**[22]). In many grid projects, clusters are also referred to as *sites*.

Grid

The term grid was coined in 1998 by I. Foster and C. Kesselman in the book "The grid: Blueprint for a New Computing Infrastructure" [67] and was used as an analogy with the electric power grid. Thus defined, the grid was supposed to be a medium, which would provide pervasive access to computing power. Later on, Foster and Kesselman tried to define the characteristic features of real grid systems and treated the grid as a special type of software infrastructure: "A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities" [66]. In his article "What is the grid? A Three Point Checklist" [66], Foster enumerates the conditions that a real grid should conform to, saying that it:

- coordinates resources that are not subject to centralized control,
- uses standard, open, general-purpose protocols and interfaces,
- delivers nontrivial quality of service.

Information services delivered by computational[23] grid*s* are distributed in a way similar to electric energy spread through the *electrical grid*, but using special a type of "cabling"[24]. However, the difference is that *computational grids*[25] should deliver *computational power* to the customer in an invisible way. This invisibility (or transparency) should be achieved in such a manner that the end user is not aware of:

- the complexity of services behind the socket with "computational power",
- the geographical dispersion of computational resources,

[22] Message Passing Interface.

[23] "Computational" is not the most appropriate world for describing current grids. However, it is obvious that the first grids were oriented towards computation and delivery of computational power to the end client. Contemporary grids offer many other services that are much more orthogonal to their first computational applications – among others, they provide huge data storage spaces, high network bandwidth and transporat services, etc.

[24] Contrary to electrical energy, which does not seem to be effectively transmittable without wiring, grid services are easily shared over wireless networks. In this sense, modern grids do not closely resemble traditional wired electrical grids: they give potential users more freedom for wireless access, using any type of electrical equipment, including mobile devices such as notebooks, **PDA**s (Personal Digital Assistant), mobile phones, etc.

[25] The full expression *computational grid* seems to be less ambiguous.

- problems concerning independent access from any geographical location,
- complexity behind the management of virtual organizations that the user belongs to,
- security issues.

Authors of the Ganglia framework (one of the most common grid monitoring service) emphasize the heterogeneity of systems federated over a wide-area network, which are (in contrast to the general Internet) usually interconnected using special high-speed, wide area networks (e.g., Abilene, TeraGrid's **DTF**[26]) in order to acquire the bandwidth required for their applications [106]. These assumptions form a basis for building current grid systems and provide characteristics to which any grid service should adhere. More information on this topic can be found on the OGF web site [116] and in Globus-related publications.

Grid Network and grid Computing

Grid computing is one of grid applications and can be defined as the use of many computers in a network to solve a single problem, requiring a great number of computer processing cycles or access to large amounts of data. However, grid networks are also commonly used for other purposes, such as visualization, data management and network services. In this thesis, the notions of grid *network*, grid *computing* and grid[27] itself will be used interchangeably, depending on the context.

1.3.2 Services, Interfaces and Hierarchical Model

In order to formulate the thesis properly, several further terms have to be clearly explained. Among them, *service*, is one of the key concepts used when describing elements of contemporary distributed systems.

Service

The shortest definition of a *service* can be found in [135]: "Service is a computation that may be performed in response to a request". This computational aspect of the service concept can be extended by behavioural definition enclosed in the specification of Jini (a service-oriented model of distributed systems proposed by Sun Microsystems): "A service is an entity that can be used by a person, program, or another service. A service may be a computation, storage, a communication channel to another user, a software filter, a hardware device, or another user"

[26] The Distributed Terascale Facility network.

[27] The term grid is written by some people using lowercase, and by others using uppercase. Even Ian Foster, Carl Kesselman and Steve Tuecke, the so-called "fathers of the grid", in their papers do not follow a uniform method of expressing the word "grid". The author assumes that this issue can be connected with different meanings of the term, depending on the case (the lowercase "grid" being related to computing resources connected via public networks, while the uppercase version addresses a more philosophical meaning of the grid as a "computational power"). Regardless, the author consequently uses term grid written in lowercase and focuses on its networking nature.

[142]. Service is thus a key term used throughout the dissertation and will be elaborated upon along with more specialized architectural patterns such as SOA and WS-DM.

Coherent and Uniform Interface

Coherent and uniform access interfaces to monitored resources are strictly related to interactions between computational objects. According to **ISO/IEC RM-ODP** (Reference Model for Open Distributed Processing), such interactions are essentially asynchronous and can take the form of operations (similar to procedures, invoked on designated interfaces), flows (abstractions of continuous sequences of data), and signals (elementary atomic interactions). These simple operations can be further divided into *interrogations*, in which the server returns a response to a client request, and *announcements*, in which there is no response to a client request [90]. All these interactions form a set of communicational primitives, which should be designed in a coherent and uniform way in order to ensure seamless communication between different components. In this aspect, *coherent* implies agreement on several aspects of such primitives, such as their semantics and technological scope. *Uniform* means that the way of expressing component functionality remains the same regardless of its function and whether it is a computational node or a running application.

Hierarchical Model

Hierarchical model refers to systems where all components are organized in the shape of a tree. In other words, hierarchical model represents a system whose components form a graph in which any two vertices are connected by exactly one path. Such a graph can be shown as a layered structure, where the *root* of the tree is considered to be the *n*-th top layer, and the nodes with the distance of n[28] from the *root* form the *(l-n)*th layer, where *l* is the maximum distance from the *root* to the *leaf*. Such a structure describes an *n*-layer architecture and is often used to create extensible and scalable systems, not only in the domain of computer sciences, but also in the general society, at the organizational level.

1.3.3 Distributed Resources Monitoring

Describing the grid resource monitoring process in a distributed environment requires exact definition of several additional terms, like *monitoring*, *intrusiveness*, and *introspection*. These terms are defined below.

Monitoring

Monitoring is “the process of dynamically extracting information about the observed system, collecting this information, and presenting it to users in useful formats” [93]. Typical monitoring scenarios include different types of information delivery: pulling (operation

[28] Distance of *n* nodes means that there is a path between the considered node and the root that contains *n* edges.

initiated by the receiver of monitoring information), pushing (operation initiated by the monitored system) and tracing (obtaining information obtained from logged events). Monitoring systems can be specified using a layered model of monitoring information flows, such as for example the four-layer monitoring system model proposed in [63]:

1. generation (sensors), connected with obtaining information from monitored resources by the monitoring system,
2. processing, which can include validation, filtering and conversion,
3. dissemination, i.e. delivery of the information to the user,
4. presentation of the information in a useful form, using the command line or a **GUI** interface, or exposing monitored parameters using a certain **API** interface.

Such a layered monitoring system model is reflected in the concept of the system described in this dissertation.

Intrusiveness

Intrusiveness is the effect that monitoring may have on the behaviour of the monitored system, and results from the fact that the monitoring system shares resources with the observed system (e.g. processing power, memory, storage, communication device and media) [99]. Intrusiveness can affect the following aspects of the monitored infrastructure or applications: performance, correctness of results and execution time overhead (applications).

Introspection

Introspection is the process of runtime discovery of the observed system's internals (types, architecture, objects, interfaces, classes and their relationships, etc.) [99] In the Java platform, introspection is referred to as *reflection*, since the package containing classes for using the introspection mechanism is called Java Reflection API [104]. Introspection mechanisms are intensively used in the presented solution as a common approach to discovery of the functionality of monitored resources, which in turn enables constructing universal interfaces for monitoring resources and helps provide interoperability with other Grid components.

1.4 Thesis Statement

This dissertation puts forward the thesis that it is possible to build a distributed system for efficient Grid monitoring, based on a component-oriented, coherent and hierarchical model, regardless of the environment's complexity and geographical dispersion. Moreover, the thesis proves that such a system can be equipped with elements of self-configurability and adaptability – according to the conditions of the environment it is installed in, which has substantial significance for installation and startup of the monitoring system. This process should occur with minimum administrative effort and should be semi automatic. The system should also adjust itself to variable conditions such as addition or failure of computational nodes, addition or failure of clusters, failure or disruption of interconnecting networks etc.

Efforts to find a solution for the above problem focused on uniform resource management and monitoring in highly changeable environment. Keeping in mind all the assumptions presented earlier, the author proposes the following thesis statement:

There exists a component-based architecture, which enables the construction of a self-configurable and scalable service for monitoring of infrastructure and applications in a grid environment, equipped with elements of adaptability.

Such properties as *self-configurable* and *adaptable* require additional explanations, as does coherent and uniform approach to resource representation. This approach is part of the proposed architecture and will be further explained in the thesis.

1.5 Research Contribution

The main contributions of this dissertation are as follows:

1. functional specification of requirements for a worldwide, dynamically self-configurable monitoring service with elements of adaptability,
2. concept of a system solving the problem of grid resource monitoring in a highly changeable environment,
3. proof of concept by practical verification of the presented solution in a real grid environment, in many aspects:
    - unified interface for all monitored resources,
    - dynamic discovery at the level of clusters and the grid itself,
    - dynamic deployment of software modules,
    - interoperability,
4. long-term system evaluation resulting from using the system in real grid testbeds[29],
5. performance tests.

1.6 Research roadmap

Motivation for the presented work originated from the need for monitoring grid infrastructure in a heterogeneous and highly changeable environment. The steps, which guided the author to this solution, are shown in Figure 1.1. The solution was achieved following three consecutive iterations, each of which consisted of concept and implementation phases, which practical verification of the proposed solutions. According to the figure, the first partial proof of concept was achieved using the JIRO architecture and provided a high level of flexibility and

[29] See explanation of this term in the glossary (Chapter 8).

availability of dynamic deployment of monitoring functionality. Unfortunately, the model developed at this stage did not reflect the real architecture of the grid (with hierarchical division of computational resources into clusters and nodes) and requirements for automatic remote installation. This provided the impulse for refinement of the main concept and for designing a dynamic discovery mechanism for automatic configuration, which was the main goal of the second prototype. The last stage (the third prototype) introduced a dynamic global discovery mechanism for identification of all operating clusters in a geographically distributed environment, designed in a flexible and component-oriented way (not constraining the existing architecture of the monitoring system).

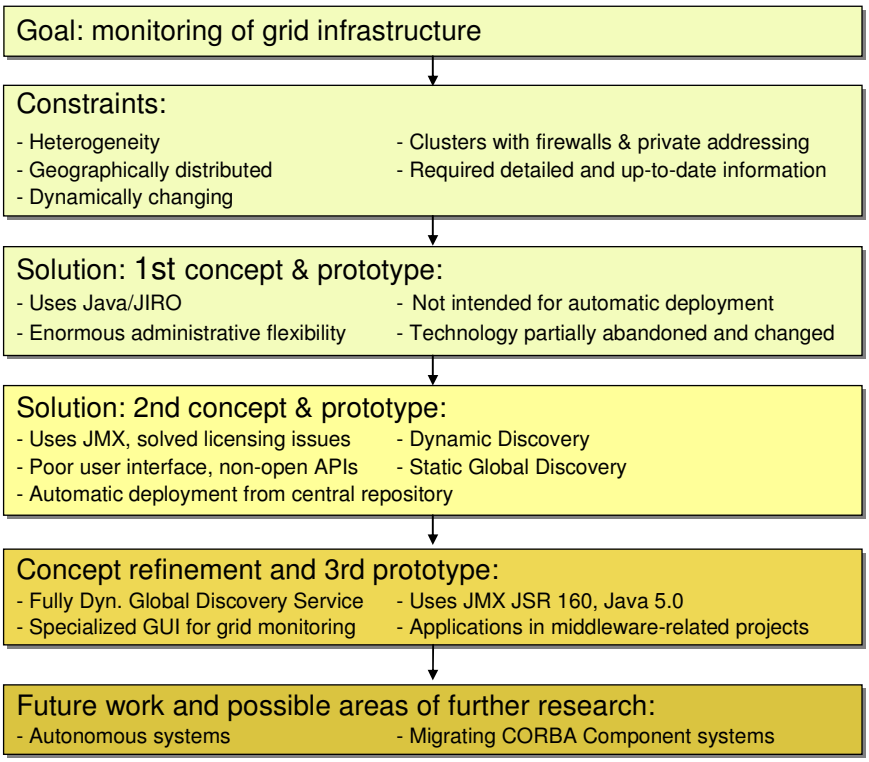


Figure 1.1 The roadmap of research – stages of concept evolution.

1.7 Dissertation Organization

The thesis is organised as follows: Chapter 1 has already presented motivation for the dissertation, introduced key terms used in the thesis (object, node, service and grid) and included the thesis statement. Chapter 2 is a survey of existing standards and technologies related to the problem of grid monitoring and provides a background for the presented research. Chapter 3 discusses the problem of building a new grid monitoring service and defines its functional and non-functional requirements. The concept and design of the JIMS system that address the problem stated in the thesis are provided in Chapter 4. Chapter 5 describes the details of JIMS system implementation using the Java platform and its JMX extensions. Chapter 6 includes system evaluation and verification of thesis results. Chapter 7 concludes the results and summarizes the dissertation. Chapter 8 lists common abbreviations and definitions that might assist readers of this document.

2 Background and Related Work

The chapter provides a background for the presented research. It presents a critical analysis of common information models, standards, monitoring systems, related projects and technologies.

The main goal of this chapter is to evaluate the state of the art of research[30] in the grid-monitoring domain and to see how many existing research results, systems and technologies can be potentially reused in the presented dissertation. In order to achieve this goal, the author performed a survey of topics that are related to the presented research. The survey starts with standards for monitoring information representation and helper transmission protocols described in Section 2.1. Subsequently, Section 2.2 is a survey of existing monitoring and management systems. Comparison of these systems and current trends is presented in Sections 2.3 and 2.4. Section 2.5 contains an overview of the available technologies in the context of management and monitoring, with focus on those that can be potentially used to solve the stated problem. The last part of the chapter (Section 2.7), describes projects related to the research presented in this dissertation.

2.1 Monitoring Information Representation Standards

Efficient operation of monitoring in information systems strongly depends on the underlying resource representation. Therefore, many standards have been developed and established to address this problem [52].

2.1.1 CMIS/CMIP

The author intentionally omits in-depth analysis of the predecessors of contemporary monitoring standards, such as **CMIS** (Common Management Information Service) and **CMIP** (Common Management Information Protocols), which are well described in literature (see [137]) and have a significant potential but are not widely accepted and used in the IT

[30] The background concerns mainly European grids, such as DataGrid and CrossGrid.

community. It is only worth mentioning that CMIP is an OSI protocol implementing the CMIS specifications and it supports information exchange between network management applications and management agents[31]. The main advantage of CMIP over the SNMP protocol (discussed in the next paragraph) is the ability to perform not only simple data manipulation but also any operation using *action* messages, coupled with the ability to define data by means of *create*/*delete* messages.

2.1.2 SNMP, MIB and RMON

SNMP is an application layer protocol and is one of the most popular protocols for network device monitoring and management. Designed in 1988, SNMP has become a *de facto* standard for TCP/IP network monitoring. SNMP was developed to manage nodes (servers, workstations, routers, switches, hubs etc.) on an IP network with defined SMI (Structure of Management Information). SNMP enables network administrators to manage network performance, find and solve network problems, and plan for network growth. Network management systems learn of problems by exchanging or receiving special PDUs (Protocol Data Units) of trap, notification or informational type. SNMP also defines specialized messages such as get and set, which allow the management station to retrieve and set the value of object attributes in the monitoring agent. SNMP-managed networks are built according to NMA (Network Management Architecture), which consists of four elements shown in Table 2.1. Split into two layers, client[32] and server[33], NMA allows for monitoring and management of almost all types of network devices. SNMP does not limit itself to initiating monitoring from **NMS**s (Network Management Stations), but has the ability to send notifications[34] asynchronously to a preconfigured management station, which can be used as an alarm whenever a monitored value threshold is exceeded or a security policy is breached[35]. SNMP makes use of the special **BER** (Basic Encoding Rules) format for encoding monitored parameters and uses the **ASN.1**[36] language [91], which defines types enclosed in **MIB** (Management Information Base) structures in a portable way (independent of internal object representations).

---

[31] There also exists a specification of the CMIP protocol for **TCP/IP** networks, called **CMOT** (CMIP over TCP/IP). CMOT is a successor and initial competitor of **SNMP** (Simple Network Management Protocol) in IP-based networks.

[32] SNMP agent.

[33] Network Management Station.

[34] SNMP traps – types of messages about abnormal events and SNMP informs – informational messages sent from SNMP agent to NMS.

[35] For example, Cisco 2900 series switches can be configured to notify of network packets which break the security policy regarding the source MAC address. Such a notification, called trap, includes the timestamp of the event (object named *sysUpTime*), MIB OID and its value (object identifier 4.1.9.9.87.1.4.1.1.25.0.4 and the value 5 – the switch port number which generated a security trap), the agent identifier and the SNMP version of the agent which sent the message.

[36] ASN.1 defines a way of encoding simple objects (integers, character strings) and some complex types such as structures, vectors and arrays.

Table 2.1 Key components of the SNMP architecture.

Component	Description
Monitoring Agent	A process[37] running within network devices like hubs, switches, routers, printers, network servers, etc. An agent, commonly called the SNMP agent, responds to SNMP requests generated by the network management station and delivers information about specific components of devices or network resources.
MIB	Base and format of managed information, exposed for management by SNMP agents. MIB provides semantic information for monitored parameters by definition and description of SNMP enumeration types (i.e. types of network interfaces) and other networking components.
SNMP Protocol	An application-layer protocol for information exchange between agents and network management stations.
NMS	A network management station, running software for monitoring and accessing distributed SNMP agents.

The structure of MIB defines properties exposed for monitoring and is typically dependent on an implementation built with the following assumptions:

- each object is uniquely identified by **OID**s (Object Identifiers) or dot-separated names,
- it has a hierarchical structure with logical and functional groups,
- it is extendable, capable of supporting additional, vendor-specific properties[38].

Since 1995, SNMP has been extended to support *remote network monitoring* (**RMON**) by means of RMON probes operating in the monitored network. In fact, RMON is a standard, which enables various network monitors and console systems to exchange network monitoring data and measure network flows (called *netflows*). RMON provides network administrators with the possibility to select network monitoring probes and consoles for online network measurement, statistics and historical analysis [45].

To summarize, SNMP offers limited network management functionality[39], but it is very popular in markets for networking devices, software systems (operating systems and

[37] For instance, this can be a process of any open-source package installed on a linux-based server machine such as net-snmp5.11, configured and started to respond to SNMP requests and to generate SNMP traps according to the SNMP specifications. It can also be a process inside a properly configured hardware device such as an IP Cisco router.

[38] This functionality can be described according to extended SMI: *SMI for the Cisco Enterprise*, a Cisco System MIB file (CISCO-SMI-V1SMI.MIB) delivered along with Cisco equipment.

middleware) and even sophisticated professional hardware systems. Due to this fact, SNMP seems to be a mandatory standard in any system that deals with network monitoring, management and networked applications.

2.1.3 CIM Standard

One of the most popular schema for information representation in network management systems is the Common Information Model (CIM), proposed by **DMTF** (Distributed Management Task Force [54]), formed by a consortium of about 200 IT companies such as Compaq, Hewlett-Packard, Intel, Microsoft, Novell, Sun Microsystems and Tivoli Systems (acquired in 1996 by IBM). The newest version of CIM is 2.2 [53]. It provides a common definition of management information for systems, networks, applications and services, and allows for vendor extensions. Common CIM definitions enable vendors to exchange semantically rich management information between systems throughout the network. CIM is composed of a *specification* and a *schema*. The schema provides the actual model descriptions while the specification defines details for integration with other management models [53]. The CIM specification introduces basic structuring and conceptualization techniques for system and network management, while the schema establishes a common framework at the level of fundamental typology. In CIM, the management schema is divided into three conceptual layers:

1. core model – an information model applicable to all areas of management,
2. common model – an information model common to particular management areas but independent of particular technologies or implementations.
3. extension schemas – technology-specific extensions of the *common model* [53].

The CIM information model is currently being implemented by many software vendors for almost all popular operating systems (MS Windows, Linux, UNIX). These implementations are additionally supported by a specialized web-based management interface – the **WBEM** (Web-Based Enterprise Management), also developed and standardized by DMTF.

2.1.4 GLUE Schema

In addition to general-purpose standards, specifying the structure of monitoring and management information, there are also other standardization efforts which attempt to address the problem of differences in semantics of monitored information. One such standard is the **GLUE** (Grid Laboratory Uniform Environment) Schema specification, which is the result of a collaboration effort started in April 2002 by the **EU-DataTAG**[40] [50] and **US-iVDGL**[41]

[39] According to the broadly used 2c version of SNMP. Although there exists a more secure version of the protocol, SNMP version 3, it is not widely implemented and support for this version of SNMP in devices and systems is rare.

[40] Data TransAtlantic grid.

[12] projects. The aim of this information scheme is to define an information model and map it to concrete schemas to coherently represent grid resources in different countries and to enable a large-scale intercontinental grid testbed, including the EDG[42] [74] and **EGEE**[43] projects [92], several national projects in Europe, and related grid projects in the USA [9]. The GLUE Schema defines naming standard for **GIS** (Grid Information Services), which offer varied functionalities and are geographically distributed [9]. The specification describes conventions and terminology related to grid networks as well as the possible types of attributes and their possible ranges of values for each given grid property. The GLUE Schema focuses, among others, on:

- sites and services,
- computing resources (computing elements, clusters, subclusters), storage resources (storage elements, storage areas, access protocols, control protocols, storage libraries),
- computing and storage domain relationships.

Finally, the GLUE Schema was adopted as a standard in the Globus Toolkit, and now stands as a common framework for describing semantic information of monitored resources in grids.

2.2 Overview of Existing Monitoring and Management Systems

Frameworks for resource monitoring and management are another aspect of the literature survey conducted by the author. A large number of systems deal with this subject but each of them only addresses a particular, well defined and rather narrow part of grid computing, ranging from real hardware and software resources that form the fabric of grid infrastructure, and ending with security-related information management and monitoring as well as organizational aspects. This is why grid monitoring is not a closed subject and substantial effort is being invested in European and world grid projects, addressing different aspects of grid applications.

An interesting comparison of currently available grid monitoring and management systems can be found in a white paper issued by the European **APART** (Automatic Performance Analysis: Real Tools) WG: "Grid tools" [75], written by M. Gerndt and R. Wismüller. The document lists many available monitoring systems (the majority being **FLOSS**[44] software) and provides a rough comparison of all of them, based on selected factors such as functionality, target community, performance monitoring, analysis, steering and visualization, instrumentation for resources and applications, scalability and others. The list includes widely

[41] United States part of the International Virtual Data Grid Laboratory project.

[42] An interesting example of mapping between EDG and GLUE data models can be found at http://www.cnaf.infn.it/~andreozzi/datatag/glue/EDG2Glue.html.

[43] Enabling grids for E-science in Europe.

[44] Free Libre/Open Source Software, meaning free and Open Source software, typically based on **GPL** (General Public License).

used frameworks such as **MDS** (Monitoring and Discovery System), EDG **NMA**[45] (Network Monitor Architecture), Ganglia, Nagios and MapCenter, covering many modern applications of monitoring systems.

2.2.1 MDS

One of the most common monitoring systems used withing the context of GT/GIS is MDS, an extensible framework with a hierarchical structure for management of static and dynamic information about the status of grid components generated by information providers. MDS Index Services provide an aggregation service of lower-level data using a special registration protocol and caching to minimize transfer of non-stale data. The most popular second version of MDS is built on top of LDAP, while the third version is a Java system, based on the GT3 Information Services and using architectural elements defined in **OGSA**. The newest release of MDS, version 4, is built according to the **WS-RF** (Web Services Resource Framework) specification proposed by **OASIS**[46] [117].

Development of MDS takes place at **ANL** (Argonne National Laboratory) and **USC/ISI** (University of Southern California/The Information Sciences Institute) and is driven by key advocates of grid computing: I. Foster, C. Kesselman and K. Czajkowski. More information on MDS can be found on the Globus web page [76] and in [129, 77]. MDS is highly influenced by its affiliation with Globus, and supports almost all Globus-based grids. At the time this thesis was written the stable version of MDS was version 2. It treats monitored data in a rather static way, providing the motivation to develop another system which would be more dynamic and which would provide more detailed and up-to-date information.

2.2.2 Ganglia

Ganglia is a scalable distributed monitoring system for high-performance computing systems such as clusters and grids. It is based on a hierarchical design composed of federations of clusters. It relies on a multicast listen/announce protocol to monitor state within clusters and uses a tree of point-to-point connections among representative cluster nodes to federate clusters, aggregating their state. Data is represented in XML, compressed using **XDR** (eXternal Data Representation) and available via the Ganglia web front-end, which can be used to inspect historical data from the past hour, day or month. Ganglia is developed at the Berkeley University of California by Matt Massie [106] and is probably the most widely used grid monitoring system, enabling inspection of the state of infrastructures built on the top of a variety of hardware and operating systems such as Linux IA32/64bit architectures, 64-bit Solaris/UltraSparc, MacOS X, AIX and PPC64 [75]. Ganglia is FLOSS software, available from SourceForge.net.

[45] In SNMP nomenclature, NMA also stands for Network Management Architecture (see Chapter 8).

[46] Organization for the Advancement of Structured Information Standards.

2.2.3 GEMINI

GEMINI (GEneric Monitoring and INstrumentation Infrastructure) [13, 14] is a grid monitoring system developed in part under the EU IST[47] **K-WfGrid** (Knowledge-based Workflow System for Grid Applications) project [58] and coordinated by M. Bubak at the Institute of Computer Science, AGH/UST and ACC Cyfronet AGH in Kraków (Poland). GEMINI is as an example of a grid monitoring system, which is fully compliant with the current WS-RF specification and is designed to run on the WS-RF implementation provided by GT 4.0 [77]. The system is oriented towards workflow monitoring and requires additional monitoring *sensors* in order to monitor given parts of the workflow infrastructure or applications.

2.2.4 GridIce

GridICE is another monitoring infrastructure for grid systems. It enables monitoring of distributed resources and acquisition of a rich set of measurements. These measurements are collected through GIS and then visualized with a web graphical user interface in different aggregation dimensions. GridIce is focused on various aggregations and partitions of monitoring data, based on the specific needs of different users categories (VO, **GOC**[48], site). It supports detection and notification services and is able to derive some network statistics. GridICE is developed at **INFN** (Istituto Nazionale di Fisica Nucleare) within the European DataTAG project, in Italy, by Sergio Andreozzi, Sergio Fantinel, and Gennaro Tortone. GridIce is FLOSS software, available under a BSD-like license, and is used in many European projects, including CMS-LCG0, INFN-GRID and LCG 2 [75].

2.2.5 GridMon

GridMon is a network performance monitoring toolkit developed in the United Kingdom (at the **CCLRC** Daresbury Laboratory, led by M. Leese) to identify grid faults and inefficiencies. The toolkit is composed of a set of tools that are able to provide measures concerning different aspects related to network performance: connectivity, inter-packet jitter, packet loss, Round Trip Time (RTT), TCP and UDP throughput [75]. More information can be found in [102]. Unfortunately, gridMon is not publicly available [75].

2.2.6 SANTA-G and R-GMA

SANTA-G (Grid-enabled System Area Networks Trace Analysis) [46] is a framework that supports tracing rapidly generated monitoring data, with access through GIS [77]. It collaborates with the **R-GMA** (Relational Grid Monitoring Architecture) monitoring service, which provides an information monitoring and logging facility in a distributed computing

[47] Project number 511385.

[48] Grid Operations Center.

environment. It submits all its data to one, large relational database, which may then be queried to find the information required. It consists of *producers*, which publish information into R-GMA, and *consumers*, which subscribe to the monitoring system [151].

2.2.7 Other Monitoring Systems

In addition to the monitoring systems presented above, many other tools exist for management, monitoring, and visualization of grid systems. Most of them are released as FLOSS and are freely available (this group includes – among others – EDG Data Access Prediction, the Network Monitoring System, the Network Cost Estimation Service, the Logging and Bookkeeping Service and Lemon). Others focus on grid organizational aspects and deal with administrative tasks related to user management, virtual organizations and geographical maps of the grid. One of the most successful frameworks in this area is MapCenter [31], which displays the status of each site (cluster), selected services (MDS, GSIFTP, etc.) as well as ICMP, UDP and TCP connectivity. However, such a view of the grid is rather static[49] and is beyond the scope of this dissertation, which instead focuses on online monitoring and resource discovery. Another solution in the form of GT/IS is built on top of MDS and is rather a general framework for indexing WS-RF services and publishing this information in one location. Both MDS and GT/IS represent a fairly general approach following the GMA *Publisher/Directory Service/Consumer* concept, not intended for direct use, but instead utilizing grid monitoring information collected by other tools.

2.3 Monitoring Systems Comparison

As we can see, the divergent approaches presented here have resulted in a vast variety of tools. Some of them are fully grid-enabled; others are only labelled as such or focus only partially on the problem of grid monitoring. Nevertheless, despite their differences, all of them can be very useful for grid users and administrators and can complement one another. In order to show how the presented systems compare to each other within the area of interest of this dissertation (and taking into account the problem formulated in the thesis statement, whose solution is represented by the proposed JIMS system), a diagram has been prepared, comparing common grid monitoring and management systems against arbitrarily-chosen classification criteria.

2.3.1 Classification Criteria

The classification criteria are partially based on the analysis found in [75], and are partially adjusted to show the facilities interesting from the author's point of view. The five classification criteria for comparison of grid monitoring systems are shown in Table 2.2.

[49] Most of the information delivered by MDS is static, i.e. it is based on configuration information entered by local cluster administrators. Furthermore, MDS focuses on coarse-grained resources – for example complete computational nodes and their availability (i.e. not separate CPUs and their actual individual load).

Table 2.2 Classification criteria for comparison of grid monitoring and management tools.

Criterion	Description
Target communities and wide acceptance of specified tool	End users, application and middleware developers, system administrators and other grid components.
Functionality and non-functional capabilities	How the tool is designed for the grid.
Architecture and interfaces	How the system supports common standards for monitoring and management, driven by worldwide initiatives such as ISO/IEC and large consortia of software vendors.
Dependency on third party components	How the tool is portable between different systems and whether it depends on external grid middleware (such as **GT**).
Performance (response time, intrusiveness)	How the tool is suitable to a dynamically changing environment, how it operates under heavy load and how its operation affects the system being monitored.

2.3.2 Comparison of Selected Tools

For the purposes of the comparison, the author selected monitoring systems commonly installed in grid testbeds. The reasons for choosing this particular set of tools involved their popularity in European and worldwide grids and their relation to this thesis. Having in mind the presented criteria and rules stated above, the following frameworks were sleected: Ganglia, gridICE, gridMon, **G-PM/OCM-G**[50], **GT/IS**[51], JIMS, MDS, Mercury, **NWS** and R-GMA, all of which are collected in Table 2.3. For completeness' sake, the JIMS system, which is a practical proof of concept for this thesis, was also included. An in-depth description of this system will be provided in subsequent chapters. The table thus presents monitoring systems addressing different areas of grid infrastructure monitoring. Differences concern various monitoring aspects, scope of information provided by the systems, details of monitoring information and available programming interfaces and access protocols.

Unfortunately, almost none of the presented systems are suitable for detailed, online infrastructure monitoring. Moreover, almost all of them lack easy and semiautomatic (using, for example, LCFG) installation and in most cases require time-consuming configuration (e.g. Ganglia). Furthermore, many of them are still in early stages of development (gridIce, RGMA

[50] Grid Performance Measurement and OMIS-Compliant Monitor for the Grid (OMIS stands for Online Monitoring Interface Specification).

[51] Globus Toolkit 4.0, Information Services/Index Service [77].

Table 2.3 Comparison of monitoring systems.

	Target Community		**Type of Monitoring**			**Interfaces**			**Non functional Requirements**	**Visualization**		**Independency**	**Online Monitoring**
Monitoring System	End users and administrators	Grid components	Sampling	Tracing	Notifications	WS	SNMP	JMX	Automatic installation and configuration	Standalone GUI	WWW interface	Grid middleware (like GT) independent	Deatiled, per-CPU monitoring
Ganglia	✓	-	✓	✓	-	-	-	-	-	-	✓	✓	-
GEMINI	-	✓	✓	-	-	✓	-	-	-	-	-	-	-
GridICE	✓	-	✓	✓	-	✓	-	-	-	✓	✓	-	-
GridMon	✓	-	✓	-	-	✓	-	-	-	-	-	-	-
G-PM/OCM-G	✓	-	✓	✓	-	-	-	-	-	✓	-	✓	-
GT/IS	✓	✓	✓	✓	✓	✓	-	-	-	-	✓	-	-
JIMS	✓	✓	✓	-	✓	✓	✓	✓	✓	✓	✓	✓	✓
MDS	✓	✓	✓	✓	-	-	-	-	-	-	-	-	-
Mercury	✓	-	✓	✓	-	-	-	-	-	✓	-	-	-
NWS	-	-	✓	-	-	-	-	-	-	-	-	-	-
RGMA	✓	✓	✓	✓	✓	-	-	-	-	✓	✓	-	-

or Mercury). Some are not available at all (GridMon) or are not ready to deal with a dynamic environment, impose high levels of network intrusiveness or present only small subsets of monitored parameters (Ganglia), all of which prevents these systems from being used for our purposes.

2.4 General Trends in Grid Services

In the area of massive computing, we can observe many trends that involve changes in hardware infrastructure in order to better support distributed computing. We can also see constant evolution of middleware, convergence with SOA architectures, large-scale effects of merging national grids and trends shaping the construction of huge installations, spanning many countries (European EGEE, EGEE II, commercial grids built by computer companies, such as HP and HP ChinaGrid). Figure 2.1 shows the basis architecture concepts and specifications that have had the biggest impact on the development of grid architectures. The first concept, SOA, was the basis for building grids and heterogeneous distributed systems in general, as described in section 2.4.1. SOA enhancements, which help facilitate the benefits of grid computing, were addressed by two specifications defined by GGF[52]: OGSA and OGSI, proposing a service-oriented architecture for stateful resource management by means of WS. Further work of GGF and related organizations focused on the development of the WS-RF framework for management of stateful resources which are loosely coupled with WS components and which expose their management interfaces. Final work in this area has been undertaken by OASIS and DMTF, working on general architectures for resource management using WS.

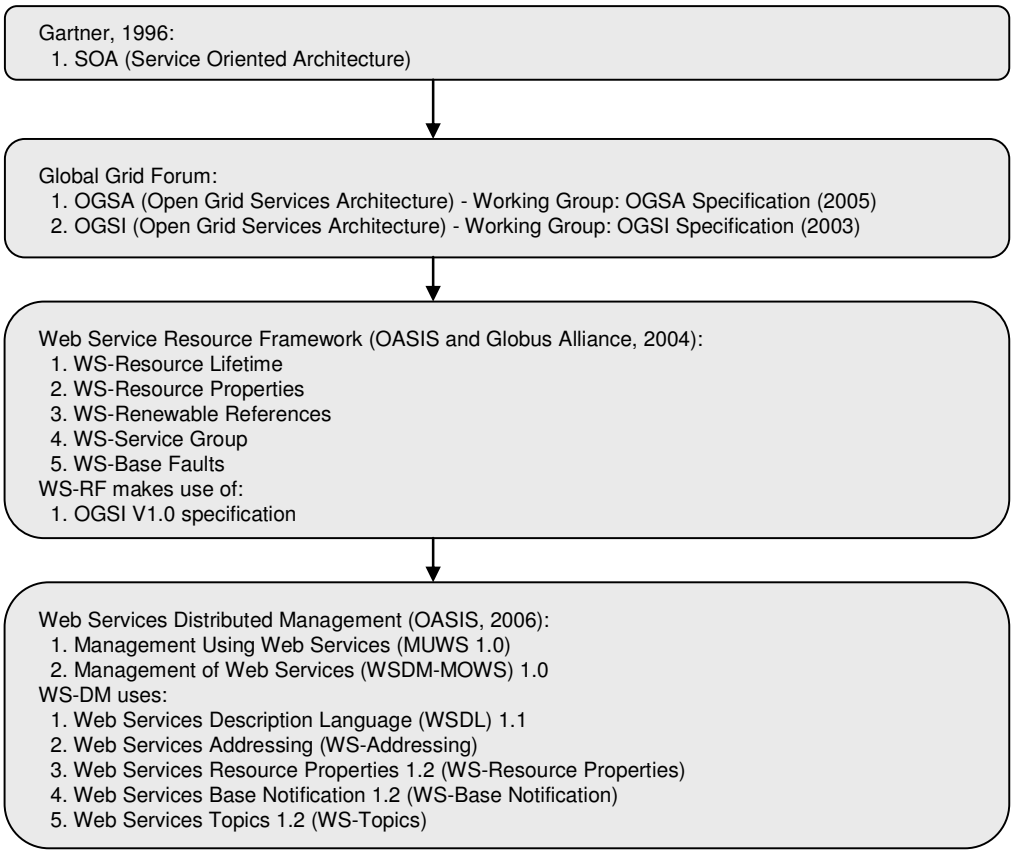


Figure 2.1 Evolution of distributed services architectures.

[52] Currently the OGF.

2.4.1 SOA Architecture

Introduced by Gartner in 1996 [130], SOA is a trend observed in the software layer of grid computing infrastructures. This architecture provides an overall means for moving from strict coupling between software components to open interconnection standards based on SOAP and **WS** (Web Services). Basing on the SOA concept, such standards as OGSA/**OGSI**[53] and WS-RF have been developed, representing milestones in the evolution of grid services.

SOA and WS form a general architecture, which provides the basis for two big frameworks: .NET and J2EE, introduced by Microsoft[54] and Sun Microsystems[55] respectively. According to Microsoft, SOA "is a world-wide mesh of collaborating services that are published and available for invocation on a Service Bus [108]". Sun Microsystems provides a definition, which is more accurate and describes SOA as "an architectural style for building software applications that use services available in a network such as the web. It promotes loose coupling between software components so that they can be reused. Applications in SOA are based on services. A service is an implementation of well-defined business functionality, and such services can be consumed by clients in different applications or business processes". Both these definitions of SOA follow the general paradigm behind the concept described in [107]: "Service-Oriented Architecture (SOA) is an architectural style that formally separates services, which are the functionality that a system can provide, from service consumers, which are systems that need that functionality". A simple illustration of a SOA architecture which adheres to the above definitions, is presented in Figure 2.2

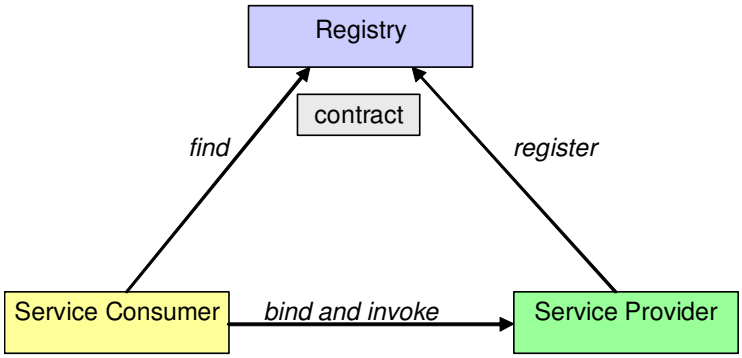


Figure 2.2 Concept of SOA architecture[56].

[53] Open grid Services Infrastructure.

[54] Microsoft is promoting WS as a key technology of their .NET platform as well as in the grid computing area., as shown by its recently released **HPC** (High Performance Computing) version of Windows (CCS 2003) with support for grid computing.

[55] Sun Microsystems is also strongly committed to SOA architecture due to their JINI specification.

[56] The term *bind* is used to express the operation of building a client for a chosen service. More details, including an in-depth explanation of the SOA contract can be found in [179].

Firm commitment to the SOA architecture is possible thanks to wide use of WS technologies, which have become a *de facto* standard, providing a common approach for defining, publishing and using services in a grid environment. Other enhancements to SOA can include additional means of communication, other than using simple services. For example, Oracle is expected to introduce the SOA 2.0 concept, explained as "the next-generation version of SOA", combining a service-oriented architecture and **EDA** (Event Driven Architecture).

2.4.2 Web Services, SOAP & UDDI

Web Services are the most common realization of SOA. According to **W3C** (World Wide Web Consortium), WS are programmatic interfaces to **WWW** (World Wide Web) and are designed to support interoperability between computers in a distributed environment, particularly in grids. Focused on services themselves, WS provide means for describing the functional interfaces of a service in a machine-processable format such as **WSDL** (WS Description Language). The communication layer for WS is typically based on SOAP (Simple Object Access Protocol) messages formatted according a given **XML** (eXtensible Markup Language) schema and exchanged over **HTTP** (HyperText Transfer Protocol). Instead of the SOAP protocol, we can currently observe trends which promote a simpler means of communication called **REST** (Representational State Transfer), based directly on HTTP. The role of the lookup service, which facilitates discovery of WS of a given type, falls to a special service called **UDDI** (Universal Description, Discovery, and Integration). Its role is important in two aspects: service registration by service providers and service lookup by service requestors. These two operations involve WSDL as a description of the registered and looked-up services, while further communication proceeds directly between service requestors and service providers. A typical scenario for WS with UDDI is shown in Figure 2.3 and it is very similar to the SOA scenario (see Figure 2.2).

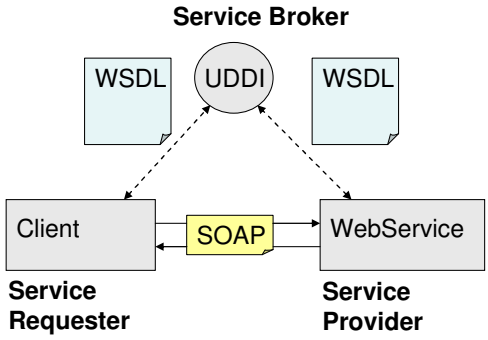


Figure 2.3 Typical scenario for Web Service usage, involving a UDDI registry.

The WS scenario involving a UDDI registry as a lookup service for locating services, which fulfill a well-defined contract, shows that WS strictly follow the SOA paradigm and allow loosely coupled communication between many subjects based on the interface specification included in WSDL. The flexibility of WS has also influenced the design phase of the proposed JIMS monitoring service in the areas of interoperability and communication with applications from different technology domains.

2.4.3 OGSA/OGSI Specification

Introduced by GGF and further developed by OGSI-WG, the OGSA/OGSI (Open Grid Services Architecture/Infrastructure) specifications define mechanisms for creation, management and exchange of information between entities called grid services. According to these specifications, every grid service is a WS that conforms to a set of conventions (interfaces and behaviours), which define how clients interact with this grid service. These conventions, along with other OGSI mechanisms associated with grid service creation and discovery, provide controlled, fault-resilient, and secure management of the distributed and persistent state that is commonly required in advanced distributed applications [164]. OGSI enhances the WS specification and introduces new additional elements: Service Data[57], **GSH**[58] (Grid Service Handle), **GSR**[59] (Grid Service Reference) and **GSI** (Grid Service Interface)[60]. Currently, the OGSI specification has been superseded by the WS-RF framework, which decouples the WS interface from stateful resource representation into two logically independent components.

2.4.4 WS-RF Architecture

D. Fergusson[61] defines the WS-Resource Framework as a set of five technical specifications, shown in Table 2.4. WS-RF defines a special type of WS manageable resource, called WS-Resource, to be declared, created, accessed, monitored for changes and destroyed via conventional WS mechanisms. It is also said that "WS-RF does not require that the WS component of the WS-Resource that provides access to the associated stateful resources be implemented as a stateful message processor (as it was defined in OGSI)[62]" [48]. Work on the WS-RF framework specification was influenced by general specifications for resource management using WS such as WS-Addressing [174], WS-Notification [138], WS-BaseNotification [127], WS-BrokeredNotification [139], WS-Topics [84], WS-Resource [64] and WS-MetaDataExchange [16]. Nowadays, the WS-RF framework is the most recent specification for development of GT-compatible grid services.

---

[57] Means of exposure of data access as a stateful service in order to allow common operations on the exposed data such as querying, updating and changing notifications.

[58] A GSH is a minimal name in the form of a URI and does not carry enough information to allow a client to communicate directly with the service instance.

[59] Actual reference to a grid service, i.e. any client wishing to communicate with a service instance must resolve the GSH to a GSR.

[60] Grid Service Interfaces encompass many types of so-called Port Types. There are many grid Service Port Types such as Handle Resolver (mapping from GSH to GSR), Notification Sources and Sinks, Factories, Service Groups, etc.

[61] Also including other members of the OASIS consortium. OASIS is a group made up of the companies and organizations (Globus Alliance, IBM, HP, SAP AG) clustered around GGF as the **SGML** Open[61] consortium. In 1998 the group renamed its organization OASIS.

[62] A/N.

Table 2.4 Five normative specifications defining the WS-RF framework.

Specification	Description
WS-ResourceLifetime [73]	Describes mechanisms for WS-Resource lifecycle management.
WS-ResourceProperties [83]	Definition of a WS-Resource and mechanisms for retrieving, changing, and deleting WS-Resource properties.
WS-RenewableReferences[63]	Conventional extension of a WS-Addressing endpoint reference with policy information required to retrieve an updated version of an endpoint reference when it becomes invalid.
WS-ServiceGroup [105]	An interface to a heterogeneous by-reference collections of WS.
WS-BaseFaults [165]	A base fault XML type for use when returning faultss in WS message exchange.

2.4.5 WS-DM Architecture

WS-DM is a newer proposal of a modern architecture for resource management, developed by OASIS. Based on the specifications mentioned above, WS-DM not only covers grids, but provides a general architecture for resource management using WS, and for management of WS themselves. According to OASIS, WS-DM can be used to provide WS interfaces for resources described by any generic model such as SNMP, CIM or **NGOSS[64] SID** (Shared Information/Data Model). Focus is on manageable resources, accessible by a WS Endpoint (**EPR**), as defined in the WS-Addressing [174] standard. WS-DM follows the WS-RF approach for resource management and represents all resources as SOA services, with WS components attached (Figure 2.4).

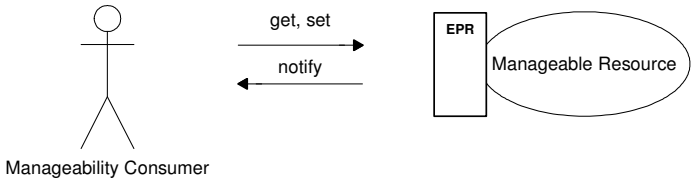


Figure 2.4 WS-DM interaction model.

WS-DM allows creation of resource relationships and subscriptions for notifications as well as advertising newly created resources, which is crucial for a system operating in a highly changeable distributed environment.

[63] Work in progress.

[64] NGOSS, the New Generation Operations Systems and Software is a TeleManagement Forum [157] initiative which integrates business and technical goals of modern telecommunication networks in the direction of more efficient operation and interoperability.

2.4.6 Other Frameworks and Specifications

There are many open source frameworks related to grid monitoring and to monitoring of distributed systems in general. Many are open source[65] reimplementations of commercial software, while others are so-called reference implementations of specifications developed by industry communities such as DMTF, OASIS or IETF. The third set of open-source frameworks (partially shown in Table 1.1) originates from numerous grid-related projects, especially in Europe, since the European Commission places great emphasis on FLOSS software, freely available to developers and not constrained by commercial companies. Selected specifications and frameworks are shown in Table 2.5.

Table 2.5 Distributed system management and monitoring-related specifications.

Standization Body	System Name
OASIS	WS-based specifications: WS Security, WS Distributed Management (WS-DM), WS Resource Framework (WS-RF), WS Notification (WS-N)
GGF/OGF	OGSA, OGSI, OGSCMM (Common Management Model), GRAAP (Grid Resource Allocation Agreement Protocol)
DMTF	CIM (Common Information Model), WS-CIM, WS-Management
IETF	SNMP (Simple Network Management Protocol)
JCP	Java Management Extensions: JSR 003, JSR 160
W3C	WS, Solution Install Schema

Specifications that have had the biggest impact on standardization in the area of architectures of grid services are described in Chapter 3, which discusses general requirements for grid services and grid monitoring services in particular, in the context of standards developed by W3C, GGF/OGF, and OASIS.

2.5 Available Technologies in the Context of Monitoring Systems

Platform-independent resource monitoring in a dynamically changing environment requires a special approach and use of proper technologies. Two technologies presented below, .NET and CORBA, are the key general-purpose platforms considered for building the distributed monitoring system. It should be noted that each of them only covers a selected area of distributed computing and communication, which is important from the author's point of

[65] Open source software is commonly confused with "freely" available software. A discussion on strict distinctions between both types of software is beyond the scope of the thesis; for simplificaty's sake the author uses the term "open source" meaning "free" software (which is true in most cases).

view, in addition to their narrow applicability (.NET platform) and high complexity (CORBA platform).

2.5.1 .NET/DCOM

Released in 2002, .NET is a commercial framework for building distributed applications based on the MS Windows platform. MS .NET provides an intermediate layer of compiled code which is executed within the .NET virtual machine – the .NET runtime environment, sometimes called CLR – the Common Language Runtime. Parts of the application written in different languages are compiled into common, intermediate binary code, which is then executed within this virtual machine. Such an approach enables integration of applications written in different programming languages. It also allows inheritance of classes written in different languages, which had not heretofore been possible on the Windows platform.

The network communication model of .NET is based on the Distributed COM model, which facilitates dynamic loading of application components and development of distributed applications. **DCOM** supports remote objects by running on a protocol called **ORPC** (Object **RPC**[66]). This ORPC layer is built on top of **DCE** (Distributed Computing Environment) RPC and interacts with **COM** (Component Object Model) runtime services. DCOM server-side components can be written in diverse programming languages such as C++, Java, Object Pascal (Delphi), Visual Basic and even **COBOL**[67]. As long as a platform supports COM services, DCOM can be used on that platform (DCOM is now heavily used on the Windows platform [4]). However, as a closed standard, .NET is rarely used for building grid services[68] and neither is it used for implementation of the concepts presented in this dissertation.

2.5.2 IIOP & CORBA

OMG proposes a portable solution for creation of distributed, location-transparent and language-independent applications. Their specification of the **CORBA** (Common Object Request Broker Architecture) architecture provides the possibility of developing applications using a common interface specification language (**IDL**, i.e. Interface Definition Language), which is a declarative way of specifying application component interfaces. CORBA applications can use either static or dynamic interfaces for implementation of server-side logic and for invocation of server functions from the client side. The first case leverages static mapping based on *stubs* and *skeletons*[69] generated at compile time. The second case makes use of DII (Dynamic Invocation Interface) and DSI (Dynamic Skeleton Interface) interfaces

[66] Remote Procedure Call.

[67] COmmon Business Oriented Language, one of the oldest programming languages still in active use.

[68] One recent Microsoft contribution to the domain of grid computing is its new (released in 2006) Windows **CCS** (Compute Cluster Server) 2003, with support for MPI, Job Scheduler, etc.

[69] Stubs and skeletons are helper classes used by the client and server sides respectively, responsible for marshalling of parameters and communication between client and server parts of a distributed application.

for invocation of code by the client side and for implementation of server logic on the server side. All these implementations can leverage different languages, independently on both sides. A key element of the CORBA architecture is the **ORB** (Object Request Broker), which is a collection of interfaces for client and server parts of the application. These interfaces enable creation of **POA** (Portable Object Adapter)[70] objects, adapters for server-side objects, redirecting requests coming from the network to servant objects (the ones which implement all functions defined in IDL, mapped to a particular language). ORBs communicate using **GIOP** (General Inter-ORB Protocol) implemented over specific network protocols, with **IIOP** (Internet Inter-ORB Protocol) being the most popular mapping of GIOP. CORBA also specifies different access methods for interconnecting ORBs developed by different vendors, among which **IOR** (Inter-Operable Reference) is the most common[71] and mosty portable way of storing and interchanging references to remote objects.

The latest specification of CORBA, namely **CCM** (CORBA Component Model), also addresses aspects of designing heterogeneous and distributed systems in the form of components conforming to a given specification. It provides a very high level of abstraction of the underlying system (even higher then JMX), providing a means for defining component characteristics without concern for language implementation details. The author is convinced that CCM is the most advanced component environment currently in use, providing users with a high level of component locations transparency. However, it seems that such a high level of component abstraction is not required for implementing the described grid monitoring service, which is one of the main reasons behind the decision to use another version of component middleware – namely JMX.

2.6 Java Related Technologies

The Java platform supports another, distinct group of technologies, which are either mostly used within the context of Java, or strictly targeted for the Java platform. An example of the former, the **JXTA** platform, is a general, platform-independent specification for direct peer-to-peer communication with its reference implementation and majority of applications written in Java. The latter group is mostly based on **RMI** (Remote Method Invocation), a standard for network communication in Java, and encompasses such technologies as Jini, Jiro, JMX and JDMK.

2.6.1 JXTA

Peer-to-peer is one of the most popular types of networking, allowing creation of ad-hoc networks of equal network entities (peers), for the purposes of exchanging files,

[70] According to CORBA specification, POA is a portable version of the generic (or basic) object adapter called **BOA** (Basic Object Adapter).

[71] Another possibility is the usage of CORBA context for exchanging references to remote objects. This metod is not vendor-independent, however, and for this reason its use is discouraged.

communication and many other more advanced applications (telecommunications, data replication, etc.) Peer-to-peer networking is currently exploited on top of TCP/IP OSI networks, providing a network abstraction, with unified network addressing, independent of underlying technologies and obstacles (firewalls, **NAT** – Network Address Translation, etc.) JXTA, with its specification and implementation, as developed by Sun, gives us an opportunity to build a network of interconnected peers based on many underlying protocols and technologies such as UDP, TCP, multicast and many others [80]. As an open standard, JXTA is not strictly tied to the Java technology; however most JXTA applications are written in Java. As a flexible means of communication, JXTA is also currently being investigated for its potential for adding capabilities to current grid monitoring services and frameworks, including the JIMS system.

2.6.2 RMI

RMI is an **OO** (Object Oriented) platform for communication between distributed applications written in the Java language. It is a successor of the RPC (Remote Procedure Call) approach and it relies on the Java Object Serialization Protocol [148], allowing objects to be marshalled (transmitted) as a stream between the communicating parties. Since Java Object Serialization is specific to Java, both the Java/RMI server object and the client object should be initially written in Java. Each Java/RMI server object defines an interface, which can then be used to access the server object outside of the current JVM and on another machine's JVM. The interface exposes a set of methods, which are indicative of the services offered by the server object [4].

The RMI protocol can be implemented over many transport protocols. Historically, the first transport layer for RMI was called the **JRMP** (Java Remote Message Passing) protocol and operated over TCP/IP. It was a vendor-dependent, proprietary, stream-based protocol, which posed interoperability problems. Current implementations of RMI make use of RMI-IIOP, which is language-independent and seems to be replacing JRMP even within the pure Java world. The reason behind this is probably that the use of RMI-IIOP exposes Java objects to CORBA ORBs and applications written in languages other than Java.

2.6.3 Jini and Jiro

Jini is a service distribution architecture, which enables the creation of network-centric solutions supporting rapid changes. The Jini specification ([149,150]) was proposed by Sun Microsystems and is based on the Java RMI technology. It provides developers with tools for creating an infrastructure of federated services, which can then be used, in a distributed system. This infrastructure consists of three parts:

- a set of components building the set of federated services,
- a programming model supporting the production of reliable distributed services,

- services that can take part in federated systems (such as Java Spaces[72] [71], a persistent store for Java objects).

The Jini specification also describes additional elements of the infrastructure, covering security, transactions and event notification mechanisms. Jini focuses on the lookup service and discovery mechanisms shown in Figure 2.5.

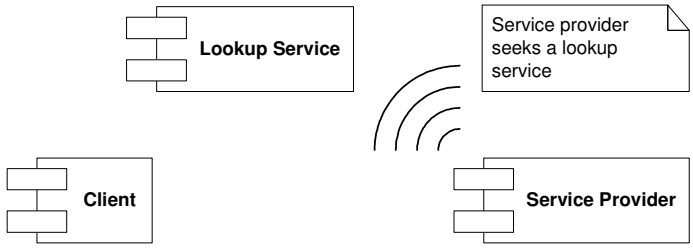


Figure 2.5 Jini Lookup Service discovery by Service Provider [149].

First, the Service Provider searches for the Lookup Service (Figure 2.5) and registers the service object[73] with it. Next, a service object for the service is loaded into the lookup service and the Service Provider *joins* the Jini federation (Figure 2.6).

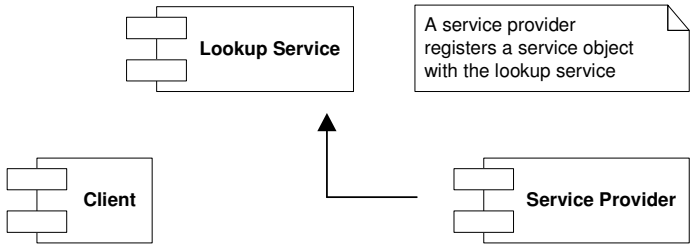


Figure 2.6 Registration of Jini service by Service Provider [149].

The final step of Jini communication is a client request for joining the federation and acquiring the service object (Figure 2.7).

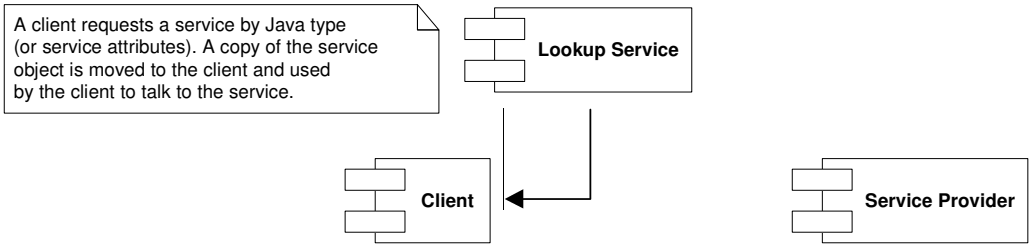


Figure 2.7 Client request for Jini service object [149].

[72] An example of a Jini system providing distributed persistence and a Java object exchange mechanism.

[73] RMI proxy object, with a defined client interface and Service Provider location.

The client uses a similar mechanism of discovery as the one used by the Service Provider for registration purposes. Clients can obtain the Lookup Service location and join the community of Jini entities.

At later stages of the Jini specification process, Jini architecture was extended with the Jiro technology [96], which was a Java-based implementation of the **FMA** (Federated Management Architecture) specification [142]. It provided developers with technologies required to build distributed management solutions, along with the ability to dynamically deploy new services from packages downloaded from remote locations through HTTP. However, the complexity of this technology, overlapping with different areas of Jini[74], caused it to be abandoned in favour of separate Jini and JMX technologies. Jini and JMX took over almost the entire legacy of Jiro[75], especially the lookup service, discovery service and dynamic deployment of software components from remote locations.

2.6.4 JMX

JMX stands for Java Management Extensions. It is a set of specifications introduced by Sun Microsystems at early stages of Java language evolution, and is a standard part of the current version of the Java language since the release of **JDK** (Java Development Kit) 5.0. The goal of the first **JSR** (Java Specification Request) 003 specification [147], was to develop a uniform platform for monitoring and management of distributed resources such as network devices, hardware computational platforms, applications, mobile equipment and many others. The second specification (number 160, [145]) extended JMX with an API for accessing remote resources by means of so-called connector architecture and special protocol adaptors. A key component of this architecture is the MBean, representing a managed resource (any physical or logical object equipped with a Java interface), and the MBeanServer, acting as a container and mandatory mediator for MBeans and providing additional services such as notifications, remote access by means of connectors and protocol adaptors. Since JMX was used for prototype implementation of the proposed monitoring service, JMX will be presented in more detail.

JMX Tiered Architecture

JMX covers three levels of the monitored environment: instrumentation level, agent level, and distributed services level (Figure 2.8). The main concept of JMX focuses on objective resource representation and exposure of their management interfaces, providing remote access and other elements of functionality for managed objects.

[74] Jini specification and Jini framework were part of the Jiro technology.

[75] After 2003 all information about Jiro disappeared from the Internet and Sun's own website although the implementation of FMA continued as a subproject of Jini.

Figure 2.8 shows key components of the JMX platform: client, server (with the MBeanServer) and MBean. Client is an application, which controls servers by submitting requests and activating new services. Server is an application, which makes local resources available remotely and provides one or more services for managed objects. MBeanServer is also a registry for MBeans logically placed inside the server. Managed Bean is a Java class implementing a management interface and representing a resource to be managed or monitored [146] (see Figure 2.10 and 2.11).

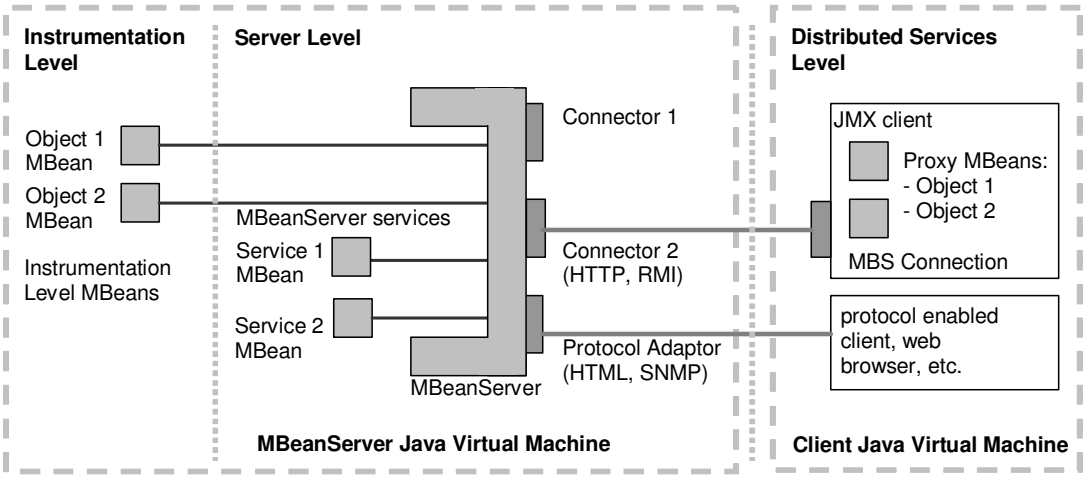


Figure 2.8 Tiered architecture of JMX with connectors and MBean Proxy objects [147].

JMX Instrumentation Level

The instrumentation level contains JMX manageable resources (a service, device or user – specifically, any resource developed in Java or having a Java wrapper).

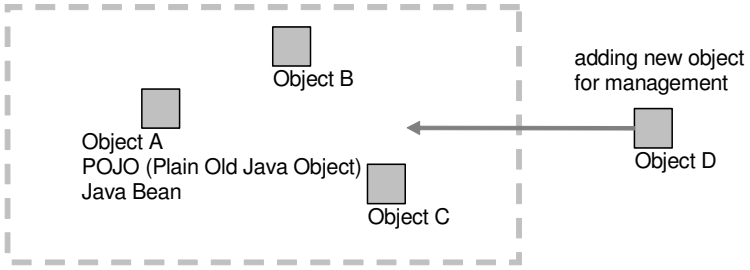


Figure 2.9 Basic concept for Java object management.

Resource representation as an MBean component (simple Java components[76] with well-defined management interfaces) provides unified access to functionality and allows discovery of this functionality at runtime. It also allows us to define a management interface not only at compile time (Standard MBean, Figure 2.10), but also at runtime, through the use of a

[76] JavaBeans, the components defined by the Java 2 Platform, Standard Edition (J2SE) [85].

specialized type of MBean called the Dynamic MBean (Figure 2.11) or the Model MBean. The MBean management interface is a standard Java interface[77], which helps MBeans expose their management interface in terms of attributes, operations, notifications (optional) and constructors. In order to be visible remotely, an MBean must be registered with the MBeanServer. In the monitoring service, MBeans play the role of monitoring sensors and usually reflect the functionality of the available resources.

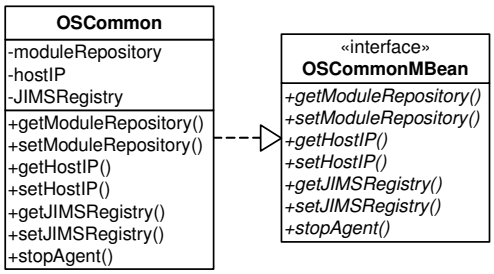


Figure 2.10 Standard MBean management interface[78].

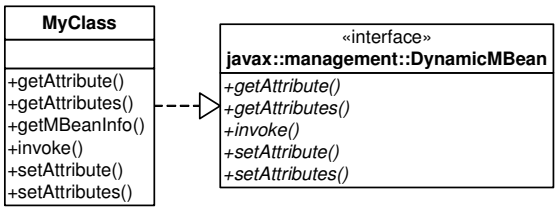


Figure 2.11 Dynamic MBean management interface.

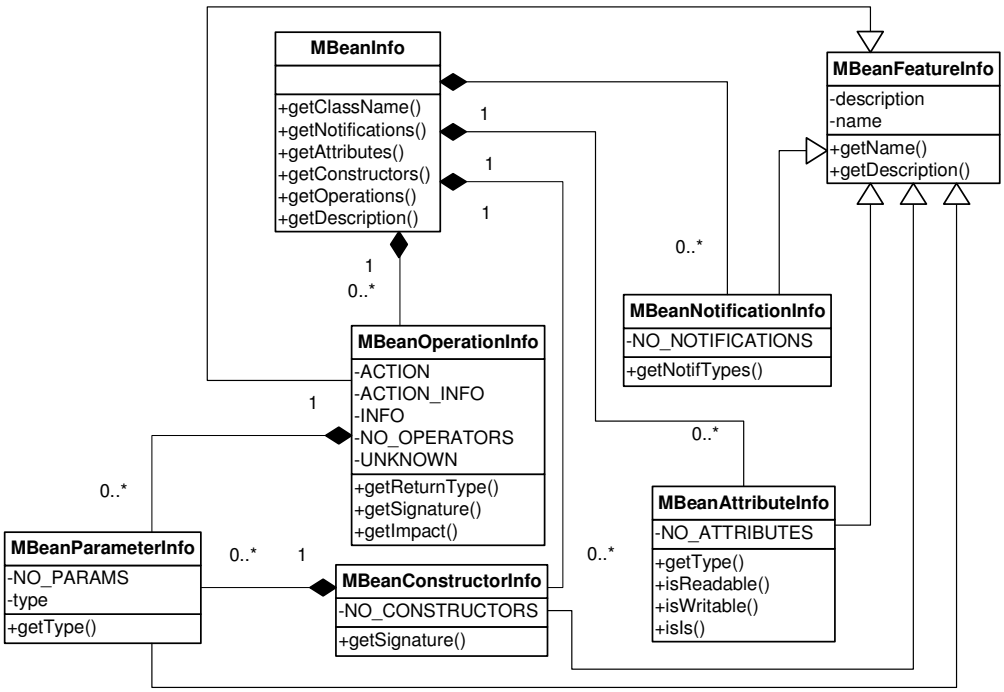


Figure 2.12 MBean metadata classes (used directly in Dynamic MBeans) [147].

[77] The only requirement is the name of the interface, which has to end with the *MBean* suffix for standard MBeans (see Figure 2.10). For dynamic MBeans, the interface is derived from the Java package java.management and is called DynamicMBean (see Figure 2.11).

[78] The presented class of OSCommon MBean is a real example of an MBean used in the described monitored service. It holds common information used by any instance of the monitoring agent, i.e. the Computing Element's IP address (the access node address), the host IP addres and the JIMS Registry value.

In order to define the Dynamic MBean and for introspection of standard MBeans, special classes are used to describe the management interface of an MBean in terms of its attributes, operations, notifications and constructors. These classes are called MBean metadata classes and are shown in Figure 2.12. By using these metadata classes, the JMX agent exposes all its MBeans in a unified way, regardless of their type.

JMX Agent Level

The JMX agent level consists of a JMX server, which directly controls resources and makes them available to remote management applications. This server in turn consists of an MBean server and server services. The MBean server is a registry of objects which are exposed to management operations on the server and only registered MBeans can be managed from outside the server's JVM, while the MBean server only exposes an MBean management interface (instead of direct MBean references) [146].

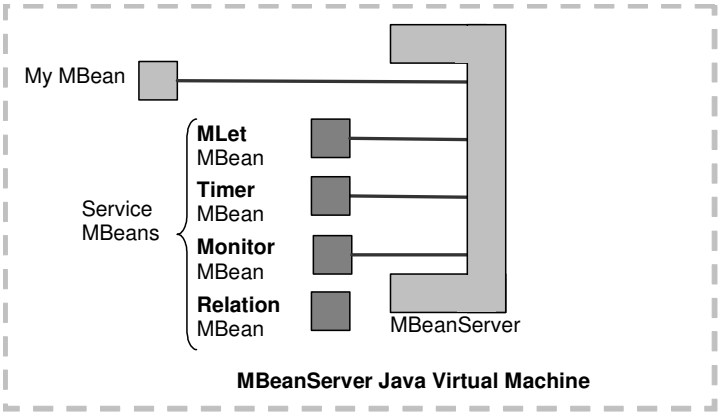


Figure 2.13 Service MBeans in JMX architecture: MLet,Timer, Monitor and Relation.

Figure 2.13 presents the MBeanServer and some service MBeans that provide optional management intelligence inside the server. The following types of service MBeans are available: *MLet* (dynamic class loading service, which retrieves and instantiates new classes from arbitrary network locations), *Monitor* (observes an MBean attribute value and generates notifications), *Timer* (provides scheduling based on one-time or repeated periodic notifications), and *Relation Service* (defines associations between MBeans and enforces the cardinality of the relation based on a predefined relation type).

JMX Distributed Services Level

The distributed services level defines a distribution mechanism, which consists of the following components: connectors, protocol adaptors and proxies. It specifies interfaces for JMX clients and defines management interfaces and components that operate on remote servers. All these modes of access to managed resources are shown in Figure 2.8. Each connector component (such as the RMI connector or the SOAP connector) is composed of a client part and a server part, which connects a JMX client application with an MBean server. The client part provides a protocol-independent API to access the remote MBean server and it

tunnels management operations over specific means of communication such as RMI, HTTP or JXTA. Protocol adaptors (such as the HTML and SNMP adaptors shown in Figure 2.8) only have a server part, placed on the server, and create views of MBeans using a given protocol and information model such as HTTP/HTML or SNMP. The final means of accessing managed resources is to create automatically generated Proxy MBeans, which are shown on the right-hand side of Figure 2.8. A proxy MBean is an image of an MBean, viewed as a stub object on the client side. All operations performed on the proxy MBean are transparently propagated to the MBean, including notifications emitted by the remote MBean.

The presented levels and features of JMX are very interesting from the grid monitoring and management perspective. First, JMX provides means for dynamic loading of classes from remote locations (Java m-let service, short for *management applet*[79]). Second, it provides a very high level of communication abstraction, thanks to its connector/adaptor architecture, which covers almost all underlying protocols and communication systems (RMI/IIOP, SOAP, SNMP, HTTP, JXTA). The distinction between JMX communication and monitoring is consistent with the **SoC** (Separation of Concerns) pattern. Moreover, JMX currently plays a very important role as a general framework for distributed resource management and offers architecture suitable for construction of monitoring systems and management applications written in Java. The spectrum of JMX applicability covers, among others, grid systems [164, 68, 172], active networks [166] and mobile systems. The JMX standard is also suitable for adapting legacy systems as well as implementing new management and monitoring solutions.

2.6.5 JDMK

JDMK (The Java™ Dynamic Management Kit) is a software development kit for management and monitoring of applications written in Java. It includes a commercial implementation of JMX and extends it with additional tools, supporting creation of highly distributed and network-enabled applications, leveraging dynamic discovery facilities, JMX and SNMP management and monitoring. The architectural model of JDMK is similar to the one depicted in Figure 2.14, whilst JDMK only extends this model by adding specialized adapters (e.g. SNMP, HTML), additional connectors (e.g. HTTP connector) and modules facilitating active and passive discovery (discovery clients, monitors and responders). JDMK seems to be an ideal environment for design of grid management and monitoring services. Its main limitation is that it uses proprietary protocols[80] and applies a licensing model, which prevents the author from using these vendor-dependent JDMK features.

[79] According to the JDK specification, *MLet* is a classname, whereas the word *m-let* is used to refer to a service or a MLet descriptor.

[80] For example the protocol encapsulated within HTTP and used by the HTTP connector.

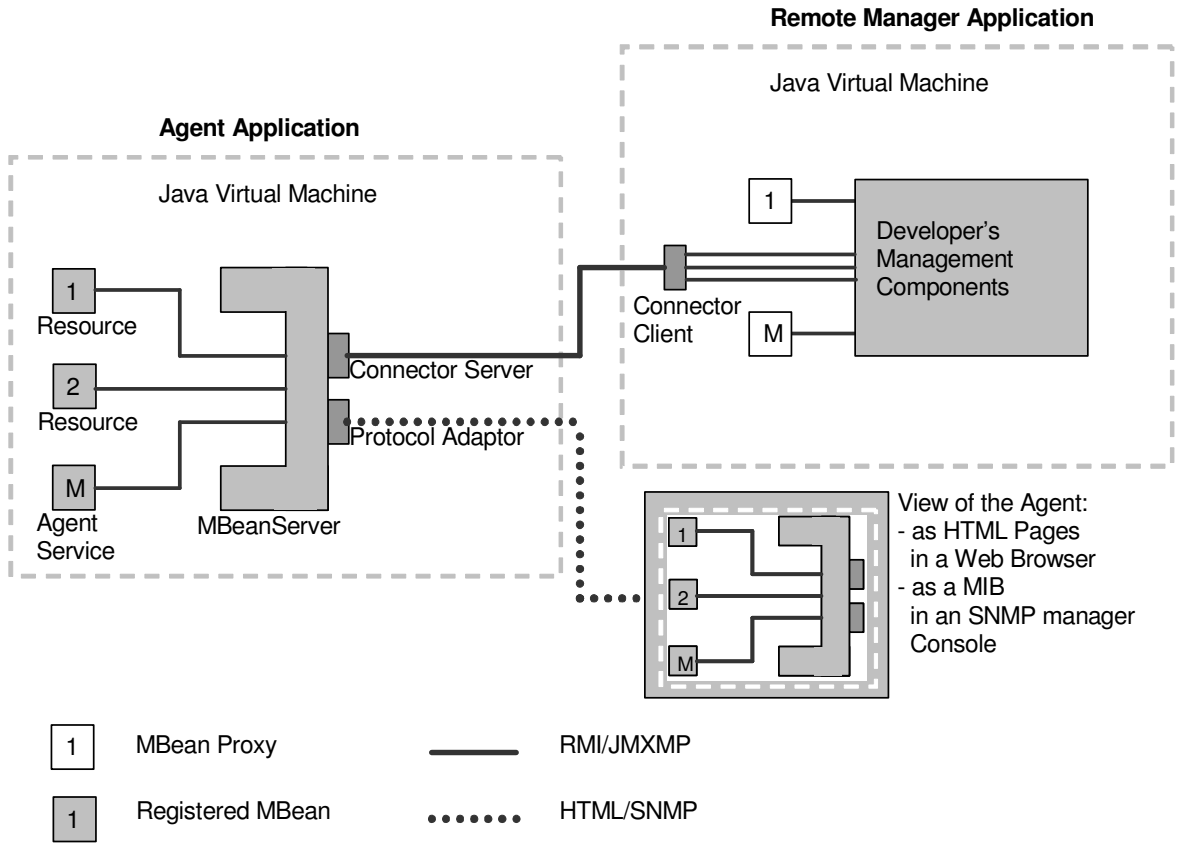


Figure 2.14 Key concepts of the Java Dynamic Management Kit [143].

2.7 Related Projects

In addition to the technologies, which may significantly support constructing a distributed system for grid monitoring, the author would like to mention several projects that seem to be interesting from the point of view of this dissertation. These projects also provide a background for verification of results of the presented thesis.

2.7.1 DARPA Active Networks

The first project that addresses dynamic instrumentation of network resources was a program called Active Networks [166], funded by **DARPA** (Defense Advanced Research Projects Agency). The program commenced in 1996 in USA, at **MIT**, and indicated important requirements of modern telecommunication networks:

- modification and injection of new functionality to network nodes,
- security policies governing the injected code.

The Active Networks initiative covered numerous projects addressing different aspects of dynamically changing networks, including the **JANOS** project (Java-oriented Active Network Operating System [166]), oriented towards building a platform- and language-independent system for deployment of functional modules in network nodes.

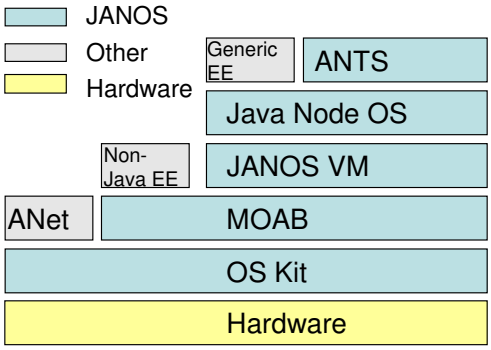


Figure 2.15 The Janos system architecture[81].

Although envisioned ten years ago, all these frameworks address issues, which are crucial for contemporary systems. Objectives that are very similar to the ones addressed by Active Networks can be found in another project called **MVM** – the Multi-tasking Virtual Machine. Led by Sun Microsystems, the project aimed at extending the current **JVM** (Java Virtual Machine) with the ability to execute multiple Java applications, called isolates, within one instance of a JVM on a given operating system. Though the concept proved very difficult to implement and the project was finally abandoned[82], it nevertheless presented interesting ideas for starting multiple Java applications within one instance of Java VM. It can be said that the key goals of MVM remain valid, since there are many other ways of executing separated Java code in parallel, usig for example only simple Java class loaders (this is possible in pure Java or in more sophisticated environments such as JMX or JDMK[83]).

2.7.2 DataGrid

DataGrid is a project funded by the European Union in the years 2001-2003. Its objective was to build the next generation computing infrastructure allowing intensive computation and analysis of large-scale shared databases, from hundreds of terabytes to petabytes, across widely distributed scientific communities [74]. Among other aspects, DataGrid developed specialized services for infrastructure monitoring such as SANTA-G, R-GMA, gridIce, MapCenter, and Mercury[84]. The project resulted in well-integrated, data processing-oriented grid middleware and numerous specialized, reusable grid components.

[81] The OS Kit is an operating system exposing platform-independent interfaces for higher layers of JANOS. MOAB is a C implementation of NodeOS API. ANTS supports dynamic deployment of custom code in active nodes [166].

[82] The project was abandoned in 2006, probably due to low performance gains and high resource consumption when sharing computational power and memory between many isolates.

[83] Java provides a URL class loader which helps load Java code from remote location. JMX and JDMK also support descriptions of the code being downloaded, which makes writing modules easier and gives better opportunities for reconfigurability and reusability of software components.

[84] GMA-compliant grid monitoring service, supporting monitoring of resources and grid applications, with extendable metrics and controls.

2.8 Chapter Summary

The chapter presented standards, protocols and systems for resource representation, management and monitoring. It should be noted that these frameworks follow the notion of using component (or modular) architectures and SOA/WS as a grid integration technology. Furthermore, in the area of frameworks for grid infrastructure monitoring, we can observe a very high level of specialization, which is not surprising, given the complexity and scale of modern grids. We also presented monitoring tools commonly used in current grid systems. Work on these systems continues and is an essential part of many ongoing EU research projects[85].

From the presented frameworks and technologies, the author chose the JMX platform as a basis for prototyping the proposed grid monitoring system. First, JMX has a simple architecture, which supports the concept of simple **POJO** (Plain Old Java Object) objects, equipped only with management interfaces and called MBeans. Secondly, the JMX connector architecture allows remote access to monitored resources, which are agnostic of communicational issues. Moreover, the JMX agent layer hides the location of real monitored resources and allows remote access using any available types of transport protocols such has RMI, SOAP, Burlap, Hessian, etc. JMX also offers dynamic loading functionality, which is an extension of Java class loaders. Built on the top of the Java platform, JMX seems to be a flexible platform that can prove useful for implemenation of grid monitoring services. It is widely accepted by the industry and has become a *de facto* standard for management and monitoring of enterprise-class systems over the last years[86].

[85] Among others, this research is continued in EGEE, EGEE-II and int.eu.grid (Interactive European Grid [57]) projects.

[86] It is used in JDK 5.0, application servers (i.e. JBoss [124]), specialized **DBMS** access systems (i.e. Hibernate [25]), etc.

3 System Requirements and Problem Statement

This chapter describes conditions for building a grid service in general and presents requirements for building grid infrastructure monitoring services in particular.

This chapter focuses on the problem of building a grid service for infrastructure monitoring in a highly changeable and geographically distributed environment. Requirements for building such a service result from the general conditions common to all grid environments and are complemented by requirements stemming from the grid monitoring domain. Standardization processes and conditions present in real grids have significantly influenced the scope of the problem addressed by this thesis.

3.1 General Grid Characteristics

General requirements for grid services result from common grid characteristics such as heterogeneity, usage of any available resources[87] and focus on the user [128]. Design and implementation of grid services poses many technical and socio-political challenges, which should be overcome in order to succeed in proposing a new grid concept.

3.1.1 Evolution of Enterprise Computing Towards Grid Systems

In order to better understand the changes that are currently occurring, let us see how computational systems have evolved over the recent years. Figure 3.1 (a) shows the first systems that supported computing-intensive applications. They were built on top of centralized hardware called mainframes and provided vertical scaling of computational power[88]. In the age of network development, this architecture was decentralized, providing separated workload management systems for applications running on many hardware

[87] Resources are seen not only as computers and networks, but also as data management, etc. This problem concerns locating data, storing terabyte data sets, replication of data and data access over networks [58].

[88] Vertical scaling means the possibility of adding new processors and other resources to a mainframe.

platforms, as shown in Figure 3.1 (b). Such architecture was a very significant step forward as it allowed horizontal parallelization of applications on many nodes[89]. The last stage of the described evolution involved reintegration of these distributed systems in order to provide vertical decoupling of workload management and physical resources so that different components could be assembled dynamically to meet the needs of applications (see Figure 3.1 (c)).

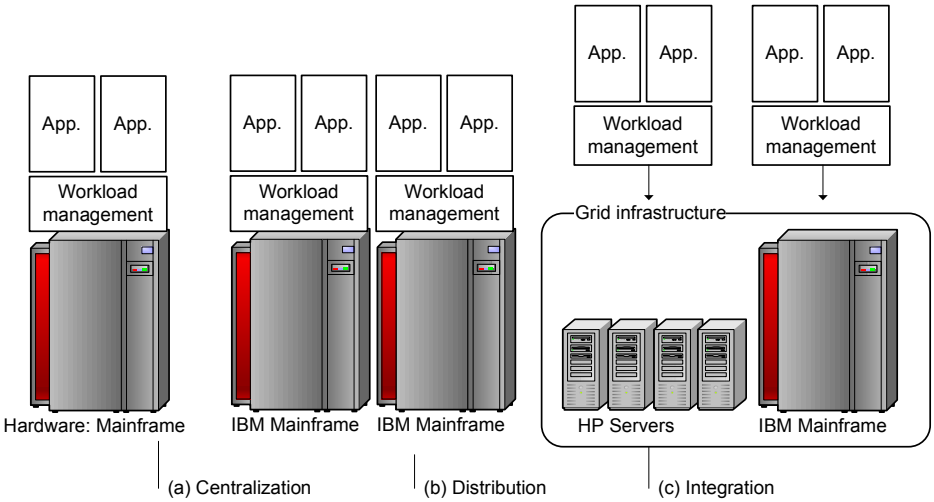


Figure 3.1 Evolution of enterprise computing [69].

An important outcome of this evolution is the cluster-based and hierarchical architecture of modern grids. In such an infrastructure, workload management systems and clusters are loosely coupled, which has strong implications for the architecture of the proposed monitored system.

3.1.2 Basic Requirements for the Grid

A detailed definition of the grid was already presented in section 1.3.1. J.M. Schopf and B. Nitzberg in their *Top Ten Questions* [128] enumerate ten problems that should be addressed by the grid. Among others, they state that (i) the design of any additional grid functionality (like interactivity[90]) has to be preceded by first providing working functionality at the basic level[91], (ii) the target grid community are real application developers and users who use the grid for everyday work[92], (iii) the users need tools that really facilitate the use of a grid, (iv)

[89] Horizontal scaling means the possibility of adding new computational nodes (here: mainframes) to the existing computational infrastructure.

[90] A/N.

[91] This seemingly simple assumption is very hard to implement in reality, due to the complexity of modern grids.

[92] Most current grid applications are custom-developed for particular grids and chiefly driven by scientific institutions, due to the relative lack of standalone grid users. However, this problem has already been addressed by new grid projects such as EGEE and EGEE-II.

deployment and software setup in the grid should be as easy as possible, (v) the users need a reliable and an easily used security infrastructure, (vi) the business model of the grid should show its benefits, encouraging people to use it, (vii) the grid needs standard interfaces and definitions since the lack of standardization continues to hinder the interoperability between different systems, (viii) the changeable behaviour of the grid must be specifically addressed.

Many of these issues are addressed by current grid concepts, specifications and frameworks that have evolved along with distributed computing. They range from the simple concept of SOA (introduced by Gartner in 1996 [130]), to more grid-oriented specifications such as OGSA/OGSI, defined by GGF, as well as WS-RF, WS-DM and WS-Management[93] specifications, introduced by the Globus Alliance, OASIS and DMTF.

3.2 Requirements for a Grid Monitoring Service

The design of a monitoring system, which would fulfill the specific needs of target applications (for example detailed online monitoring of interactive jobs), requires careful analysis which adheres to general recommendations for building grid monitoring services. Most of these general requirements for grid services originate from existing standards, driven mainly by OGSI, WS-RF, and WS-DM.

3.2.1 General Requirements for a Grid Monitoring Service

The GMA architecture [161], one of the most popular architectures for grid monitoring systems, announced in 2002 by GGF Performance WG, states that a grid monitoring service must address the following grid characteristics:

1. be scalable across wide-area networks and able to encompass a large number of heterogeneous resources,
2. its naming and security mechanisms must be integrated into the grid middleware and be able to protect sensitive data,
3. should be characterized by low latency, as performance data is typically relevant for only a short period of time,
4. should be prepared for performance data which can be generated at high data rates,
5. should introduce minimal measurement overhead,

[93] WS-Management is a competitive to WS-RF specification, developed by the DMTF consortium. It aims at development of a general SOAP-based protocol for managing systems such as PCs, servers, devices, Web services and other applications as well as other manageable entities [55]. WS-Management is not discussed in this thesis because it does not rely on standards which form the basis of the WS-RF framework, widely used for building grid services.

6. should be scalable, because there are potentially thousands of resources, services, and applications to monitor and thousands of potential entities that would like to receive this information.

Moreover, performance-monitoring information produced by a monitoring system can be distinguished from information produced by other types of systems:

1. performance information has a fixed, often short, lifetime of utility,
2. updates of monitoring information can be very frequent (unlike the more static forms of "metadata", dynamic performance information is typically updated more frequently than it is read),
3. performance information is often stochastic, while it is frequently impossible to characterize the performance of a resource or an application component by using a single value.

All these general requirements should be taken into account during the specification phase for any grid monitoring service.

3.2.2 Functional Requirements

Based on these general requirements for grid monitoring systems, the author would like to present additional requirements addressed by this dissertation and forming the motivation for building a system with the capabilities described in the thesis.

Monitoring Methods: Pulling, Tracing and Notifications

Infrastructure or application monitoring can involve different types of users: regular system users, monitoring system administrators and other grid components (such as resource brokers, etc.) Use cases for these actors can include obtaining grid information in a request/response pattern, tracing logs of monitored entities and subscribing to notifications of monitoring events (such as addition or removal of new resources, reaching some monitoring thresholds, etc.) The first method, called *pulling*, is a sampling method initiated at the client side (information consumer)[94]. The second one, *tracing*, involves log analysis. It is used for finding information about events that were recorded in the database in the past. The last method, called *pushing*, relies on periodic status reporting (sending *notifications*) about the state of monitored resources (Figure 2.4, the *notify* arrow). This type of monitoring is typically initiated by monitoring agents (information producers). Monitoring system elements used to obtain information about monitored resources are typically called sensors or software probes (the two terms are used interchangeably).

[94] Similarly to the scenario shown in Figure 2.4 (the get arrow).

Target Elements of the Monitored Infrastructure

The following elements are expected to be monitored by the grid monitoring service:

1. grid structure as a whole, taking into consideration reaction to possible changes,
2. all grid clusters, taking into consideration reaction to possible changes in any single cluster,
3. resources of all computational nodes (hardware, CPUs, memory, hard disks),
4. network infrastructure (network interfaces, bandwidth utilization, latency),
5. grid middleware such as job management systems, etc.,
6. applications, services and servers.

In addition, the monitoring service should provide possible a wide spectrum of accessibility to monitored resources and be accessible by means of an API, a CLI and a GUI (standalone or web-based applications).

Elements of Adaptability

Adaptability is one of the self-management system features and is the system ability to change its behaviour alongside the changing type of environment. Author believes that grid-monitoring service should be equipped with such ability to adapt to different hardware and software platforms without the need for manual reconfiguration and administration during the system startup and at runtime. Adaptability in this sense includes the ability to detect the properties of the surrounding environment and to adjust the available functionality to these detected requirements. Among other things, expected adaptability elements should follow common properties of contemporary grid infrastructures, i.e. differentiated hardware platforms[95], types and versions of the operating system[96], versions of the operating system

[95] Common hardware platforms used in grids are based on CPUs with Intel architecture, with both 32 and 64 bit architectures (i.e. i686 and Intel Itanium 2), in single or multiprocessor configurations. Grids are also known to include computational nodes with different CPU architectures (CISC, RISC) for common UNIX systems, such as N1 grid, running on both Sun UltraSPARC® and Intel® Xeon™ processors used in SunFire B1600 Blade Systems.

[96] Common operating systems used in grids are based on various Linux and Unix distributions: Red Hat (including RH Enterprise Linux distribution), Scientific Linux/DESY (Linux systems), FreeBSD, Sun Solaris (UNIX systems) etc.

kernel[97], versions of the IP protocol[98] as well as types and versions of software installed in the grid[99].

Extendability

For development, debugging and administrative purposes it is expected that the initially loaded functionality can be changed at runtime and that administrators can managed the lifecycle of the loaded modules (i.e. to perform their deletion or creation). Such a system should provide means for profiling, runtime instrumentation and resource management, which is extremely important in highly changeable grid environments, and which would enable runtime modification and deployment of sensor modules, giving the possibility to add or upgrade their functionality without restarting the monitoring system. All these operations should be easily accessible by the system administrator, and should not require developer intervention.

Monitoring Service Self-Configuration

Configurability is a next self-management system feature and is the ability to organize the monitoring agents' instances (concerning their number and roles). Such ability is called the self-configuration and it involves: (i) discovery and registration of new computational nodes, (ii) discovery and registration of new clusters, (iii) discovery and registration of newly initiated applications anywhere in the grid.

Self-configurability complements elements of adaptability described before (and intended for use during startup), and is a long-term monitoring service runtime feature. The self-configuration facility should be designed and operated in a manner, which does not affect monitoring system operation, therefore minimizing the downtime typically required for reconfiguration. According to the two-level architecture of contemporary grids, consisting of clusters connected into one, large computational grid, two levels of self-configuration are foreseen: cluster-level self-configuration and grid-level self-configuration. Though both configuration facilities involve the discovery of computational resources, the difference is in the granularity of the discovered resources: on the cluster-level, they are discovered and registered as computational nodes, while on the grid level entire computational clusters are discovered and managed. Each of these two scenarios is characterized by specific

[97] For example, the system should distinguish between two main Linux kernel version series: 2.4 and 2.6, which provide different interfaces for monitoring and management through the */proc* **VFS** (Virtual File System).

[98] Current implementations of monitoring systems should allow communication using both IPv4 and IPv6 protocols, since IPv6 is being introduced alongside current IPv4 infrastructure, which is a common scenario in many autonomic systems and grids (see for example the Clusterix project [95], which is oriented towards IPv6).

[99] Such adaptability can concern for example the presence and type of queueing system installed in the grid (i.e. Open PBS, Torque, SGE, etc.) It can also concern the installed version of the Java Virtual Machine, i.e. 1.4, 5.0 or 6.0, each of which introduces different scopes and types of management and monitoring interfaces.

requirements, related in nature to **LAN**[100] and **WAN**[101] networks in which the monitored resources are available.

Cluster-Level Monitoring and Self-Configuration

Computational resources in a typical grid cluster are composed of many computational nodes and other deditacated nodes such as access nodes, storage elements, etc., typically interconnected via a LAN. The configuration of these resources can change over time, and this should be reflected by the monitoring system, including in particular:

1. addition of new computational nodes,
2. removal or failure of existing computational nodes,
3. network disruption, resulting in temporary unavailability of existing nodes.

An important enhancement of this self-configuration facility should be the ability to send notifications concerning any changes in the list of available and monitored nodes. The system should be also equipped with self-monitoring for easy integration with other grid components, development and debugging. Finally, the monitoring system should address the following technical aspects of the local cluster environment:

1. monitoring of ISO/OSI layer-III and layer-II network devices[102] using the SNMP protocol,
2. monitoring of network performance, i.e. available bandwidth or network latency,
3. monitored nodes in local clusters should be able to use private addressing spaces[103], preventing them from being publicly accessible from the Internet – this presents another challenge on the road to making monitoring information publicly available,
4. the system should provide basic security in terms of authentication and authorization, to protect sensitive monitoring data.

Grid-Level Monitoring and Self-Configuration

Requirements for grid-level monitoring are similar in nature to requirements for cluster-level monitoring, besides the fact that grids are typically deployed over WANs, and hence the monitoring service should collect coarse-grained information.

Requirements for grid-level monitoring can be specified as follows:

---

[100] Local Area Network, typically a broadcast network based on IEEE 802.3.

[101] Wide Area Network, typically a non-broadcast network based on a specialized set of point-to-point protocols.

[102] According to the OSI/ISO network reference model – routers and switches, respectively.

[103] IP protocol version 4 specigfies three private addressing spaces: 10.0.0.0/8, 172.16.0.0/12 and 192.168.0.0/16, which are not intended to be routed to the Internet.

1. collecting information about the availability of particular clusters,
2. autonomous operation, i.e. the system should not require administrative intervention and should adhere to critical situations which can occur in the network (node failures, network disruption),
3. lack of any central repository storing information about available clusters (to avoid introducing a so-called “single point of failure”),
4. dynamic discovery of clusters newly added to the grid,
5. handling unavailability, failure or removal of a cluster,
6. handling temporary unavailability of a cluster due to network disruption,
7. proper handling of restarts of more than one cluster (including a situation where all the clusters are simultaneously restarted),
6. support for notifications about all the changes listed above,
7. security mechanisms with authentication and authorization,
8. self-monitoring of system operation and exposing system state to the administrator.

3.2.3 Non-functional requirements

The functional requirements presented above are supplemented by additional non-functional requirements, resulting from the specific features which distinguish grids from other distributed systems.

Uniform and Interoperable Interfaces

The interface to monitored resources should stay the same regardless of the protocol used to access monitoring information and regardless of the changes in configuration at any level of grid monitoring. The interfaces should also be sufficiently flexible to handle functionality, which is not known at the time of monitoring system compilation or even at runtime, allowing the administrator to load such functionality, which can become available at any point in the future.

In order to fulfill the latest WS-RF specification requirements, the functionality of the monitoring system should be made available using WS. Any communication among clusters inside the grid, as well as communication between the client application and the grid monitoring system should be also performed using WS. In particular, WS should be used to solve interoperability issues. Additionally, WS should allow remote management of monitored resources and their discovery (in the scenario of grid-level self-configuration). WS should be used for sending notifications about newly added or removed computational clusters, as required by management applications.

Component-Based and Coherent Resources Representation

The monitored resources such as infrastructure and applications, as well as their internal functionality, should be represented in a coherent way, using a component-based approach. It should be noted, however, that component-based representation should not hinder the scope of the monitored grid resources or any other functional elements of the monitoring system itself.

Possibilities of Future Extensions for Infrastructure Monitoring

The design of the infrastructure monitoring system should be open for future extensions. It should allow loading monitoring sensors during startup and be open for modifications of the loaded functionality during runtime.

Possibilities of Application Attachment for Monitoring

The monitoring service should allow administrative or automatic attachment of monitored applications. Any application exposing a management interface[104] should be able to connect (administratively or automatically) to the monitoring system[105].

Non-Intrusiveness

It is important to notice that a monitoring system for a cluster with numerous nodes has to overcome performance issues which are not present in a supercomputer with the same number of nodes: “Taking for example a 512-processor supercomputer, we can use only one copy of the monitoring service, while monitoring a 512-processor cluster with, for example, 256 dual CPU nodes [requires using] 256 copies of the monitoring service” [100]. We are also forced to use the network bandwidth for transportation of monitoring data between nodes and for monitoring system administrative tasks. This is why the architecture of the monitoring system for the grid introduces strong performance issues if not designed properly. System architecture should therefore minimize intrusiveness at both the level of computational resources (CPU, memory) and network bandwidth utilization. Building a non-intrusive[106] system is one of the main design goals of the project.

Automatic Installation

Software management in grids involves the use of specialized tools (such as LCFG [7] or Quattor [120]) for software package installation and configuration. Such tools require that

[104] The application management interface is a programmatic interface which allows monitoring and management of its component modules.

[105] As an interface for management of a certain type of applications, JMX can be utilized, with extensions for remote management and monitoring [147,145].

[106] Non-intrusive means intrusive at a level which is not measurable within the chosen range of values of monitored parameters.

services designed for installation in grids are packaged as special archives, e.g. RPM, with special pre-installation and post-installation scripts, well-defined dependencies on other packages and well-defined scripts for startup and shutdown[107]. Since installation and configuration of the service is automatical, further customization[108] of the installed monitoring sensors should be performed using additional administrative tools during runtime. Such functionality would allow easy upgrading of monitoring sensors without the need to restart completely the monitoring service.

Scalability

The ability to efficiently handle a growing number of computational resources as well as simultaneously connect many clients to the same monitored resource is another non-functional requirement for the proposed monitoring service. Scalability of the monitoring system can be achieved with a hierarchical structure of monitored resources and should be designed in a manner, which allows future addition of new monitoring resources.

Portability

As far as monitoring services are concerned, the portability feature has a threefold meaning and can be understood as the ability to:

- install the monitoring service on a possibly wide spectrum of hardware and software systems,
- monitor a possibly wide spectrum of hardware and software systems,
- ensure that installation of the system does not depend on any specialized grid middleware[109].

The number of lines of code, which is **OS**[110]-specific, should be minimized and all such code should be packaged as separate modules. On the other hand, the common part of the monitoring system should be designed in such a way as to be reusable (without recompilation) whenever moved from one system to another.

[107] Under Linux and Unix systems they are called *init.d* scripts and are invoked with the *start* parameter when the operating system starts up and with *stop* during its shutdown process.

[108] Configuration, loading and removal of additional monitoring modules.

[109] One of the most common middleware packages used in grid is the Globus Toolkit (version 2, 3, or, currently, 4). However, the author aims to design a system which would be independent of such middleware in order to be able to install the monitoring service on non-GT-based systems, like generic N1 grids. Moreover, specifications proposed by GGF/OGF are constantly changing (from plain Web Services to OGSA/OGSI and then to the WS-RF framework), making development of GT-compliant systems very difficult.

[110] Operating System.

Fault Tolerance

The system should tolerate possible failures of nodes and clusters. It should also properly handle unavailability of monitored computational resources caused by to network conditions. Recovery from all these critical situations should not require administrative intervention.

Open Source and Freely Available[111]

Components and technologies used for building the monitoring service should preferably be available as FLOSS software in order to avoid negative impact on the licensing model of other software installed in grid testbeds. Such a requirement is common in many European IST projects (such as DataGrid or CrossGrid) and seems to be a condition, which should be fulfilled by any software intended to be applied by the wide community of grid users and developers.

Security

The system should provide at least basic security mechanisms for authentication and authorization and remain open for integration with common security architectures, supporting authentication of clients and applications as well as authorization for accessing monitored resources.

3.3 Problem Statement

The definition of our problem results from the need to monitor grid infrastructure in a way that adheres to the general grid characteristics (section 3.1) and concerns building a grid monitoring service that addresses all the requirements enumerated above. The problem of building such a system can be split into following groups of problems:

- self-configurability and resource discovery for dynamically changing topologies of monitored computational nodes and clusters, subject to random emergence and disappearance,
- elements of adaptability in order to monitor various types of hardware and software, requiring specialized types of sensors, especially in a network of desktop systems where the diversity of software is very high. Such networks can include all UNIX-type systems such as Linux and Solaris, different CPU types (i686, IA64, Sun Sparc), different kernels (kernels in different Solaris revisions, Linux 2.4 and 2.6 series kernels), third-party monitoring software (SNMP, DRMAA[112]), etc. Elements of adaptability enable dynamic adjustments of monitoring functionality to the requirements of the current environment,

[111] In the sense discussed earlier.

[112] Distributed Resource Management Application API [123], typically used to obtain monitoring information concerning job management systems.

- interoperability between agents in different systems and between the monitoring system itself and other grid services.

Having defined these requirements, the author makes some architectural assumptions such as component-based, coherent resources representation and a hierarchical architecture, in order to verifiy, whether the system built according to these assumptions can possess all the required characteristics described above. It should support most common monitoring methods such as like push and pull, be adaptable to the environment it is deployed in, provide self-configuration in order to free the administrator from trivial tasks, and be extendible enough, to monitor not only the grid fabric (i.e. the hardware), but the network, other middleware components and even regular applications. Having defined the problem, we will devote subsequent chapters to presenting the concept and design of the proposed grid monitoring service prototype.

3.4 Chapter Summary

Analysis of the development of specifications and frameworks dedicated to resource management and monitoring results in the conclusion that significant effort is being applied by standardization bodies to interoperability issues and exposing uniform management interfaces of managed resources in the form of WS. We can see that the design of a grid monitoring system poses many challenges, both functional and non-functional in nature. Among other things, a system, which would not require additional administrative effort during operation, has to involve some self-management features for adaptability and self-configuration, both in the scope of a single cluster and the entire grid network. Modern monitoring services also exhibit significant emphasis on extendibility, uniform management interfaces and interoperability.

4 JIMS Concept and Design

The chapter presents the concepts and solutions, which are the result of precise analysis and numerous iterations addressing the problem of building a grid monitoring service. We will present basic concepts and key architectural elements of the system along with an in-depth analysis of solutions addressing its adaptability and self-configuration problems.

This chapter describes the concept of JIMS (JMX-based Infrastructure Monitoring System). Though the JIMS system concept (as its name suggests) grew out of implementations based mostly on Java and JMX, the author is confident that these concepts have a broader scope and therefore should be presented in a technology-agnostic way. We will therefore present basic concepts and describe four architectural layers of the system. First, we will focus on the resource abstraction and instrumentation layer. Next, the interoperability layer will introduce the concept of a SOAP Gateway and discovery service developed for resource discovery in local clusters (LANs). Following this, the clusters integration layer will present chosen methods for solving the problem of global discovery in wide area networks (WANs). A simple scenario with a fixed set of global registries is presented, along with a more sophisticated scenario with a dynamically changing global registry. Finally, we will discuss the client application layer, focusing on possible client interfaces, ranging from API to CLI and finally to GUI applications. The following section will include a description of grid monitoring system dynamic self-configuration mechanisms, presenting a unified approach for exposing various grid resources, which enables dynamic discovery and uniform monitoring of both grid infrastructure and applications[113].

[113] This topic is covered in more detail in chapter 5, in the case study concerning applications running within JVM with the JMX management facility enabled. However, application monitoring does not have to be limited to the Java platform and JMX.

4.1 Basic Design Concepts

Before a solution, which addresses the problem stated in the thesis, is presented, we need to introduce basic concepts taken into account during system design. The first assumption is to have one constantly running monitoring agent per each monitored node with a certain set of predefined basic services, in order to conserve resources and simplify the monitoring service model. The second assumption is the agent's ability to dynamically load a monitoring module, which would enable system runtime extendibility and adaptability. In order to simplify the development of new custom sensor modules, the agent should also provide a reasonably high level of abstraction of the deployment environment, covering both hardware and software resources (i.e. OS and network details). The interfaces for monitoring and management should be clearly defined and capable of future extensions. For maintenance purposes, all these assumptions should be realized with focus on service non-intrusiveness and according to such architectural patterns as SoC as well as decoupling the problems of communication and monitoring. The last assumption is the establishment of a reasonable trade-off between performance, flexibility and low resource consumption.

4.1.1 Monitoring Agent with Base and Extended Functionality

The typical approach to building a grid monitoring service involves the design of a special software agent[114], similar to the agent found in networks managed by SNMP, which would perform a mediating role between the user interested in obtaining monitoring information, and the monitored infrastructure. In the proposed monitoring service, this agent plays a very important role, additionally acting as a container for deployable modules and as a collection of facilities supporting distributed monitoring and management. Such an *agent*, equipped with monitoring modules, is called the *monitoring agent*: it is responsible for gathering monitoring information and exposing it the to management station.

The services exposed by the monitoring agent can be divided into two parts: base, generic services, supporting the operation of the monitoring agent itself, and additional, specialized services, strictly supporting monitoring. The first type of services is supposed to be provided by the implementation framework and is expected to support handling and dissemination of events, dynamic deployment mechanisms, timing services and other helper services (parameter monitoring, module dependency). This basic functionality encompasses:

1. connector architecture[115] as a basic method for monitoring remote agent access,
2. discovery module to accomplish system auto-configuration,
3. a component facilitating dynamic loading of monitoring modules,

[114] The simplest and shortest definition presents the **agent** as a component that connects the management system with the resources being managed.

[115] Connector architecture is further discussed in p. 2.6.4 and 2.6.5 in the context of JMX and JDMK.

4. an informational component, providing information common to all types[116] of monitoring agents[117].

The second type of services supports monitoring modules and includes functionality required by the monitoring agent. A brief description of sample monitoring modules, developed for the proposed monitoring service, is included in section 5.6.

4.1.2 Surrounding Environment Abstraction Layer

Another goal of the proposed monitoring service is to provide the highest possible level of surrounding environment abstraction, which would ease the development of sensor modules and shorten the time required for implementation of common tasks[118]. This high level of abstraction can be achieved by providing a layer of middleware, playing the role of a container for the modules being developed. The environment abstraction layer is placed above the available platforms, which execute intermediary code, such as JVM and the .NET framework. Without such a layer, the developer would have to solve the same problems each time (including installation and initialization of modules for different operating systems and hardware versions).

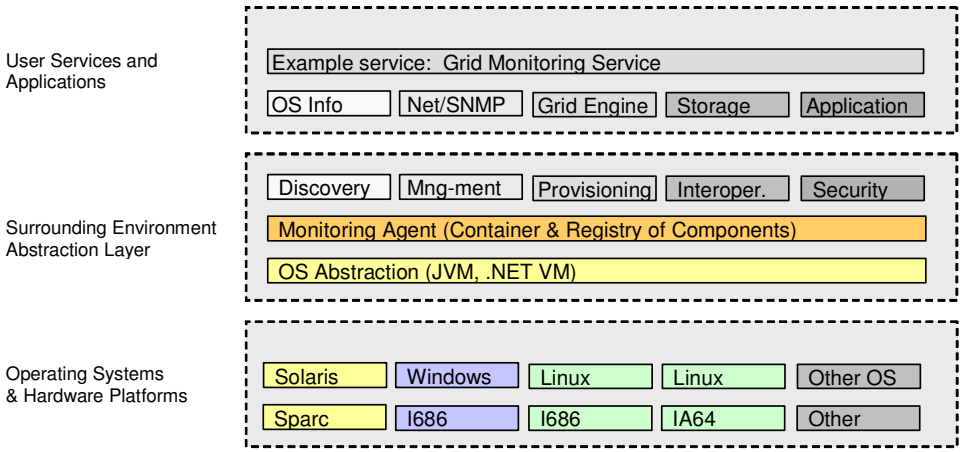


Figure 4.1 Surrounding environment abstraction layer [20].

A simple scenario, which illustrates this problem, concerns the agent's IP address. In Java, this address can be obtained through the *InetAddress* factory class, which, theoretically, should work in all possible systems (Windows, GNU Linux, SunOS, etc). However,

[116] Agents with different modules loaded are considered different due to the different roles they play in the monitoring service.

[117] This information should include some invariant environmental parameters, including the agent's network IP address, IP address of master host (whether it is its own address or that of its neighbour), and the value of its registry (see p. 5.3.1). All these parameters can be used by other agent modules, so it can be assumed that this kind of information is always available.

[118] Experience shows that almost every application of the monitoring service requires design of specialized monitoring modules, which is why the problem of fast module prototyping is of such high importance.

experience shows that this simple solution does not work correctly when each computational node has:

1. many physical interfaces,
2. many logical interfaces created on the top of physical interfaces,
3. many virtual networks configured[119],
4. source and destination network address translation enabled[120].

The problem results from the way the virtual machine obtains its network address using some arbitrarily chosen algorithm. Though in many applications this algorithm will provide correct information, in the grid it has to be customized, based on the table of interfaces available in the operating system, the routing table configured, and the network protocols installed[121]. Given a special module store this kind of information (in this case its IP address), the developer can safely focus on the real issues the module aims to solve, rather than on the trivial tasks related to solving the same problem, such as finding its proper IP address.

The proposed architecture, which provides such a surrounding environment abstraction layer, is presented in Figure 4.1. The lowest layer covers different hardware platforms and operating systems. Placed above this layer, virtual machines (such as JVM or the .NET framework) provide an abstract layer for execution of intermediate code. As a second sublayer of the abstraction layer, the monitoring agent acts as a container for modules implementing the base agent's functionality: discovery, management, provisioning (dynamic loading), interoperability and security. The highest layer includes monitoring modules for operating systems, grid engine management, adaptation to legacy devices via SNMP or active network measurements, etc. On top of the highest layer, we present a sample application of the proposed monitoring service, as a grid monitoring service.

4.1.3 Decoupling of Communication and Monitoring Concerns

The separation of concerns involving information retrieval on the one hand, and communication, discovery and monitoring on the other, means that the type of monitored

[119] According to the IEEE 802.1q standards, Virtual Local Area Networks (VLANs) are used for splitting an Ethernet network into logical subnets called virtual networks, distinguished by special tags. Such networks can be considered as independent physical networks, where communication can be assured by higher layers in the ISO/OSI reference model (VLAN routers).

[120] SNAT and DNAT – Source Network Address Translation and Destination Network Address Translation systems. These types of systems are widely used in connecting local networks with private addresses to WANs (SNAT translation), and, conversely, to connect machines with public IP addresses to private networks (using DNAT translation). Such translations provide a higher level of security, hiding private network addresses, but also generate many problems for distributed software systems, because typical NAT services do not work on all seven layers of the ISO/OSI reference model and do not translate all aspects of communication between distributed applications.

[121] The agent should distinguish between standard interfaces configured for plain IEEE 802.3 traffic and for tagged packets according to the IEEE 802.1q protocol.

resource does not affect the method of discovery and monitoring strategies. Such a separation of communication and monitoring concerns is possible thanks to the connector architecture, as it allows grid users to access monitored parameters via monitoring agents without any support for communication on the part of monitoring modules. Monitoring sensors are not aware of being remotely accessed, which helps building a system that is easy to develop and maintain, and remains open for future extensions, since sensor module developers do not have to bother with communicational issues.

4.2 Hierarchical Architecture of JIMS Monitoring Service

Basing on the requirements elaborated in Chapter 3 and the basic design goals described above, the author would like to present the concept of a JIMS self-configurable service for monitoring infrastructure and applications in a grid environment. The concept relies on a component-based, coherent and uniform approach to resource representation, assuming that resources are represented as software components. Leveraging the unified instrumentation layer, the concept of a monitoring service offers built-in mechanisms for adaptability by means of a centralized modules repository[122] and dynamic service loading, self-configuration facilities for finding monitoring agents in local and public networks[123], allowing resource discovery and registration, and, finally, monitoring sensors, which are developed with focus on their functionality, strictly separated from communicational concerns. The concept also assumes that every node will support exactly one monitoring agent, which contains some basic facilities (such as connectors[124]) providing remote access to the running monitoring service, dynamic component loading facility and, optionally, other helper modules, such as protocol adaptors[125] for service visualization, development purposes or management[126]. Each time the agent is started; this basic functionality is extended and adjusted to the current monitoring requirements. Next, the agent loads proper monitoring modules, including sensors for hardware and software infrastructure monitoring[127], SNMP proxy[128], network sensors or queuing system (like **PBS**[129] or **SGE**[130]) monitors.

---

[122] Available for example by HTTP.

[123] This functionality is called the discovery service and is responsible for finding nodes that are currently running in the cluster. The second part of the discovery service is responsible for locating clusters in a wide area network and is called the global discovery service.

[124] A connector is a software entity deployed in an agent and allowing remote communication with the agent. It enables remote application to access the management interfaces of other monitoring modules.

[125] A protocol adaptor is a software entity that translates one protocol into another. For example, the HTML protocol adaptor is a component that provides a HTML interface for the monitoring agent and other components and translates HTTP requests into local calls inside the monitoring agent. It enables users to change monitored parameters and invoke monitoring agent management methods. Another example is the SNMP adaptor, a component that exposes a resource management and monitoring interface as an SNMP agent instance.

[126] All these basic facilities can be found in the JMX platform, described in p. 2.6.4.

[127] Different monitoring modules are loaded for operation on different hardware platforms (Sun Sparc, Intel 32/64-bit architectures), operating systems (Solaris, Linux), Linux kernel versions (series 2.4 or 2.6), etc.

To summarize, the main concept relies on extending the monitoring system properties (compared to other modern distributed systems for grid monitoring) with the following additional features:

1. coherent and universal resource representation (using a uniform interface), allowing grid users (real users or other grid components) to access the monitored parameters of the infrastructure, network and applications in a uniform way, conforming to contemporary monitoring and management standards[131] (see section 4.3),
2. elements of functional adaptability to differentiated environmental conditions and types of monitored objects (section 4.5),
3. a self-configuration ability at the level of clusters and the grid (section 4.6).

The above-mentioned aspects are reflected in the proposed system communication architecture (shown in Figure 4.2), which encompasses four layers: instrumentation, interoperability, integration, and the client applications layer. The proposed hierarchical and layered architecture reflects a common approach to solving complex problems: it is based on decomposition of the problem into smaller parts and than solving each decomposed part separately, addressing each system aspect by a separate layer. Such architecture reflects the structure of modern grids and therefore, in the JIMS system, information is represented as the following triple[132]:

$$(grid, cluster, node),$$

where *grid* is the name of the grid testbed (i.e. CrossGrid, DataGrid, or CLUSTERIX), *cluster* is the cluster's network address (or that of its access node), and *node*[133] is the agent representing monitored node[134]. Such triple uniquely describes each part of monitored infrastructure. This approach is a typical solution for scalable distributed systems and network protocols (such as dynamic routing protocols) and perceives the grid as a hierarchical network of clusters and nodes.

---

[128] A component that exposes an SNMP-enabled device as a software component deployable in the JIMS system.

[129] Portable Batch System – a flexible batch queueing system developed for NASA in the early- to mid-1990s. It operates on networked, multiplatform UNIX environments, enabling batch submission and execution of jobs in a multi CPU environment [167].

[130] Sun Grid Engine - The Grid Engine project is an open source community effort to facilitate the adoption of distributed computing solutions. Sponsored by Sun Microsystems and hosted by CollabNet, the Grid Engine project provides enabling distributed resource management software for wide-ranging requirements from computing farms to grid computing [152].

[131] Specifications of existing monitoring and management models and protocols are discussed in chapter 2. Examples of such a protocol and monitoring/management interfaces include SNMP [42] and JMX [147,145].

[132] A triple is a sequence (not a set) of three elements, as ordering of the elements matters.

[133] Any monitored and discoverable node. This can be an operating system monitoring module as well as a running application.

[134] The model does not distinguish between numerous monitoring modules registered with one monitoring agent.

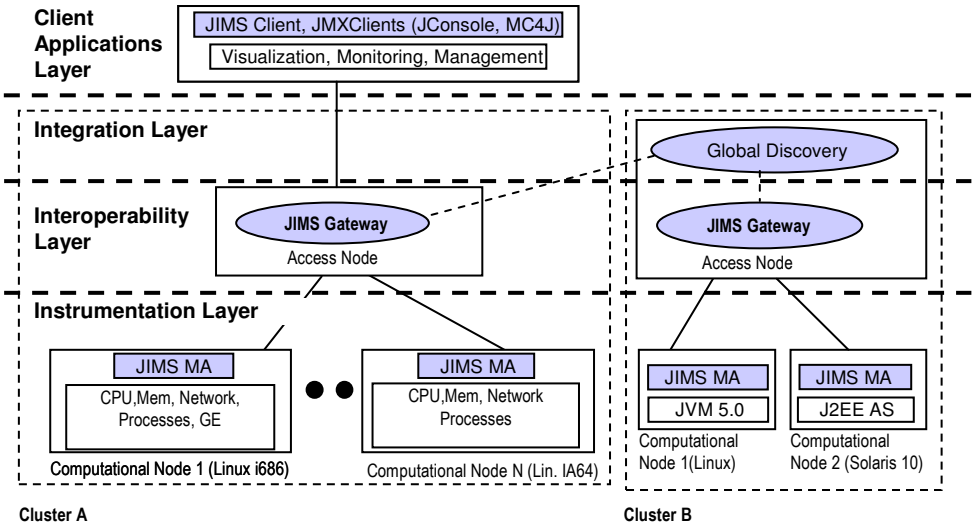


Figure 4.2 The layered architecture of communication between JIMS modules [178].

The layered architecture of the proposed JIMS sytem is shown in Figure 4.2, where JIMS MA is the basic element of the system and stands for Monitoring Agent. This agent, along with all other JIMS elements, will be described later on in this chapter.

4.2.1 Instrumentation Layer

The instrumentation layer represents monitored resources (devices, application, network sensors) as software components that can be registered with a JIMS agent and then remotely managed and monitored. In general, components can be seen as sensors which collect monitoring information by means of different sources, i.e. operating system interfaces such as */proc* **VFS** (Virtual File System) in Linux or Solaris, operating system calls using C or Java/**JNI** interfaces, monitoring and management protocols (such as SNMP) or JMX interfaces to connect to Java applications[135].

4.2.2 Interoperability Layer

Interoperability can be defined as the possibility to remotely invoke operations in a system implemented in a technology which differs from the one used to build the client. In a wider sense, it can be seen as an enabling technology, which circumvents problems with access to service functionality. In the presented monitored service, the interoperability layer also encompasses additional features, which aim at easy management of numerous nodes using a simple interface. It covers additional management and monitoring logic, which frees the client from performing trivial tasks and improves system performance. Next, it hides the complexity of monitored station management and exposes a single interface, which is

[135] Java 5.0 applications with monitoring agent enabled or other specially-designed application servers in **EIS**es (Enterprise Information Systems) providing JMX management interfaces.

consistent with other grid services. Finally, it allows external communication with agents operating on internal cluster nodes, behind firewalls and NAT.

4.2.3 Integration Layer

The role of the integration layer is twofold. It integrates monitoring services across all clusters and provides users with information on their availability. Cluster integration is performed by the GDS (Global Discovery Service) module operating inside JIMS and is described in section 4.6.2.

4.2.4 Client Application Layer

The client application layer is the consumer of monitoring data and is responsible for management and monitoring operations available by means of an **API** (Application Programming Interface), a CLI (Command Line Interface) or a GUI (Graphical User Interface). API is an interface for developers and other grid components and is not intended to be used by grid users (or administrators) directly. An API can be exposed via WS, HTML, or SNMP, using specialized components, such as connectors and adaptors. The other two types of interfaces cover standalone tools, which can be used for online monitoring, management, and troubleshooting from external locations, preferably using SOAP as an interoperable protocol.

4.3 Resource Representation Aspects

The author of this dissertation claims that component-based and coherent resource representation enables building a self-configurable, scalable and adaptable service for monitoring of infrastructure and applications. The proposed component architecture can be perceived as a way of building a complex system using simple, basic modules, which play the role of building blocks as well as monitoring sensors (see section 4.3.1). Coherent resource representation involves the ability to monitor differentiated resources using one, extendible interface (p. 4.3.3).

4.3.1 Component-based Architecture

According to the **CCA** (Common Component Architecture) [3], component architecture is a system composed of components with defined rules for linking such components together. Such component architecture was used for building a large portion of the monitoring service. Component architecture improves the abstraction level of software component development, as the installed code is not operating directly as a process of the OS, but is managed and accessible by the monitoring agent interface.

In the proposed system, numerous monitoring components (sensors) collect and expose the monitoring information from different resources. This information can concern Linux and SunOS operating systems running different versions of kernels (Linux 2.4, 2.6), different types of hardware (i686, IA64, Sparc), network interfaces, network capabilities (measurements of latency, bandwidth and throughput), storage elements (IO performance,

statistics, etc.), job management systems, SNMP-enabled network devices (hubs, switches, routers, network servers). Another group of components concerns the communication aspects of the system and includes components for dynamic discovery of nodes inside the clusters or dynamic global discovery of clusters in the grid. Another group can be composed of connectors and adaptors, such as RMI or SOAP connectors, and HTML or SNMP adaptors. For interoperability purposes, we can distinguish a group composed of components facilitating communication through gateways and firewalls, as well as groups of other, helper components (timers, dynamic loading services), HTTP servers for module repositories, etc. All these components play a very well described function, and in most cases are independent of one another. Such an approach allows dynamic construction of an application with the required functionality. Final functionality can also be modified during the monitoring system startup time and runtime, as described in section 4.5.

4.3.2 Coherent Resource Representation

The main purpose of JIMS is to enable monitoring of different types of resources, also supporting future resources, which will appear once the system is developed. This goal is achieved by packaging each monitored resource as a software object[136], which represents given resource functionality. Such components have a clearly defined interface and conform to a prescribed behaviour, common to all components within the system. This behaviour is not limited to exposed management attributes and methods, but also includes constructors, notifications and relations[137]. The software object can be coupled with the underlying resources very tightly (in the case of components communicating with resources via JNI), or very loosely (in the case of components communicating with resources via external protocols, such as RMI, SOAP or SNMP[138]). Such uniform component representation of monitored resources is the key to building a system that allows monitoring of different types of existing resources and even resources, which are not known during the development stage. Such an approach assures a unified method of accessing monitoring resources, while the hierarchical model of monitored data helps build a system, which is adequately scalable.

4.3.3 Flexible Interface

The monitoring service should be able to expose parameters in a unified and transparent way, i.e. the interface of the system should not be tied with the semantics of monitoring modules, it should remain unchanged for different monitored resources and should also be ready for future system extensions (addition of other monitoring modules). It should provide as much

[136] The sample implementation assumes that each resource/device is represented as a Java component of the MBean type.

[137] An example of such a facility is the JMX relation service.

[138] Such components can represent the state of the running application (exposed by RMI or SOAP) or network devices operating locally or remotely (e.g. remote switch, router or network server).

component functionality as possible using the simplest possible management interface. A natural solution to this problem involves the usage of the introspection mechanism, which represents (or reflects) classes, interfaces, and objects. Such a mechanism can be used to obtain information about class modifiers, fields, methods, constructors, and superclasses. It can also be used to obtain and set object attribute values or invoke methods on an object, even if the field name or the method signature is unknown to the program until runtime [104]. Introspection can also be used as a way of enabling invocation of any methods that become available in the future, as well as a means of discovery of the available component functionality.

4.3.4 Uniform Infrastructure and Application Monitoring

The JIMS system allows uniform monitoring of infrastructure and applications. This means that computational nodes and applications are discovered and accessed in a similar manner and both applications and computational nodes are represented as JIMS monitoring agents. The proposed monitoring system only addresses methods of instrumentation and attachment of Java applications (further described in the next chapter, covering JIMS implementation details). However, many general methods of application discovery are also discussed, such as static registration and runtime instrumentation to allow dynamic discovery. Concerning the latter, we propose the usage of a specialized dynamic discovery responder module, implementable in the following ways:

- during the development phase (addition of the module programmatically),
- during startup (startup-time application instrumentation),
- during runtime (runtime application instrumentation).

All these solutions are discussed in section 5.5 and describe monitoring of applications in terms of memory usage, memory maps, heap sizes and other details, including running processes and threads[139].

4.4 Interoperability Layer Organization

The SOAP Gateway concept addresses the functionality of the interoperability layer. Let us assume an environment where different protocols (such as WS/SOAP) are used on the client side, and other (typically binary[140]) protocols (for example RMI/IIOP) are applied on the side of monitored stations. Because of this distinction, the interoperability layer should fulfill the role of a translator, connecting monitored nodes through RMI and monitoring client applications through WS/SOAP. Such an approach provides JIMS interoperability with different types of clients using different technologies.

[139] All these parameters are available in the new version of JVM 5.0 with the JMX monitoring agent enabled.

[140] Such as RMI, with an underlying SUN proprietary binary protocol, or OMG IIOP.

Next, we can assume that grid clusters consist of many monitored stations; therefore, the interoperability layer should also perform the role of a router, forwarding requests from WS clients through one point of cluster communication, to specified nodes. The interoperability layer should expose this point of communication as one[141] well-defined application address and port (due to the presence of firewalls and packet filtering), with a WSDL document describing its functionality.

In large installations, covering thousands of monitored stations, an administrative method of registering each monitored station address would be ineffective. That is why this layer should allow automatic configuration of monitoring station addresses and support their dynamic discovery. A heartbeat mechanism should be used to remove stations that are not operating properly and fail to respond for a certain period. It should also become possible to notify monitoring clients of addition or removal of monitored stations. Such notification mechanisms should be offered using standard subscribe/unsubscribe methods. The mechanism of cluster-level self-configuration itself (dynamic discovery and heartbeat) should be hidden from the user's point of view.

Jus as any other grid service, the interoperability layer in the infrastructure monitoring system should support additional transient service instances, created and destroyed dynamically (covered in section 4.5). Such a facility should enable automatic installation of modules to facilitate management of numerous nodes in clusters. It should also provide a unified way to access different monitored resources (covered in section 4.3). In some cases, the described functionality should be doubled for failover purposes.

4.4.1 SOAP Gateway Concept

The problems described above are addressed by JIMS, which leverages the concept of the SOAP Gateway (**SG** in short), providing additional logic which includes the described discovery functionality and notifications. The concept of SG is based on a general approach, where a grid service is a WS defined using **WSDL** (Web Service Definition Language), and conforms to a set of conventions (interfaces and behaviours) that define how clients and services interact with each other. The SG approach is similar to the facilities described in WS-RF[142] [48], which enhances the WS interaction model with additional addressing facilities (WS-A) and notifications (WS-Notification).

[141] One per cluster.

[142] One WS-RF component specification, WS-Addresssing, defines such use of Web Services as a mediating point between incompatible protocols (see Synapse [10]).

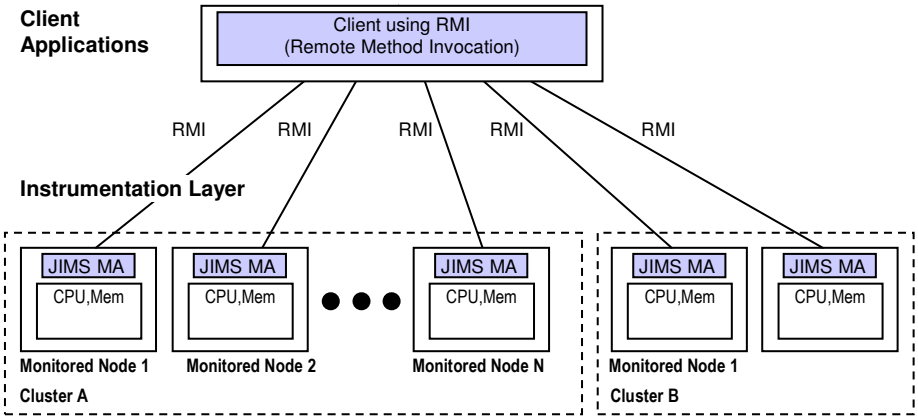


Figure 4.3 Direct access to each JIMS instance through RMI.

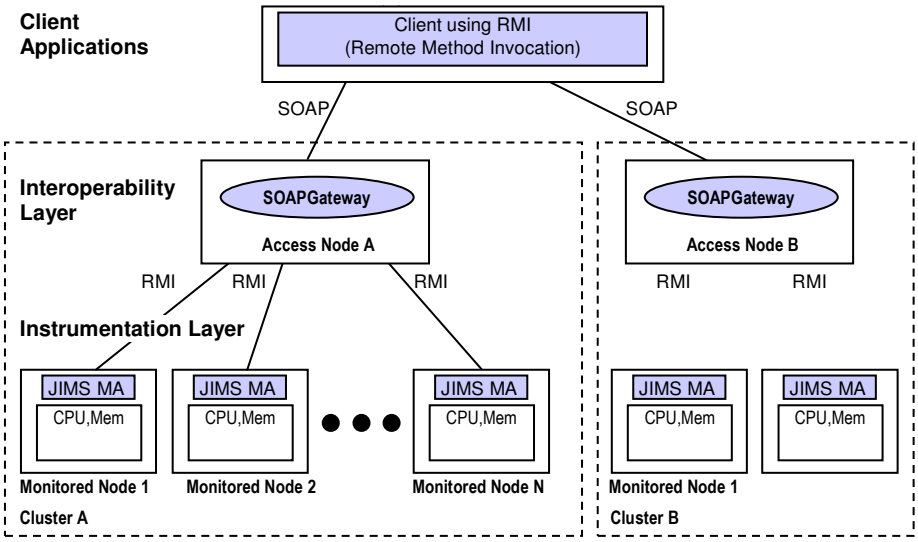


Figure 4.4 SOAP Gateway concept and its place in the interoperability layer.

4.4.2 Access to Monitored Parameters via SOAP Gateway

The figures presented in this subsection compare direct (Figure 4.5) and indirect (by means of SG, Figure 4.6) access to monitored data. Since the presented monitoring service allows resource management, the same method can be used by grid users to set a managed resource attribute of one selected WN or all WNs in a cluster.

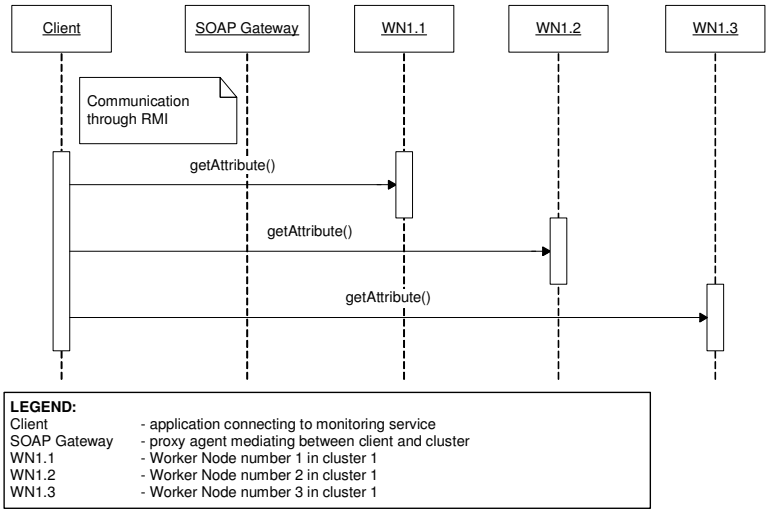


Figure 4.5 Direct access to monitored parameters using an internal, binary cluster protocol (here: RMI).

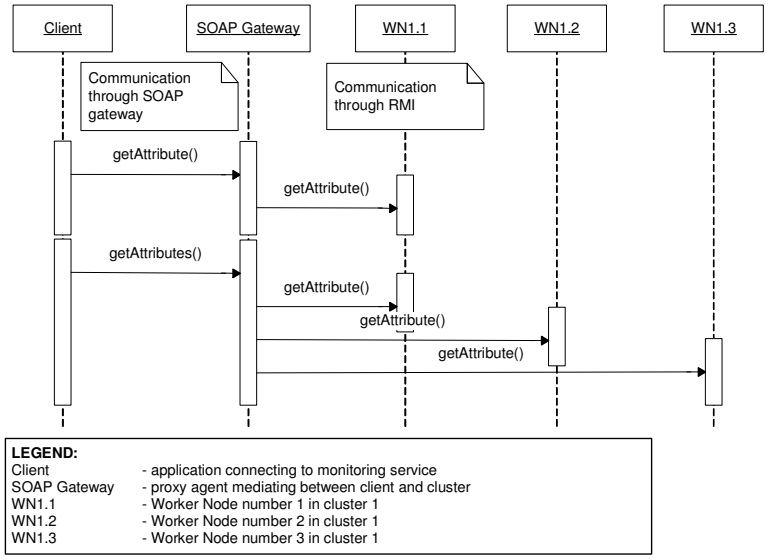


Figure 4.6 Indirect access to monitored parameters by means of the SOAP Gateway.

To summarize, the presented system exposes parameters of monitored stations through WS, performs the role of a gateway and manages active monitored stations in the clusters. As can be expected, SG requires very little logic on the client side because the complexity of dynamic node discovery is hidden behind the interoperability layer. It encapsulates all complexity of discovery (not mentioned here) and heartbeat mechanisms, as well as other mechanisms for dynamic loading of monitoring modules. One of the advantages of using SG as the monitored data access point is location transparency of the managed nodes. Each change in the monitoring station (nodes vanishing or changing addresses due to monitoring agent restart or physical crash) is handled by SG, hence the client application obtains a proper list of valid and active nodes each time. Proxy communication via SG helps manage and

monitor nodes with different protocols (RMI[143] or IPv4/6) and deals with firewalls[144] and private addressing[145] used in clusters.

4.5 Elements of Adaptability

In the scope of the presented thesis, adaptability is the ability of the monitoring system to modify its functions during startup or at runtime, in order to adjust its monitoring functionality to the requirements of the monitored environment (i.e. whether additional elements of the grid infrastructure need to be monitored) and also to the requirements of the monitored system itself (i.e. whether the monitored node is an access node with additional monitoring functionality[146] or simply an ordinary computational node).

4.5.1 JIMS Adaptability Features

There are two types of adaptability. The first type, relevant during monitoring agent startup, concerns types of loaded modules and depends on the discovered features of the environment in which the monitoring agent is started. The list of features should be passed as an argument to the monitoring agent. Such a way of agent startup separates the environment and monitoring requirements information gathering phase from the phase of real agent adaptability, i.e. the loading of the modules. The second type of adaptability is the ability to detect changes in the surrounding environment (during the agent's operation) and to adapt to these changes via logic hidden in monitoring modules.

4.5.2 Startup Time Adaptability

Within the process of startup time adaptation, we can distinguish two stages:

1. auto-sensing of environment features which have to be handled by the monitoring system,
2. explicit adaptation, i.e. the process of adjusting the functionality that is required from monitoring during startup.

The first stage relies on a set of predefined tests that are performed by the monitoring system in order to recognize the functional requirements introduced by environmental conditions.

[143] SG is accessible not only by a SOAP connector component, but also using a standard binary connector deployed in each monitoring agent. In the current implementation this is an RMI connector, available on the JMX platform.

[144] It is enough to open only one communication port (in terms of TCP) per cluster to communicate with all cluster nodes. Since no so-called ephemeral ports are opened, firewall configuration remains relatively simple.

[145] Typically, SG is installed on an access node and allows external clients to communicate with nodes inside the cluster, which often uses private addressing and is accessible only via NAT (SNAT and/or DNAT).

[146] One of the common elements of access node functionality in the JIMS system is the module repository function. Typically, this role is delegated only to one access node, selected from all available access nodes in the grid.

The second stage involves loading and startup of modules, which implement the required functions. The concept presented in this thesis foresees a wide spectrum of loaded functionality, covering different aspects of monitoring systems, starting with the instrumentation layer (accessing monitored resources), and ending with high-level layers, equipped with special interfaces for interoperability with other systems. Modules loaded at the adaptation stage can play different roles depending on the location on which the monitoring system is being deployed. Additionally, the concept foresees a primary set of modules installed in every monitoring agent[147] to provide the initial services required for:

1. installation of other, more specialized modules,
2. providing basic connectivity,
3. allowing agent discovery by a remote management station or a management agent.

Modules loaded at the second stage should meet additional environment requirements. These requirements may concern different kernel versions, grid engine configurations and network configurations. For example, grids commonly include systems which:

1. utilize Intel i686 32-bit, Intel Itanium 2 64-bit, or Sun Ultra Sparc III architectures,
2. have different types of operating systems installed (Sun Solaris[148], GNU/Linux[149], etc.)[150],
3. have different numbers of CPUs per node,
4. use different job management systems[151].

Additionally, installing modules from remote locations at the second stage addresses the problem of maintaining current versions of the software on all nodes on the grid. Moreover, loading software modules from remote locations can be performed on existing standard platforms for development of distributed applications[152] and depends on the application of the monitored system.

---

[147] The monitoring agent in the described monitoring system is a single process which has all the functions required by this system in the scope of one **node**.

[148] The described monitoring system has the ability to monitor only Sun Solaris 8 systems. Newer versions of Solaris OS include different versions of the kernel, with no backward compatibility of the monitoring interface.

[149] GNU/Linux systems are the main platforms for which the described monitoring system was developed. Linux is a base operating system for grids in most European and worldwide projects. For example, the EU IST CrossGrid project was fully based on the Linux Red Hat 7.3 distribution (the use of a seemingly outdated version resulted from the requirement for a stable testbed which had already been established in some older projects; most importantly in the EU IST DataGrid project).

[150] The problem with heterogeneity of hardware and software arises from the large number of combinations of different hardware and software elements: for example, there are Solaris-driven machines with both Sun Sparc and Intel CPUs; on the other hand, there are also Linux machines based on Sun Sparc and Intel hardware.

[151] Such as SGE [152] or PBS [167].

[152] Two common platforms that provide ready-to-use systems for loading components from remote repositories are JINI and JMX.

4.5.3 Runtime Adaptability

Further adaptability is hidden in monitoring sensor modules and concerns adaptation to environment conditions, which can change over time. This type of adaptability can, for example, cover the version of the IP protocol (IPv4 or IPv6) in use by monitoring agent connectors and related addresses returned by the agents in the proces of cluster-level discovery. If an agent is first available via IPv4 and then switches to IPv6 (the form of the returned IP address changes), the monitoring service should handle this event and dynamically adapt to the new version of the communication protocol (see Listing 5.19, Chapter 5). This type of adaptability can be perceived as dynamic adaptability, as opposed to startup time adaptability. It is, however, only dynamic with respect to the time in which it occurs, because the logic itself is statically coded.

4.5.4 Module Repository and Dynamic Module Loading

The term *module* is typically used to name a part of a program. In the described monitoring system, this term is denotes a distributable and installable element of a monitoring service. Such a module is packaged as an archive containing software *components* with corresponding helper configuration files and compiled binary code.

The ability to dynamically load software components is a key feature of the proposed monitoring system. It allows to start up a very simple agent with only base modules installed and then to install only these monitoring modules, which are required. This fulfills the requirement of adaptability to different environmental conditions and provides the functionality, which meets hardware specifications. One problem, which arises here, concerns the fact that the agent should know which modules to install[153] and should allow the following operations of dynamic component deployment:

1. creation of the component inside the container[154] with given parameters (i.e. component name and constructor parameters),
2. creation of the component outside of the container and then registering it within the container, enabling it to access all the available container services,
3. dynamic loading of components from a remote location using a common solution for module transportation and storage (preferable HTTP and an ordinary file system[155]).

The first two methods of dynamic component deployment differ only with regard to module creation time and do not carry any additional advantages for developer. Both of them concern components which are available locally (their binary code is stored on the machine the agent

[153] The problem is addressed by p. 5.3.1.

[154] The term container means an agent which holds the list of all registered components and exposes their management interface. It should not be confused with J2EE containers.

[155] We can imagine, for example, modules stored as **BLOB**s in a DMBS system.

is running on) and only load the binary code from the local file system. This scenario is designed for installation of basic agent services, initialized during startup, which definitely cannot be installed from a remote location. An example of such a module installed locally is a HTTP server used as a remote module repository. Such a module cannot be downloaded and instantiated from a remote repository because it is the first service initialized when no repository is yet available in the system. Other agents that load modules from an existing remote repository do not need to install and start any such HTTP server. This necessarily distinguishes some monitoring agents from others. The first agent will play the role of a master, while others will be designated as slaves and will use the third method of component deployment. The concept of dynamically loading modules from a central repository is presented in Figure 4.7.

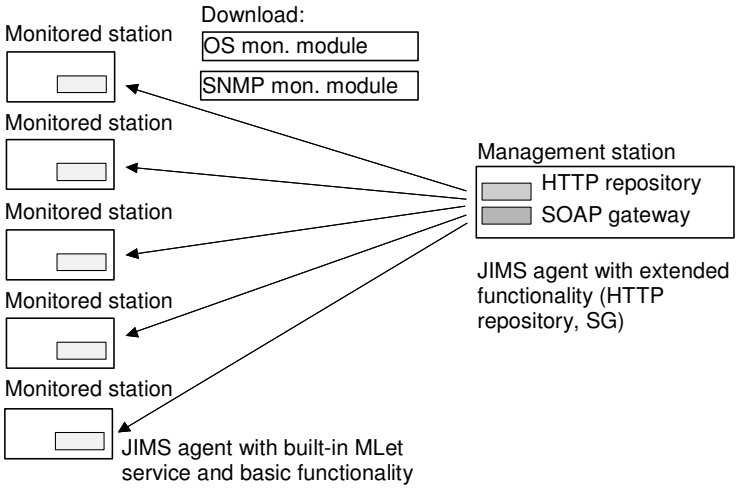


Figure 4.7 Dynamic deployment of JIMS components: download and installation.

4.6 Self-Configurability

Self-configurability is an important feature of the proposed monitoring system. During startup and operation of the JIMS monitoring system we can distinguish two phases: dynamic deployment of sensor components and their configuration inside the local clusters (cluster-level dynamic configuration of connections to monitored nodes), and then global configuration of JIMS monitoring agents running in the grid (grid-level dynamic configuration, including algorithms for cluster registration with a set of chosen global nodes called a global registry).

These two aspects express a hierarchical approach to scalable system design, reflecting the two-tier grid infrastructure, facilitating local discovery of computational resources at the cluster level, and permitting global discovery of computational clusters at the grid level. Such an approach allows building a grid monitoring service, which addresses the problem of finding and registering dynamically appearing monitored resources in the grid, as well as deregistration of resources that fail or are disconnected by the network. Although both problems appear similar in nature, they have to be solved in different ways, since in a local

cluster we can typically use multicast communication to find new resources, but in a global network we are only able to use unicast[156] transmission and have to implement an announce-and-register type of protocol. Nevertheless, in both situations some type of heartbeat mechanism should be used for de-registration of deprecated resources. In the first scenario, this mechanism is based on responses received regularly from multicast parties, while in the second scenario it is based on periodically incoming registration announcements.

4.6.1 Cluster Level Self-Configuration

The cluster level self-configuration mechanism is the first element of a self-configurable monitoring service built using the component approach based on the JMX specification. The solution presented here is based on the active discovery approach to finding monitoring agents. As such, it should be discussed and compared to other methods[157] of finding resources in the cluster, which is briefly done in this paragraph.

Static Configuration

Static configuration is the simplest solution to the presented problem and can be used in all HPC[158] clusters where the configuration of computing and other helper nodes is established only once and not subject to frequent changes. Such a solution can assume that the monitoring system stores a list of all worker nodes and that this list properly reflects the current state of the cluster for the purpose of node failures or replacement.

Naming Service and Leasing

A naming service represents another approach to solving the problem of finding computing nodes available in the cluster. It is based on the process of registration of addresses of monitoring agents in one (or a set of) well-know naming service(s). This service should allow registration of any computing node address under a uniquely distinguished name, for example basing on its IP address. The naming service can be complemented by leasing functionality which removes unused registered objects after a given registration timeout. In-depth analysis of naming service and leasing issues in the context of Jini can be found in [60].

Peer-to-Peer Discovery Service

P2P (Peer-to-Peer) is a modern communication model with direct communication between equal network entities. P2P technology stands opposed to the classical server-client

[156] We assume that we are using the standard Internet WAN (Wide Area Network), without special modifications by network administrators and without multicast routing. This assumption does not seem to be very strong, because the majority of solutions based on special networking functions such as multicast routing, modification of headers of IP or TCP PDUs, are rare in grid systems.

[157] Methods such as static configuration, naming service, and passive/active discovery [143].

[158] High Performance Computing.

communication model, because it does not require a dedicated server. Moreover, the P2P approach shifts the communication model to a higher abstraction level and is independent of the underlying network protocols, like TCP/UDP, and network capabilities of unicast or multicast transmission. Research on a P2P communication model, which could be used in the JIMS monitoring service at the cluster or grid level, has already started and there are already some interesting results concerning the implementation of a JIMS connector using the P2P architecture[159]. However, this approach has not yet been proved useful in grids, and is beyond the scope of this thesis. Preliminary results of this work can be found on the website of a project focused on development of a P2P connector for JIMS [27].

Passive Discovery

In order to describe passive and active dynamic discovery mechanisms, the author refers to the JDMK framework, which addresses these issues in detail[160]. In the JDMK passive discovery scenario, the searching entity listens for answers sent by special components, called discovery responders, which are being activated or deactivated. When discovery responders are started or stopped, they send out messages, which contain all discovery response information. The object that waits for any such messages coming from a multicast group is called the discovery monitor and it maintains an up-to-date list of all agents in a given multicast group (see Figure 4.8) [143].

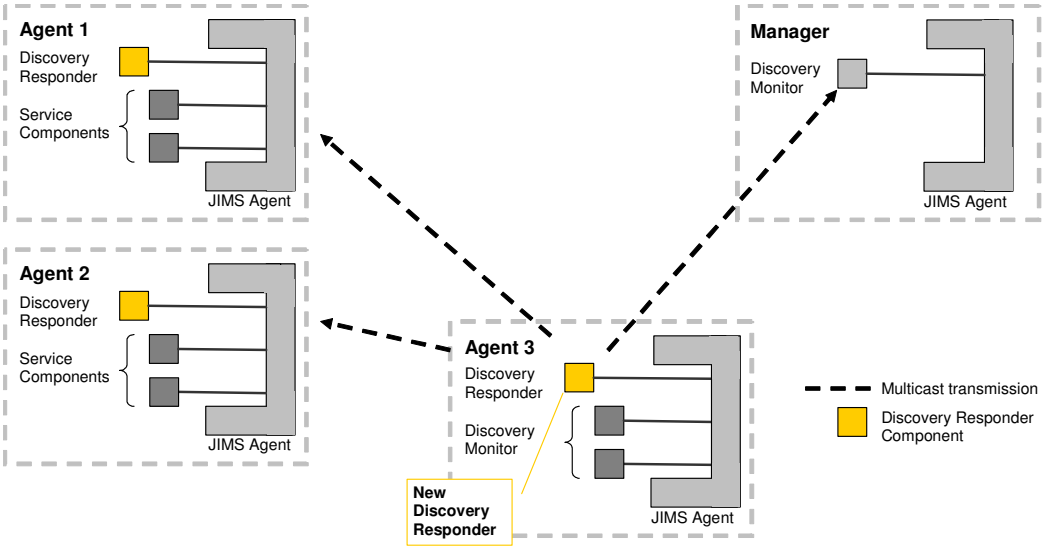


Figure 4.8 Passive discovery of discovery responders [143].

The agent that wishes to be discovered must have an active discovery responder, which plays a role in both passive and active discovery. First, when the discovery responder is started, it

[159] The experiments are performed using the JXTA™ technology proposed by Sun Microsystems.

[160] The JDMK framework was briefly described in p. 2.6.5.

sends out a multicast message indicating that it has been activated. Second, when the responder is stopped, it sends out a multicast message to indicate that it is being deactivated [143].

Active Discovery

In the active discovery scenario, it is the discovery client that initiates searches for agents in the network (Figure 4.9). The client sends out discovery requests and a discovery responder in each agent responds to these requests. Each instance of the responder supplies the return information about the agent in which it is registered. The application containing the discovery client can initiate a search at any time. For example, it might perform a search when first launched and periodically search again for information about communicators, which may have changed. For each search, the discovery client broadcasts a request and waits for return information from any responders. The discovery client provides methods to discover agents. The active discovery operation sends a discovery request to a multicast group and waits for responses. Discovery clients can only discover agents present in the same multicast group and listening on the same port, therefore the design must coordinate this information between the discovery client and all responders [143].

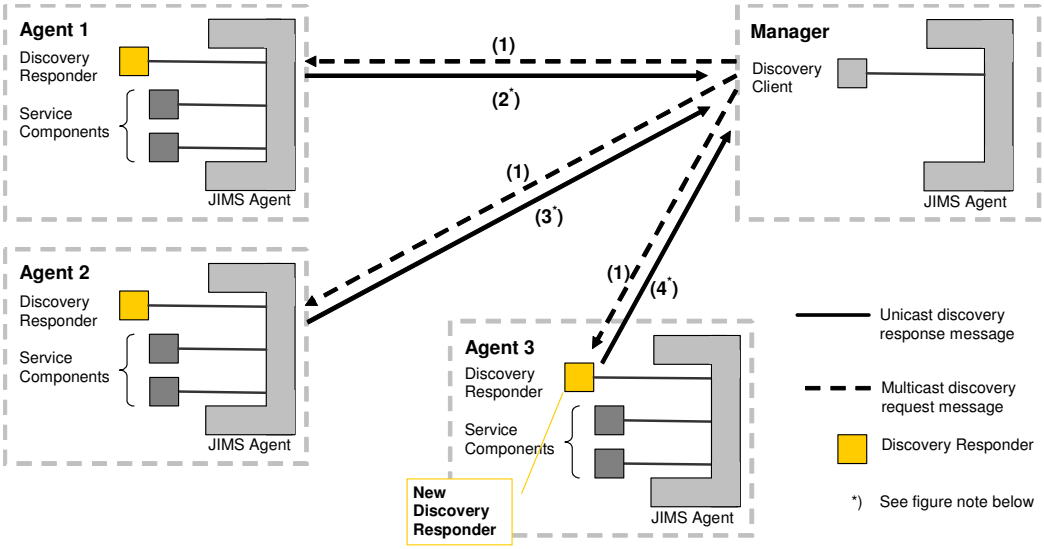


Figure 4.9 Active Discovery of Discovery Responders[161] [143].

The active discovery mechanism is supported by the heartbeat process, used to locate monitored stations, which fail to respond for some reason. If a station does not respond a certain, predefined number of times, it is removed from the list (i.e. the registry in SG) and is not treated as available anymore (Figure 4.11).

[161] The sequence of discovery response messages can differ in practice.

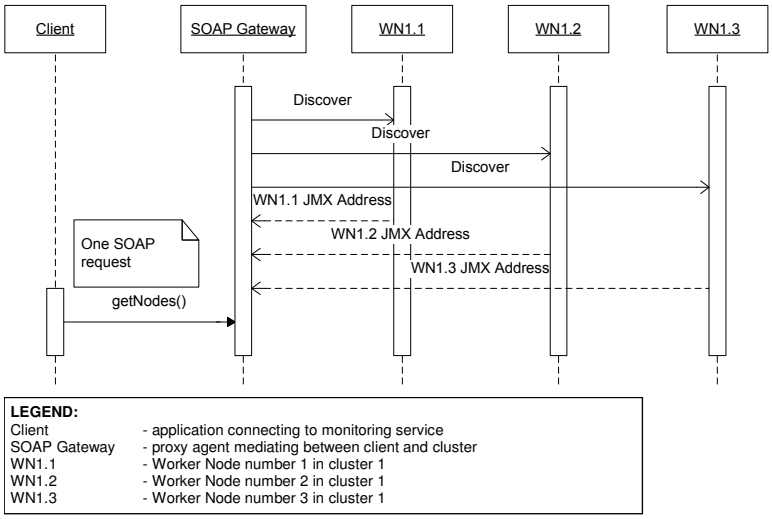


Figure 4.10 Cluster-level active discovery scenario.

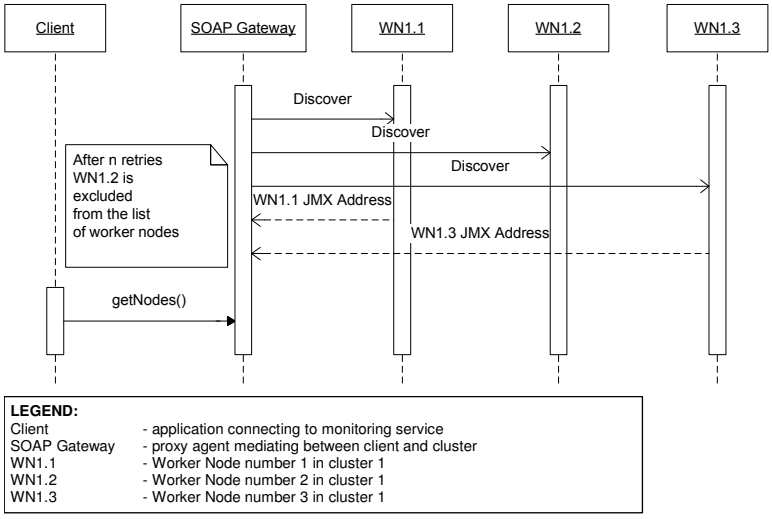


Figure 4.11 Cluster-level heartbeat scenario.

All discovery methods discussed above are listed in Table 4.1 and are compared to the static (reference) configuration of worker nodes. We can say that network traffic overhead for all dynamic methods is always higher than zero and is the highest in the active discovery approach. The first investigated method, i.e. a naming service equipped with a leasing facility, seems to be highly reliable[162] in terms of dynamic discovery; however, it requires a separate method (active or passive) to dynamically discover the naming registry. Another approach, based on P2P, is a hybrid method which can be successfully used for both local and global networks. However, this approach, including P2P communication through JXTA, has not yet been tested in terms of its applicability to the JIMS architecture. Similarly, the passive

[162] In terms of how the changes in the dynamic environment are reflected and detected by the discovery mechanism.

discovery mechanism was abandoned due to its low reliability, resulting from lost discovery notifications.

Table 4.1 Comparison of the presented discovery methods.

Discovery Mechanism	Traffic Overhead	Dynamic Discovery	Reliability	Implementation
Static Configuration	No	Not available	Low[163]	Very straightforward
Name Service and Leasing	Medium	Medium	High	Complicated, requires discovery of naming service
P2P (Peer-to-Peer)	Medium	High	High	Very complicated
Passive Discovery	Low	Near-real-time	Very low	Simple
Active Discovery	High	High	High	Simple

Following investigation and practical verification of some[164] of the presented dynamic discovery mechanisms, we have selected the active discovery mechanism, which was then used for implementation of worker node discovery in clusters. Such a mechanism operates efficiently[165] in an environment where the dynamics of monitored resource availability are very high, i.e. the probability of addition of a new monitored computational node or of node failure[166] is very high. The second level of discovery is addressed by the grid-level self-configuration mechanisms and is discussed in the next paragraph.

4.6.2 Grid Level Self-Configuration

The second level of JIMS self-configuration concerns clusters distributed across the grid. The mechanism of grid-level site discovery is determined by the unicast communication allowed in public networks and can be described as a mechanism of publishing information about cluster presence to a well-known registry or set of registry nodes[167]. In the JIMS monitoring service, the node that holds this information is called the **GR** (Global Registry) and can be configured statically by a grid administrator or chosen dynamically by means of a

[163] Values are assigned arbitrarily by the author.

[164] The tested mechanisms included static registration, leasing and passive/active discovery mechanisms.

[165] In terms of reliability with reference to implementation complexity.

[166] A failure can result from the failure of computational node hardware or software, or from a disruption in the interconnecting network, called *network partitioning*.

[167] Another option for finding clusters in the grid is to use one of the P2P technologies (such as JXTA). However, the author does not currently posses sufficient results which could be presented in this matter.

special algorithm. In both cases, the process of collecting site availability information globally in the grid is called **GDS** (Global Discovery Service).

Static Global Discovery Service

The first attempt to build a self-configurable service at the grid level was made using the architecture of connectors, the SG facility and a static, predefined set of GR nodes. The novelty of this (as well as the second) solution is hidden in the usage of specialized JIMS layers, which separate communication issues from the global discovery algorithm itself, allowing the development and testing of many different discovery approaches, ranging from very simple algorithms with statically chosen registries, to sophisticated algorithms with dynamically elected single or multiple global registry/-ies.

A typical communication sequence diagram for the *pull* monitoring scenario in JIMS is presented in Figure 4.12. In this scenario, the client connects directly to SG, and obtains a list of all available monitored WNs. Subsequently, it requests certain parameters from monitoring agents operating in the retrieved WNs[168]. This scenario assumes that the client knows the address of the SG, which lists the monitored stations. If the client wants to gather information about all available WNs in all clusters, it has to maintain a list of all SGs in the grid, which is the main problem addressed in this section.

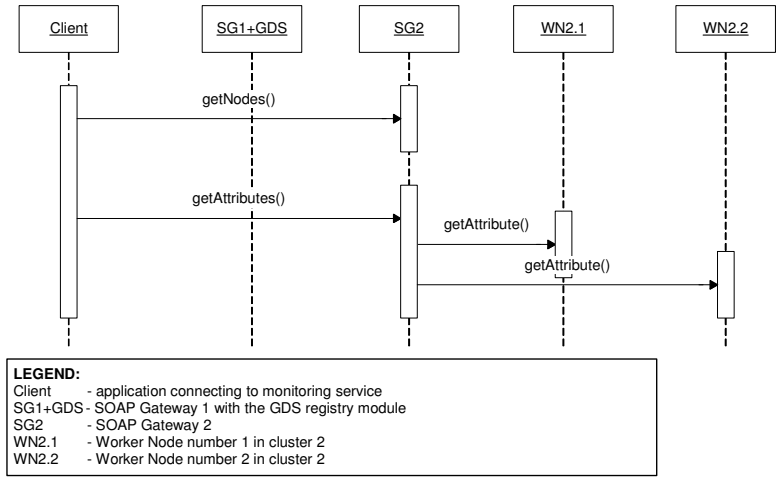


Figure 4.12 Direct connection to SG.

The GDS that solves this problem should have some well-defined features. First, it should allow monitoring agents to register any new SGs[169] and should expose a list of currently registered SGs. Next, it should be equipped with a heartbeat mechanism for removing old entries of defunct SGs. Furthermore, the GDS should provide failover mechanisms and be

[168] The client can simply obtain all attributes of the same type in one call, or obtain monitoring attributes by separate calls to chosen worker nodes, using the previously retrieved list.

[169] With timestamps indicating when the registration took place.

immune to software and network failures. Finally, in order to allow any client that knows only one SG address (even if it does not refer to a specific GR), the GDS should provide the address of the GR node in each instance of the SG.

The basic usage scenario, with GDS defined as above, is shown in Figure 4.13. It is similar to the scenario presented in Figure 4.12, but is supplemented by the *getRegisteredSGs()* call which returns the list of valid SG addresses, accessible at the time when the method is called. Once this method is called, communication between the JIMS client and monitoring services takes place in the same way as before, i.e. the client connects to SG, then requests monitoring parameters from WNs or executes a method on WNs to perform some measurements.

An important element of such a GDS service is a well-defined set of a certain number of SOAP gateways serving as GDS registries. These GDS nodes should be configured in all clusters, because all SOAP gateways should register themselves in these chosen nodes, and hence the list of chosen nodes should be accessible in every cluster. This postulate is fulfilled because the list is exposed as an attribute of the GDS component and is available for clients at any time. Given such a facility, clients do not need to store a list of all GDS nodes: they only need to maintain the address of one SG – not necessarily the SG that performs the role of GDS (see Figure 4.13). It is only required that the GDS component is installed and GDS node addresses are available, so that the client can read them, connect to a GDS node and then read a full list of all available SGs.

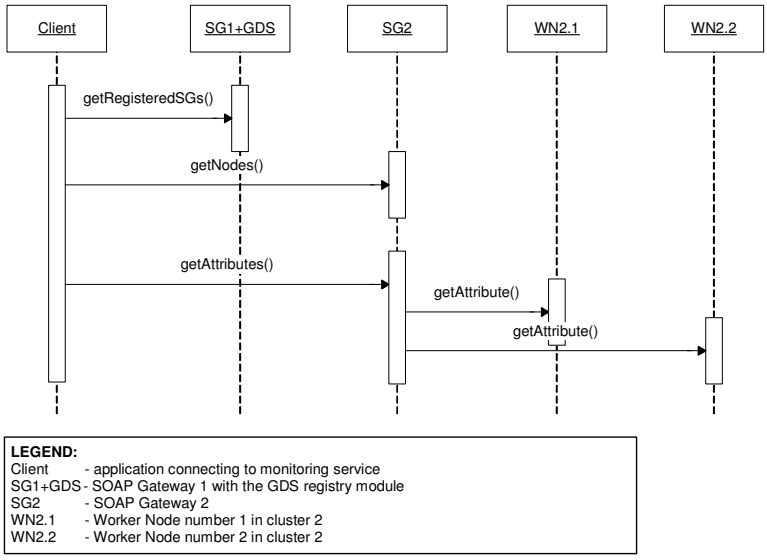


Figure 4.13 SG localization using a GDS node.

The client can also connect to all available GDS nodes, compare lists and, by using the majority method, choose the statistically most valid set of SGs. Using the list of SGs; it can then query all sites with SGs and read information about all monitored nodes.

The most sophisticated scenario is shown in Figure 4.14. In this case, there are three SGs: SG1, SG2 and SG3. SG1 also performs the role of a GDS node, as specified in the component configuration file. Let us assume that the client is not aware of the addresses of GDS nodes,

so it chooses the first known address of the SG (in this case this is SG2), which can be the local SG from the client's point of view. Such an SG is typically located in the cluster's access node and it is assumed that its address is known by the external client. From SG2, the client obtains a list of GDS nodes by using the *getGDSNodes()* method. Having this list, the client then connects to one of the GDS nodes (in the figure this is SG1) and executes the *getRegisteredSGs()* method. In the next step, the client connects to SG1, SG2 and SG3, and requests information about interesting worker nodes (for instance for SG3).

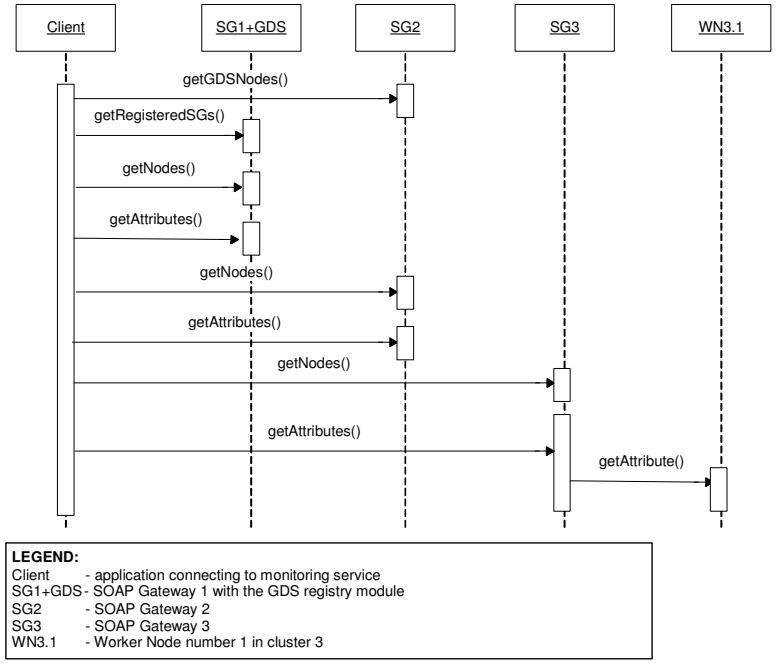


Figure 4.14 GDS localization using SG with GDS MBean installed (in CE2).

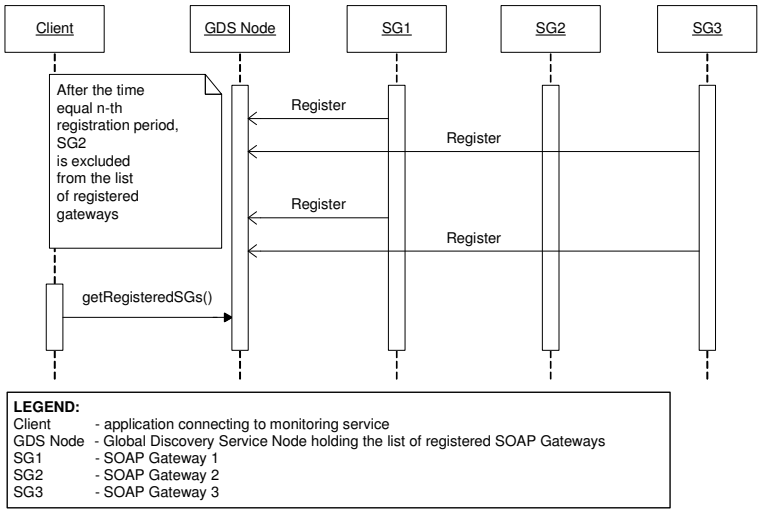


Figure 4.15 Registration and heartbeat mechanism scenario in the GDS.

In order to obtain a full list of WNs available through a given SG, the client connects to it and performs the *getNodes()* method, which returns the addresses of all WNs in the cluster.

Subsequent calls only retrieve the values of requested parameters. It is also possible to call (indirectly) a particular WN, using the returned WN monitoring agent address.

Dynamic Global Discovery Service

The second experiment involving a GDS service for JIMS was developed using the same connector architecture, but with the use of a specially designed algorithm for dynamic global registry selection. Its main activity is to spread information about sites belonging to the grid. Such an approach allows collecting information about network addresses of existing and correctly working sites included in the grid. This information is particularly important for the monitoring application, which manages sites connected to the grid. The following requirements for the creation of such an improved Global Discovery Service mechanism involve:

1. gathering and saving the availability information of all sites belonging to the grid,
2. providing a list of all sites operating in the grid,
3. handling cases of cluster addition or removal,
4. no central, static repository (so-called single point of failure), which is the key change in comparison to the previous solution,
5. the protocol must work and handle network and cluster failures without any administrator activities.

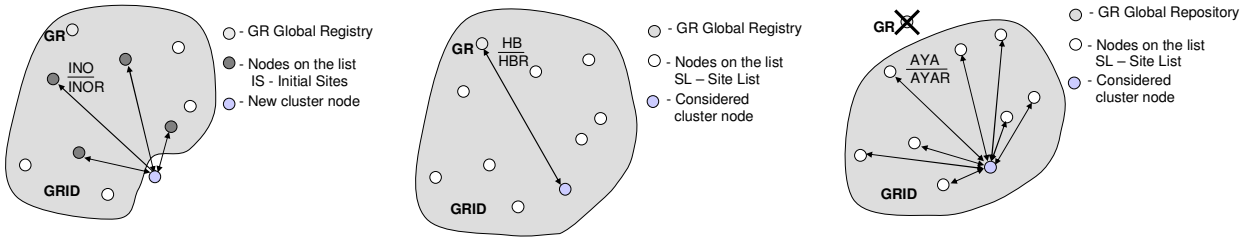
Analysis of the above requirements has led to the design and implementation of the GDS service as a managed component[170] deployed in JIMS. Authors[171] of the module have decided that there would be one repository of site addresses called the Global Registry (GR). This repository contains network addresses of each registered and operational site. GR localization is dynamically selected and might be automatically changed at any time. All clusters, which are going to be attached to the grid, must register in GR and maintain their availability status entry through periodical heartbeats. Moreover, each site can obtain a list of clusters currently belonging to the grid. The repository checks if the cluster should be unregistered in case of lost heartbeat (cluster crashed or turned off). If the GR crashes, a new repository is chosen, using the election algorithm, in a randomized way. From that moment on, sites refresh their state in the already-chosen GR. At any moment, the application can request from any cluster in the grid a complete list of working clusters. The protocol works by exchanging messages (request-response) between sites following strictly defined rules. For that purpose, nodes in the system with GDS are divided into two types: active nodes, representing a cluster, which

[170] JMX MBean component.

[171] A solution for this improved discovery mechanism was proposed by K. Wojtas and L. Wasilewski and was described in [172]. However, the author of this dissertation took the liberty of briefly presentating this mechanism, because he participated in the development of the presented component and in its integration with the JIMS monitoring system. The described solution was successfully applied in the CLUXTERIX grid project.

actively exchanges protocol messages, and passive, application nodes, which only query the site in order to obtain a list of clusters. Full activity of the protocol is based on **FSM** (Finite State Machine), where each active node is in one of the following states:

1. *Normal Working* – cluster[172] registered in GR,
2. *Global Registry* – cluster is a GR,
3. *Looking for GR* – cluster is not a GR and attempts to find or select a GR in order to register in it.



(a) cluster added (phase when a new cluster node is joining the grid)

(b) heartbeat (grid topology monitoring – phase of normal activity)

(c) Global Registry election (new Global Registry discovery)

Figure 4.16 Global Discovery Service operation scenarios [173].

In this distributed global discovery mechanism, the newly started GDS node (scenario (a) in Figure 4.16) registers itself in a predefined set of so-called Initial Sites and exchanges INO[173] messages in a procedure referred to by the same name (INO). After finding the GR, it moves to the HB[174] scenario and keeps an up-to-date list of all functioning GDS nodes (Figure 4.16 (b)). In case of GR failure or network disruption/partitioning, a new GR is elected within the part of the network that has no valid GR (Figure 4.16 (c), the AYA[175] procedure). More details on this GDS mechanism, including all FSM/STD diagrams, can be found in [172].

4.7 Chapter Summary

The chapter discusses the concept and architectural details of the JIMS grid monitoring service. The concept is based on a layered architecture which separates instrumentation and communication concerns, thus facilitating fast development of sensors, uniform monitoring of different aspects of grid infrastructure (including hardware infrastructure, network, and applications), and dynamic discovery of monitored resources.

[172] In GDS, each cluster is represented by a corresponding GDS node, typically the same as the cluster's SG.

[173] *I'm a New One.*

[174] *HeartBeat.*

[175] *Are You Alive?*

The presented uniform interface to monitored resources and component architecture proposed in this system allows it to adapt to the encountered environmental characteristics. Moreover, a built-in self-configurable mechanisms allow the system to automatically retrieve a complete list of available computational nodes inside particular clusters, and, finally, to obtain a list of all available clusters. Such a mechanism represents an efficient (in terms of communication overhead and intrusiveness) and scalable approach for the two-tier architecture of modern grids. The presented access interface concept can be easily generalized in order to provide access to other aspects of system functionality, such as its configuration and management.

5 JIMS Implementation Details

This chapter describes the design of the JIMS system prototype, implemented on the Java platform, using specialized Java extensions (the JMX technology) for management and monitoring. Three prototypes are presented, leading to the current shape of the described monitoring system. Selected implementation details are discussed, concerning discovery, self-configurability and adaptability. Sample sensor modules for monitoring different elements of the grid infrastructure are also presented, along with a discussion on possible security solutions.

As the implementation framework of the monitoring system prototype, the Java platform was chosen, with specialized JMX extensions for management and monitoring. The Java platform provides the required level of portability and interoperability[176], and its built-in object-oriented capabilities such as introspection (described later), allow us to programmatically manipulate elements of the language[177], while also allowing developers to define monitoring and management interfaces on the fly[178]. It is also important to note that the Java platform is a technology that is widely accepted by the grid developer community[179] and commonly used in current grid middleware implementations.

This chapter describes the implementation of four JIMS layers, including the representation of monitored resources as manageable software components called MBeans, JIMS self-configuration mechanisms, solutions that allow uniform infrastructure and application monitoring[180], and, finally, monitoring sensors.

[176] For example, the Java RPC mechanism based on RMI/IIOP allows interoperability with CORBA/IIOP, widely used in heterogeneous distributed systems.

[177] It is possible to load and instantiate a class or call its methods using their names and signatures.

[178] Moreover, by using JMX, it is possible to achieve runtime instrumentation of any class, which in turn allows monitoring of any type of resource encapsulated as a Java object.

[179] The most popular implementation of the current WS-RF specification, GT 4.0, is written in Java.

[180] Including dynamic discovery of applications running in the grid.

5.1 JIMS Prototype Implementation

The proposed prototype implementation of the JIMS monitoring system was designed using open source software, based on J2SDK 5.0, with a standard implementation of JMX [147,145] (JIMS agent, dynamic loading, connectors, adaptors, etc.), WS (AXIS) [16], and network monitoring via SNMP [5]. The final shape of the described system is the result of several years of development and deployment of the system in real grids. The system has undergone long evolution, resulting in three prototypes, ranging from a Jiro-based prototype, to a JMX-based prototype and its enhanced version, equipped with additional features such as the MX4J SOAP connector as a WS interface, dynamic GDS, and adaptability to future grids based on IPv6 and IA64 architectures.

5.1.1 First Prototype of Jiro-based Grid Infrastructure Monitoring System

The first prototype [17] of the system was based on the Java 1.3 platform, JDMK and Jiro technology (Sun FMA implementation), and was called the Jiro-based Grid Infrastructure Monitoring System (Figure 5.1). Originally, this system was composed of five tiers: *instrumentation*, *agent*, *management logic*, *database* and *user interface*. The instrumentation layer was the equivalent of the JMX instrumentation layer, responsible for abstract resources representation in the form of MBeans. The agent layer was composed of a JDMK MBeanServer and additional Jini and Jiro services (Jini Lookup Service and Jiro Deployment Station) and was responsible for exposing MBean interfaces to external systems. The management logic layer provided the *programming* facility (failure reporting, notification mechanisms, detection of certain events) to the monitoring service and was available through BeanShell scripts (a Java-compatible scripting language, [153]). The database layer was used for storing the history of the grid and to build statistics or views of this history. User interface formed the topmost layer of the system and was intended to visualize monitoring information using the JDMK HTML adapter (text information), a custom GUI application and a Web front-end leveraging the **JSP** (Java Server Pages) server-side technology.

The proposed prototype solution provided administrators with enormous flexibility and manageability, inherited (to a great extent) from JDMK and Jiro. The system was equipped with a very general instrumentation layer providing notification and query mechanisms. It was easy to manage, as all services could be created and configured dynamically during runtime. The concept of an embedded management logic layer also seemed to be very powerful and could be used for automation of the management process, which is very important in large grid installations [100]. The initial implementation of the system did not fully support automatic installation and deployment. Neither the database layer nor the rule

engine, nor the JSP visualization part were available at that time[181]. Another disadvantage of this system was lack of support for unattended installation and startup[182]. Furthermore, the architecture only allowed cluster monitoring, which finally convinced author to switch to a slightly different solution, preserving the instrumentation layer in an almost unchanged shape.

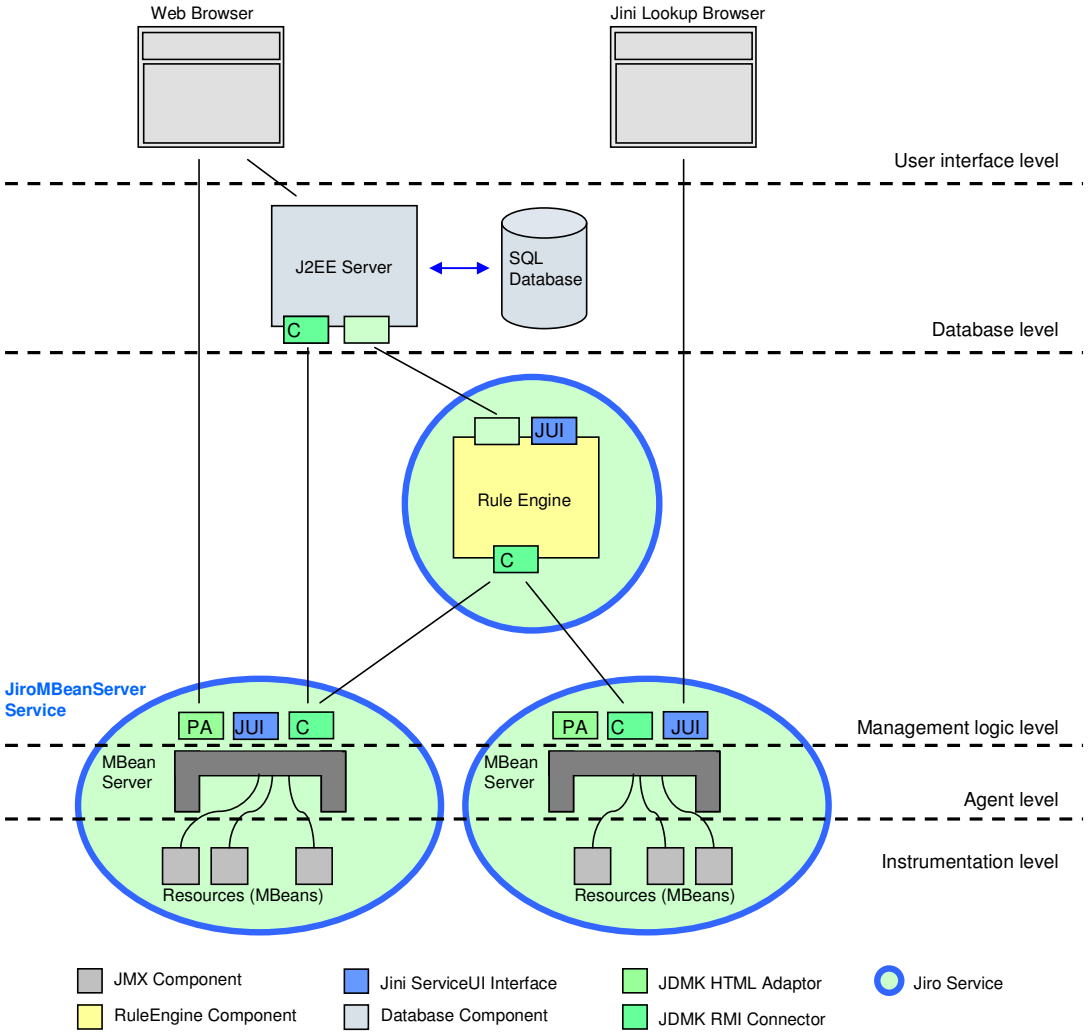


Figure 5.1 Architecture of the Jiro-based grid infrastructure monitoring system [100].

5.1.2 Second Prototype: JMX-based Infrastructure Monitoring System

Due to the problems encountered during the implementation of the first prototype of the monitoring system in real grid installations[183] and the fact that Jiro technology was

[181] The development of the database system was initiated as a separate task in the DSRG group at AGH-UST. Similar effort was also undertaken at the Department of Computer Science of the University of Dublin, Trinity College, in Ireland, also participating in the EU IST CrossGrid project.

[182] The system used a custom GUI application (the OKI Service Finder) to locate Jiro services and the Bean Shell tool for administrative deployment of monitoring agents.

[183] The CrossGrid project.

abandoned[184] by Sun Microsystems, the author decided to simplify and change the system architecture in order to prepare the system for automatic installation and autonomous operation, including elements of adaptability and self-configuration. Because the main functionality of the system was based on MBeans and agents implemented as JDMK MBean Servers, this idea was preserved and the two layers (instrumentation and the agent) were redesigned to use JMX Java extensions, which are almost fully JDMK-compatible[185]. After switching from Jini/Jiro to JMX we introduced a new name for the monitoring system: JIMS, i.e. the JMX-based Infrastructure Monitoring System. JIMS implementation altered resource discovery from the Jini Lookup Service to JDMK Dynamic Discovery. In the new system, resource instrumentation was performed using an implementation of the JMX version 1.2 specifications, which was only a supplement to JDK 1.4 at that time. Experiments involved WS access interfaces for the monitoring service using the SOAP Connector from the commercial WASP platform but a switch was then made to the open-source AXIS WS framework. The JIMS SG concept was also introduced for the purposes of mediating between SOAP-enabled clients and the monitoring service.

The second prototype had many advantages over the first one, i.e. fast dynamic discovery of monitoring agents in local clusters, static global discovery of clusters in the global network and fully automatic deployment of proper monitoring modules from the central module repository. The main disadvantages of this prototype, which necessitated further changes in the system, included closed APIs (taken mainly from JDMK, and Systinet/WASP[186]) and limited possibilities of JIMS integration due to the use of such APIs (WASP), which was the main obstacle that prevented the system from being widely used in a grid environment. The last important feature of the second prototype was the fact that on some nodes two different types of JIMS agents had to be installed simultaneously (separate agents providing the monitoring facility and agents with WS interfaces and dynamic discovery functions), causing additional resource consumption, not acceptable in a production environment.

5.1.3 Third Prototype and JIMS Extensions

As an enhancement of the second prototype, the last, third prototype of the monitoring service focused on non-intrusiveness, reliable self-configuration in the global network, rich visualization capabilities and open source implementation. Low intrusiveness was achieved thanks to special architectural design with one agent per node and by moving almost all functionality[187] to dynamically loaded modules. The concept of single-type monitoring agents

[184] Jiro was eventually abandoned and work instead focused on FMA, under the existing JINI [149,150] project. Jiro is no longer supported, and there is no longer any website on this technology the author could refer to.

[185] This incompatibility results from different package naming conventions. JMX carries some limitations in comparison to JDMK, in the area of dynamic discovery and SNMP interoperability.

[186] See p. 5.2.2.

[187] Regardless of whether it is a lightweight module for simple resource monitoring or a sophisticated module holding a specialized container[187] with a WS module deployed.

and encapsulating almost all functionality in separate modules allowed us to use only one virtual machine on each monitored node, resulting in a significant performance advantage. The prototype also remains open for future development and deployment in GT 4.0, which offers a container for certain types of Java modules, started within one, generally-accessible JVM on each monitored node. Performance profiling was another non-architectural aspect, which helped reduce resource usage[188]. From the client side, the functionality and interoperability was enhanced due to the use of the open source MX4J SOAP connector. The global discovery service was also extensively redesigned in order to be resistant to cluster failures and changing network conditions[189]. GDS implementation was exchanged for a mechanism supporting dynamical Global Registry selection. In order to avoid the use of proprietary software, the JDMK implementation of JMX was abandoned in favor of the JMX JSR 160-compatible implementation available in Java 5.0. The dynamic discovery mechanism was developed from scratch due to unavailability of such software, hence preserving JDMK compatibility at the API level. With the release of the Java 5.0 platform, it became possible to attach to and monitor running Java applications. This was the reason for JIMS to become an application monitoring system, which, in its third version, was also equipped with a facility for automatic discovery of applications. In the first attempt, this discovery feature was achieved by minor modifications of the applications themselves. Further work on monitoring Java applications showed that dynamic discovery of Java applications is possible using the special *Java agent* facility, which allows instrumentation of a Java application without any changes in the compiled code. Further related work on JIMS focuses on the area of visualization of monitoring data, resulting in a rich GUI (developed for monitoring of many grids from one management application), notifications on the addition of new nodes and clusters[190], and many other functions. As a side effect of the current JIMS architecture, we should also mention the high level of underlying environmental abstraction, which facilitates development of new monitoring modules. This means that the developer can focus on the monitoring sensor functionality rather than on operating system issues and that monitoring and configuration information, common to all operating systems, is available in one place and does not have to be retrieved from the OS or JVM directly[191].

[188] Mainly by lowering the excessive usage of Java Reflection APIs and by disabling functionality which was not needed in the production grid environment.

[189] Network disconnections and so-called *network partitioning* (dividing the grid into two or more separate parts).

[190] Notifications are sent using SOAP and Web Services, hence the presented solution carries a significant advantage, resulting from the use of the JMX connector architecture.

[191] Highly sophisticated configurations of grid infrastructure cause many problems while using OS-related info directly, even via JVM API.

5.2 JIMS Layered Architecture – Interfaces and Communication Modules

This section describes selected details of the implementation of JIMS layers, including instrumentation, interoperability, integration and the client application layer. According to this architecture and the selected JMX technology, the agent is modelled as an MBeanServer, and all monitoring resources and JIMS internal components are modelled as MBeans.

5.2.1 Instrumentation Layer

Instrumentation is the process of creating interfaces for the measurement of attributes of grid infrastructure elements. In JIMS this means that each of the monitored resources is equipped with a specialized MBean component which represents its state and enables access to monitoring information. The monitoring logic is encapsulated as MBeans. MBeans, which play the role of monitoring components, are called sensors. For the presented system, several such sensors were developed, including SystemInformation, SNMPMirror, NetworkMetrics, StorageMetrics, and GEMonitoring. All of them are briefly described in section 5.6.

Uniform Interface and Reflection API

The design and implementation of the JIMS instrumentation layer focused on uniform access to monitored resources. In order to provide flexibility, enabling the monitoring of any resources, which might appear in the future, while preserving one coherent interface, the author decided to use the reflection mechanism (p. 4.3.3) implemented in Java in the form of the Reflection API. Such an approach allows us to specify the monitoring and management interfaces using only *get()*, *set()* (to access attributes) and *invoke()* (to access any type of methods) operations[192]. The listings below illustrate the difference between classical method calls and calls using the Reflection API. Instead of directly reading the named attribute (Listing 5.1) or calling a particular method (Listing 5.2), the general methods *getAttribute* (Listing 5.3) and *invoke* (Listing 5.4) are used to access any type of attribute and to call any type of method.

```
SystemInformation.getL15m();
```

Listing 5.1 Reading monitoring attributes using the dedicated get method.

```
networkMetrics.measureICMPLatency(String destinationHostIP);
```

Listing 5.2 Calling monitoring methods using the dedicated measurement function.

[192] The autor is aware that this simple description omits two other elements of management interfaces, namely sensor constructors and notifications.

```
sgInvocation.getAttribute(systemInformation, "L1m");
```

Listing 5.3 Reading an attribute using the Reflection API[193].

```
String[] argTypes = { "java.lang.String" };
Object[] argValues = { ipAddress };
double icmpLatency =
      ((Double) sgInvocation
            .invoke(
                  mbsAddress,
                  networkMetrics,
                  "measureICMPLatency",
                  argValues,
                  argTypes))
            .doubleValue();
```

Listing 5.4 Calling a monitoring method using the Reflection API[194].

The Reflection API is similar[195] to DII invocations used in distributed systems, it allows invocation of any methods and access to any possible attributes, which is is why all JMX connector specifications use this approach. However, the Reflection API does not concern distributed systems directly. Through the usage of JMX connectors, the monitoring service can be extended in order to be remotely accessible, using many different network protocols (RMI/IIOP, WS/SOAP). In order to accommodate the requirement for interoperability between the monitoring service and the consumers of monitored parameters, the Reflection API can also be used for defining WS interfaces. Such an interface can be defined manually, or it can result from the use of a specialized JMX SOAP connector – an ordinary WS with a JMX connector interface, supporting stateful communication (it carries a connection identifier) and reflection methods (it allows to *invoke* a method, or *set*/*get* an attribute). Although such connectors are outside of the scope of the JMX Remote specification[196] [145], there are many open source implementations, which conform to the JMX Connector Architecture and can be used in the monitoring service. The concept of using such a connector as an interface for the monitoring system bases on the interoperability of the SOAP protocol over HTTP. One such openimplementations of a SOAP connector for JMX is MX4J, which is a complete package, ready for use in wide area networks, featuring a connector with all functions available in its RMI version, with nontrivial functionality (such as connection tracking, security, notification handling, etc.) Although its authors did not foresee the application of their connector in domains other than JMX, it was proved that such connectors can be successfully used in a heterogeneous environment, with other programming languages,

[193] The reflection method is typed in boldface.

[194] See the above note.

[195] The Reflection API offers broader functionality.

[196] This was the case at the time of submitting this dissertation. However, there is known a Sun effort aiming to standardize SOAP communication and to specify the JMX SOAP Connector API.

such as Perl and C/C++. The listings below show a sample of method call in the C[197] language (using the gSOAP API), including the connection establishment phase (required in JMX, Listing 5.5), and *getAttributes()* method calls for four monitoring parameters: *L1m*, *L5m*, *L15m* and *Mem*[198], shown in Listing 5.6. Following successful communication, the client closes the connection (Listing 5.7).

```
// connect to web service
char * _connectReturn;
soap_call_ns1__connect(&soap, argv[1], "", NULL, &_connectReturn);
printf("connection id=\"%s\"\n", _connectReturn);
```

Listing 5.5 Connecting to JMX SOAP Connector server[199].

```
// get attributes from SystemInformation module
char * attrs[] = {"L1m", "L5m", "L15m", "Mem"};
oname.canonicalName = "Monitoring:class=SystemInformation";
soap.header->ns1__connectionId = _connectReturn;
soap_call_ns1__getAttributes(&soap, "http://10.0.128.9:7702/jmxconnector",
   "", &oname, &_attributes, NULL, &response);
struct ns1__AttributeList * attr_list=response._getAttributesReturn;
printf("attributes count: %d\n", attr_list->__sizeitem);
for (int i = 0; i < attr_list->__sizeitem; i++) {
   printf("\t%s = %s\n",
      attr_list->item[i].name,
      attr_list->item[i].value);
}
```

Listing 5.6 Reading monitoring attributes using client written in C/gSOAP[200].

```
// disconnect
struct ns1__closeResponse cres;
soap.header->ns1__connectionId = _connectReturn;
soap_call_ns1__close(&soap, argv[3], "", &cres);
printf("connection closed\n");
```

Listing 5.7 Disconnecting from JMX SOAP Connector server[201].

The code above (Listings 5.5 – 5.7) shows that it is possible to access the JIMS management interface exposed using the Reflection API by means of a gSOAP library designed for C/C++-based applications[202].

[197] This example uses the gSOAP library for C/C++ development of SOAP-enabled applications.

[198] Respectively: CPU load during the last minute (*L1m* parameter), five minutes (*L5m* parameter) and fifteen minutes (*L15m* parameter), and the amount of free memory in MB (*Mem* parameter).

[199] The listing is truncated (SOAP header structure allocation was omitted).

[200] See above note.

[201] See above note.

[202] However, such a solution requires well-defined mapping from general types used in introspection to certain, well-defined types used by the gSOAP client.

5.2.2 Interoperability Layer

The main component of the interoperability layer, JIMS SG, was designed as a proxy Web Service running within the WS container on a cluster access node. For the initial implementation as a WS container, **WASP** (Web Applications and Services Platform) from Systinet[203] [156] was chosen. The WASP server scales well in a multithreaded environment, where many concurrent clients access the same WS [1]. However, WASP is a commercial product[204] and as such, in the following release of JIMS it was changed to AXIS and MX4J.

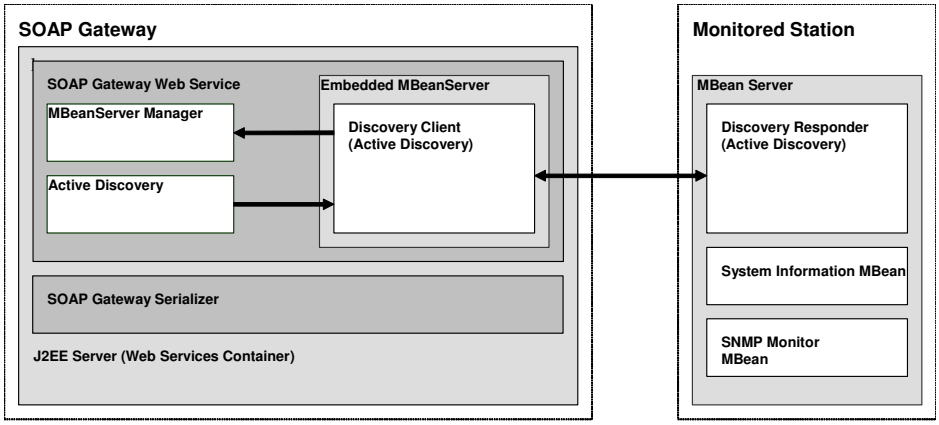


Figure 5.2 SOAP Gateway design, WS container and Active Discovery.

In the first attempt, SG was designed as a special type of monitoring agent (Figure 5.2). It was built as SG WS with an embedded MBeanServer Manager, which held the registry of all monitored stations' JMX addresses. The SG also encapsulates the active discovery mechanism, responsible for triggering discovery searches of monitored stations inside the clusters. Figure 5.2 also shows the SG Serializer component, responsible for marshalling complex objects that do not have default constructors[205].

Some enhancements to the presented SG implementation were introduced in the third JIMS prototype, which strictly adheres to the JMX Connector Architecture. In this prototype, WS functionality was moved from WASP to the open source implementation of the JMX SOAP Connector, taken from MX4J. MX4J SOAP Connector provides many other monitoring

[203] Systinet specializes in development of containers for WS applications written in Java and C++.

[204] WASP was chosen because it was free for stations with one processor. This was not an issue, because the WASP server did not have to be deployed inside the cluster, but on any station which has free access (i.e. with RMI communication not blocked) to all monitored stations. However, for use in open-source projects, where cluster access nodes cover many CPUs, the SOAP Gateway implemnetation was redesigned in order to not be dependent on commercial software.

[205] A newer version of JIMS uses MBean serializers and other ready-to-use classes of serializers available in MX4J and JMX. Such serializers have to be specified in the configuration file. However, some problems still occur while using serializers in JIMS. They are related to the problem of Java class loaders, which is not yet solved by current releases of JMX.

possibilities, such as stateful communication[206], notifications[207] and security[208] (covered in section 5.7). From the communicational point of view, the SOAP Connector is a type of WS; this enables integration of the monitoring service with almost any grid service, since WS bindings are available for all commonly used languages, i.e. C/C++, Java and Perl[209].

The interoperability layer can be seen as a set of well-defined interfaces and standards for accessing the monitoring service using different protocols. In JIMS, this functionality is provided by standard JMX connectors and protocol adaptors available in the JMX architecture. In the described monitoring service two types of connectors were tested: a SOAP connector and an RMI connector, allowing us to access monitoring modules and monitoring system internals in a transparent way. Concerning protocol adaptors, the JIMS system can be accessed by SNMP adaptors (for JIMS connectivity with SNMP-enabled network management stations), and HTML adaptors (for JIMS access using a simple web browser). Concerning the WS interface, communication via SG results in exchanging proper SOAP envelopes between the client monitoring application and the JIMS WS server in SG. Sample envelopes are shown in appendices (Listings A.8.1–A.8.8[210]). The listings are rather self-explanatory and show how the Reflection API can be used along with the WS technology, allowing the user to operate on any type of management object attributes and to invoke any type of management methods.

JIMS Interoperability – Protocol Adaptors and Connectors

Another role of the interoperability layer is to mediate between applications using different management protocols (SNMP, HTTP/WBEM, RMI, Burlap[211], Hessian[212]) and the JIMS system. For the JIMS system, just as for any JMX-based system, there two types of components are available and can be used for JIMS integration with other systems: protocol adaptors and connectors. As shown in Figure 2.8, adaptors are single components, which act

[206] Each connection has to be established and has its own unique identifier.

[207] JMX Connectors (including the MX4J SOAP Connector) allow clients to subscribe to notifications. This is interesting because WS are typically oriented in one direction (methods are called on the server side); however, the connector allows communication in the opposite direction (also using WS).

[208] Addition of security is very simple in the case of MX4J and is done by specifying another type of transport – in this case the **TLS** (Transport Layer Security) 1.0.

[209] All were tested in the context of JIMS access. For example, WS Perl and WS Java bindings were used by Postprocessing and Benchmarking in the CrossGrid project (respectively), and C/C++bindings (using gSOAP) were used to communictate with the data access optimization component in the CLUSTERIX project. The C client is faster than the one written in Java and was also used for testing purposes. Some results of these tests are presented in Chapter 6.

[210] Listings shortened, i.e. with removed links to XML schema definitions, namespaces, encoding, etc.

[211] A protocol that allows communication between JavaBeans and other applications, expressed in XML. It allows Java EJB (Enterprise Java Beans) services to interoperate with non-Java servers and clients using an XML-based protocol.

[212] Binary WS protocol that provides a messaging service layered on top of RPC calls.

as servers for the management application and provide the JIMS management interface via specific protocols. On the other hand, connectors are two-end entities, with both server and client sides, facilitating development of applications using the JMX Connector interface.

SNMP Adaptor

One of the main components that can be used to integrate JIMS with legacy management applications is the SNMP adaptor. It can be used for remote access to JIMS monitoring modules and its internals using the SNMP protocol. As an adaptor, it does not contain any additional logic (unlike SG) and therefore it is placed in the integration layer (Figure 5.3). Located inside the SG, the SNMP adaptor can work as an additional protocol translator for the JIMS SOAP gateway.

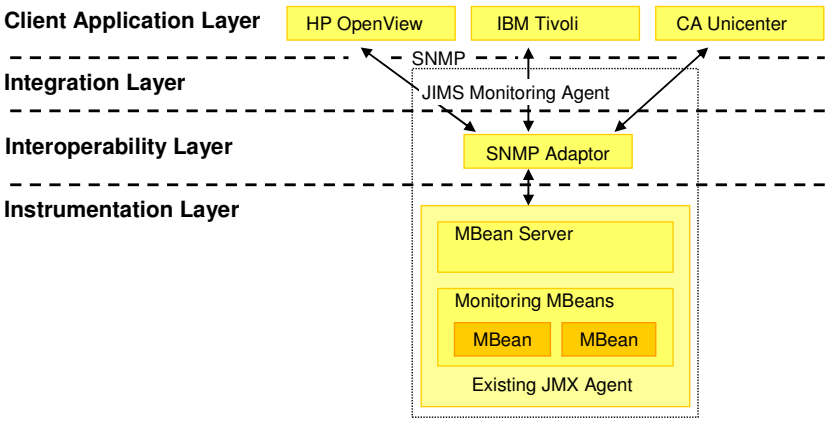


Figure 5.3 SNMP adaptor for the JIMS monitoring agent (based on [1]).

The SNMP adaptor should not be confused with SNMP proxies deployed in the instrumentation layer, which (on the contrary) use the SNMP protocol to access and monitor SNMP-enabled devices and exposes them via JIMS interoperable WS-based client interfaces[213].

HTML Adaptor

The HTML adaptor is another component, usable for prototyping and development. It provides a transparent view of the deployed MBeans and allows users to read and write some[214] of the exposed MBean attributes, as well as to invoke a limited[215] set of operations using a simple web browser.

[213] An example of such a component is the JIMS SNMPMirror sensor, implemented as a Westhawk SNMP Java client.

[214] The availability of MBean attributes for management depends on the capabilities of particular HTML adaptors and on the type of MBean attributes themselves.

[215] Due to the limited set of parameter types accepted by the MBean HTML adaptor.

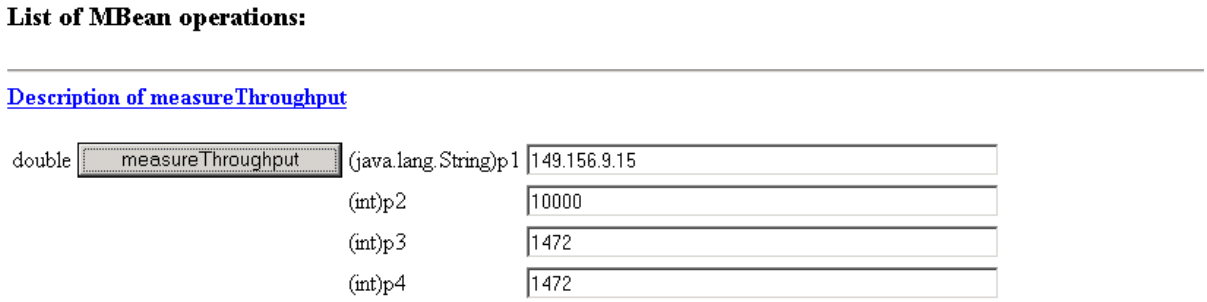


Figure 5.4 JIMS web interface – advanced network measurements.

The graphical view of a sample MBean HTML adaptor depicted in Figure 5.4 shows the *measureThroughput* method available in the JIMS NetworkMetrics sensor (which itself isn't shown in the picture). The method handles four parameters, available for passing through the management web page of the HTML adaptor: the IP address of the destination node, the number of tests and the number bytes sent *to* and *from* the destination node. This web interface is generated automatically, based on the introspection mechanism (the discovery of MBean facilities: attributes, operations, notifications, etc.)

RMI Connector

For integration with external systems based on CORBA and IIOP, the JMX RMI Connector can be used, exploiting the IIOP transport protocol (as opposed to Sun's proprietary binary protocol, which can be used for RMI communication as well). This binary connector is available in each JIMS monitoring agent and can be used to connect it to Java applications. It is accessible from any place in the grid because it is also located on cluster access nodes equipped with public IP addresses. Additionally, JIMS RMI connectors can be used for Java applications operating in local clusters. The connector can be exploited in conjunction with the dynamic discovery protocol to automatically build detectable JIMS agents, representing – for example – new, virtualized operating system resources (such as Solaris zones), as well as specially designed applications.

SOAP Connector and Web Services

The JMX SOAP Connector presents the simplest way of providing a standard WS-compliant interface for the monitoring service. Due to the use of the Java Reflection API, the interface is flexible enough to handle any type of monitoring and management operation, without changes following addition of new modules. Current releases of open-source frameworks (such as Axis 2 and .NET 2.0) indicate that this is currently the easiest and most common technology used for systems integration. The performance of new WS frameworks is also improving rapidly (for example, Axis 2.0 and .NET 2.0 perform about five times faster then their predecessors – Axis 1.0 and .NET 1.0 [136,82,87]).

To summarize, the integration layer provides global discovery functionality along with protocol adaptor components and connectors. SNMP or HTTP/HTML adaptors can be used for interactive access to monitored and managed resources without passing through the JIMS client application layer. The connectors (supporting such protocols as RMI, SOAP and many others – e.g. binary Burlap or Hessian) are used for integration by applications and can be viewed as another way of accessing the JIMS monitoring API.

5.2.3 Integration Layer

The main function of the integration layer is the discovery of active clusters in the grid. Two possible solutions were incorporated into the JIMS system for these purposes: the static Global Discovery Service mechanism and the dynamically changing set of global registry nodes available in the second prototype of the GDS service. In the first case, the integration layer consists of a certain number (typically three) of chosen SOAP gateways performing the role of GDS. GDS node designation is reflected in the configuration files for SG agents in all clusters. In JIMS, GDS was developed as a module for the existing JMX-based monitoring system. There are no counterindications to having the same GDS module in all SGs in the grid. Furthermore, all SGs must include the GDS module in order to perform registration in a set of well-known GDS nodes. Though not all CE agents are GDS nodes, all of them maintain lists of GDS nodes and can also maintain lists of registered SOAP gateways [23]. In the latter case, it is assumed that one of the GDS nodes acts as a Global Registry (GR) and stores information about the available clusters in the grid. Election of a GR is performed dynamically and, in case of unavailability or network communication failure, a new GR is elected. The protocol also supports situations where all clusters are raised at the same moment and no GR is selected yet. In such a case (similar to the one where a GR fails to respond to heartbeats, i.e. a mechanism that “refreshes” the registered references of GDS nodes), an election algorithm is triggered and a new GR is chosen. [173]. More details on these two mechanisms can be found in section 5.4.

5.2.4 Client Application Layer

Client interfaces range from programming interfaces (API) available for developers, to simple text-based command-line interface (CLI) applications communicating with the SOAP protocol, to applications offering rich GUI interfaces, with graphical charts and visualization of the monitored infrastructure. Within the client application layer, we can also mention the view rendered by the previously described JIMS HTML adaptor.

JIMS CLI

As an example of CLI applications, several types of WS-based clients were developed for the JIMS system. Due to the need for development of different types of client applications for different technological domains, some command line applications were created, aiming at testing JIMS connectivity. These applications were written in different languages, such as Perl, C/C++ and Java, and all of them use SOAP as their access protocol.

In general, the CLI interface can be divided into two classes: it can be a batch application executing one command (Listing 5.8) or a batch application executing a set of commands entered in the batch script file (Listing 5.9). It the second case, it can preserve a single communication session (i.e. a single connection identifier to SG as well as other connection-related data[216]) during invocations of several internal commands (Listing 5.10).

```
$ jims-client -g zeus24.cyf-kr.edu.pl -l
[01] 149.156.9.15
[02] 149.156.9.16
...
[09] 149.156.9.34
```

Listing 5.8 Listing computational nodes using JIMS batch CLI[217].

```
echo ********** TESTING ZEUS ********************************
echo off
connect zeus24.cyf-kr.edu.pl
module SNMPMirror
echo on
echo WNs (hostnames):
get sysName
module SystemInformation
echo CPU statistics:
cpustats
echo Memory [MB]:
get Maxmem
echo ********** TESTING FZK ********************************
echo off
connect ce010.fzk.de
module SNMPMirror
echo on
echo WNs (hostnames):
get sysName
module SystemInformation
echo CPU statistics:
cpustats
echo Memory [MB]:
get Maxmem
```

Listing 5.9 Batch script file created for JIMS testing purposes.

```
$ jims-cli
JIMS>connect zeus24.cyf-kr.edu.pl
Connection prepared for: zeus24.cyf-kr.edu.pl
JIMS>
```

Listing 5.10 JIMS CLI interactive client connecting to the JIMS SOAP gateway[218].

Output of the command line interface can also be formatted according to the predefined XML schema and can be further used for interoperability with other systems, integrated through

[216] Such as authentication credentials.

[217] The –g option sets the address of the SOAP Gateway corresponding to the cluster the client wants to connect to. The –l option enables listing IP addresses of all available worker nodes.

[218] Listing is truncated for editorial reasons.

command-line interfaces (as in CLUSTERIX [95]). An example of such XML output is presented in Listing 5.11.

```
<jims>
        <desc>JIMS CLI v. 1.5.24, executing clx-cpu.cmd</desc>
        <wn nr="1" ipaddress="149.156.9.16">
                <user cputime="200">
                        <ncpu nr="0">99</ncpu>
                        <ncpu nr="1">101</ncpu>
                </user>
                <nice cputime="0">
                        <ncpu nr="0">0</ncpu>
                        <ncpu nr="1">0</ncpu>
                </nice>
                <idle cputime="0">
                        <ncpu nr="0">0</ncpu>
                        <ncpu nr="1">0</ncpu>
                </idle>
                <system cputime="2">
                        <ncpu nr="0">2</ncpu>
                        <ncpu nr="1">0</ncpu>
                </system>
        </wn>
        <wn nr="2" ipaddress="149.156.9.17">
                <user cputime="102">
                        <ncpu nr="0">0</ncpu>
                        <ncpu nr="1">102</ncpu>
                </user>
                <nice cputime="0">
                        <ncpu nr="0">0</ncpu>
                        <ncpu nr="1">0</ncpu>
                </nice>
                <idle cputime="101">
                        <ncpu nr="0">101</ncpu>
                        <ncpu nr="1">0</ncpu>
                </idle>
                <system cputime="0">
                        <ncpu nr="0">0</ncpu>
                        <ncpu nr="1">0</ncpu>
                </system>
        </wn>
        ...
</jims>
```

Listing 5.11 XML output generated by JIMS CLI[219].

For the end user, an interactive[220] client application is the preferred way of accessing the JIMS monitoring service. It can deliver information about all WNs discovered by the given SOAP gateway, their CPUs usage and network latency between a chosen WN and all other WNs. It can also provide a list of all other available SOAP gateways so that the user can

[219] See above note.

[220] The term *interactive* is used in the sense described before, i.e. it means an application which preserves the session between consecutive monitoring requests (internal commands). Such an application is contrasted with one that is executed many times for each monitoring request.

switch to another gateway and view details of any other cluster. An interactive application does not require any initial configuration because the SOAP gateway which the client connects to can be changed during the session. An advantage of the interactive command line interface is faster operation: the time-consuming process of binding with JIMS WS is performed only once and all subsequent commands can be executed without introducing additional overhead. The available internal commands allow the user to connect[221] to the selected SOAP gateway, list available monitored stations (Listing 5.12), display CPU statistics (Listing 5.13), and measure chosen network parameters (such as latency and throughput from the chosen node to all computational nodes, using various network protocols, i.e. UDP or ICMP), as shown in Listing 5.14.

```
JIMS>list
[01] 149.156.9.15
[02] 149.156.9.16
...
[09] 149.156.9.34
JIMS>
```

Listing 5.12 Listing nodes discovered by the SG using an interactive CLI interface.

```
kbalos@access:~$ cg-jims-cli
JIMS CLI v. 2.0.60, type "help" for help
JIMS>connect 10.0.128.9
Connection prepared for: 10.0.128.9
JIMS>cpustats
TIME/CPU0,1,... USER NICE IDLE SYSTEM [hs/s]:
[01] 10.0.128.1:  2[   0    2]   0[   0    0] 2052[1029 1023]  4[   0    4]
[02] 10.0.128.2:  0[   0    0]   0[   0    0] 2055[1030 1008]  3[   0    4]
...
[09] 10.0.128.9:  0[  12    4]   0[   0    0]  184[  88   96]  0[   0    0]
JIMS>
```

Listing 5.13 Displaying cluster CPU load[222] using WS-based JIMS CLI.

```
kbalos@access:~$ cg-jims-cli
JIMS CLI v. 2.0.60, type "help" for help
JIMS>connect 10.0.128.9
Connection prepared for: 10.0.128.9
JIMS>netstats 149.156.9.15
[01]149.156.9.15:ICMP: 0.029[ms], UDP: 0.2[ms], Throughput: 8.432E7 [bit/s]
[02]149.156.9.16:ICMP: 0.143[ms], UDP: 0.3[ms], Throughput: 3.372E7 [bit/s]
...
[09]149.156.9.34:ICMP: 0.16 [ms], UDP:0.35[ms], Throughput: 1.873E7 [bit/s]
```

Listing 5.14 Network latency and throughput between the chosen node and local WNs[223].

[221] An example shows the connection to the SOAP gateway running on zeus24.cyf-kr.edu.pl CE (one of the CEs available in CrossGrid).

[222] Four types of parameters are displayed: user, nice, idle, system (for linux 2.6 series kernel additional parameters include io, hardirq and softirq). The first column represents hundreds of seconds spent on tasks in user mode, while the following columns cover nice, idle and system modes respectively. Values in square brackets indicate time spent on these tasks by individual CPUs, so the user can view how computation is balanced between many CPUs.

JIMS GUI

The JIMS monitoring service can be accessed by any type of JMX management console (such as MC4J or JConsole), communicating via the SOAP protocol. However, all consoles allow connecting only to one monitored node and are not useful for practical grid management and monitoring tasks. In order to provide the view of the grid as a whole, a specialized application was developed, which allows users to connect to the JIMS monitoring service via publicly available SGs. The client can display a list of available monitored clusters and a list of monitored nodes in each cluster. By using the JIMS GUI, the user can browse monitoring agents and view the properties of monitoring system modules (sensors) providing information about job managemnet systems installed in the cluster (see Figure 5.5), OS details, hardware, CPU utilization (see Figure 5.6), file system statistics, network interfaces, throughput and bandwidth, disk usage, etc.

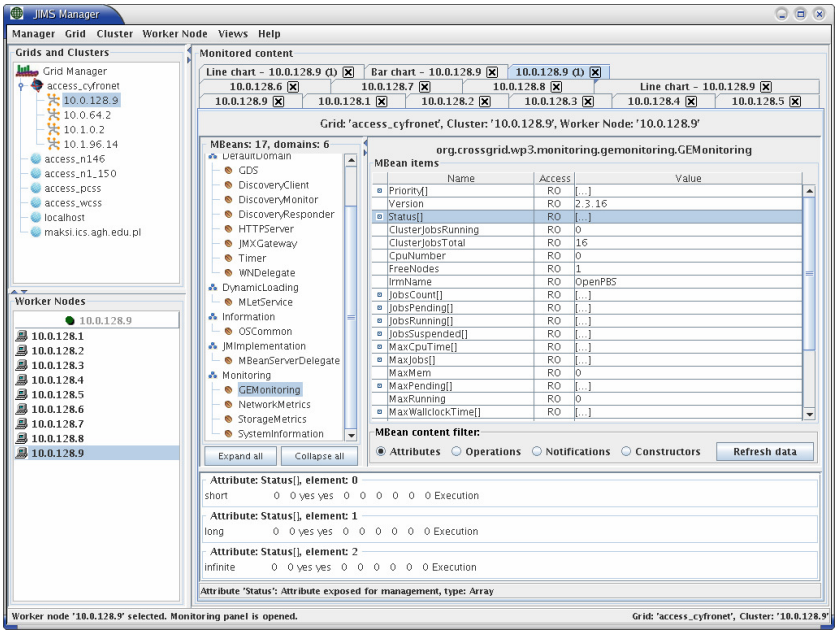


Figure 5.5 JIMS GUI leveraging WS (the Grid Engine Monitoring module) [172].

[223] The figure shows network latency between the chosen WN (149.156.9.15) and all others WNs (149.156.97.15-*.34). By specifying different IP addresses, the user can evaluate the latency (separately for ICMP and UDP protocols) and throughput (using the UDP protocol) between nodes in almost all available configurations.

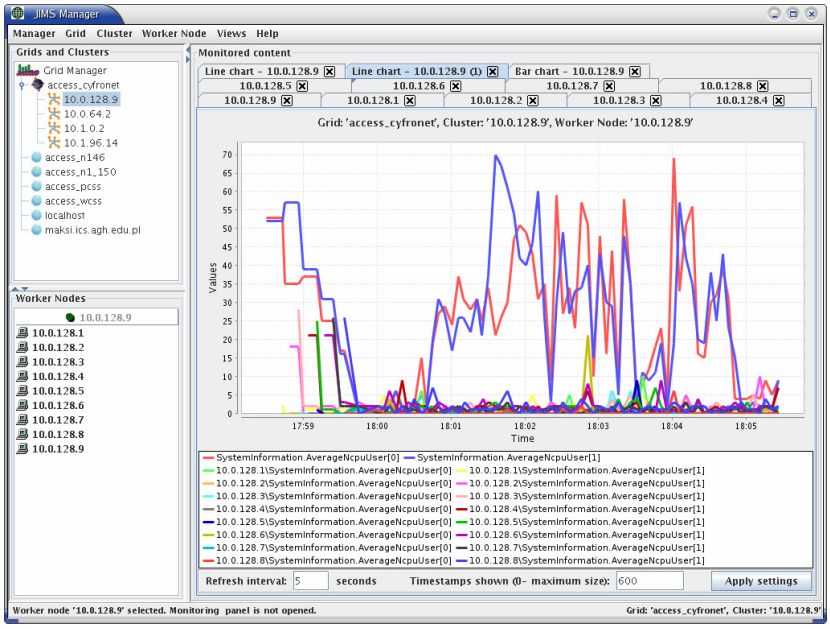


Figure 5.6 JIMS GUI leveraging WS (the System Information module) [172].

In the presented figures, the top left-hand part of the application window shows the available clusters, the bottom left-hand part shows the list of available monitored computational nodes in the cluster, while the main panel shows the CPU load in *user* mode in all CPUs of all WNs over the last 10 minutes. The presented application enables users to connect to the JIMS monitoring system through any available SOAP gateways, which are deployed on all access nodes available in the grid, thus providing the required level of system reliability.

5.3 JIMS Components as Managed Beans

The general definition of the term *component* was already presented in p. 1.3.1. However, for system description purposes, the author would like to narrow down this definition in the context of the chosen JMX technology and the presented JIMS system. In the context of JIMS, component is understood as a logical entity inside a software module (see the definition of *module* in the next paragraph), and is represented by the semantics of the compiled MBean code[224]. The component is described by the interfaces it exposes and a deployment descriptor (the *m-let* file[225]), which comes with the installable module. Such a descriptor contains at least the name of the component instance and specifies other parameters, e.g. constructor arguments, class name implementing the management interface, etc.

JIMS components can be grouped into six categories (domains), as shown in Table 5.1. The first group, called “Connector”, contains connectors that can be deployed in the monitoring

[224] See p. 2.6.4.

[225] See above note.

agent in order to facilitate remote access to monitored data. The “Core” contains discovery-related MBeans (local and global discovery MBeans), MBeans playing role of gateways (JMXGateway, WNDelegate) and other helper components (JMX Timer, HTTPServer). The third group contains only one component (MLetService) and is responsible for dynamic instantiation of MBean components encapsulated in binary modules, and transferred from a remote HTTP repository. The “Information” domain contains informational components, holding data common to all JIMS monitoring agents. “JMImplementation” is a JMX internal MBean domain. The last domain, “Monitoring”, contains all essential monitoring modules (SystemInformation, NetworkMetrics, etc.)

Table 5.1 MBeans used by the monitoring service.

Domain	**MBean**	**Function**
Connector	RMIConnectorServer	RMI Connector
	SOAPConnectorServer	SOAP Connector
Core	GDS	Global Discovery Service
	DiscoveryClient	Dynamic Discovery Client
	DiscoveryMonitor	Dynamic Discovery Monitor
	DiscoveryResponder	Dynamic Discovery Responder
	HTTPServer	HTTP Server for module repository
	JMXGateway	Gateway responsible for managing the list of computing nodes
	Timer	Timer for sensor modules
	WNDelegate	Delegates requests from SG to WNs
DynamicLoading	MLetService	Module supporting dynamic loading
Information	OSCommon	Common parameters of JMX agent
JMImplementation	MBeanServerDelegate	JMX implementation related MBean
Monitoring	NetworkMetrics	Module for network monitoring and measurements
	GEMonitoring	Grid Engine monitoring module
	SNMPMirror	Module for collecting data through SNMP
	StorageMetrics	Module for storage monitoring and measurements
	SystemInformation	Infrastructure monitoring module

The collaboration and dependencies between all these modules are shown in Figure 5.7. Components on the left-hand side are basic JIMS agent components, loaded from the local filesystem during startup. They provide minimal functionality required by the agent to perform further functions, i.e. loading and deployment of monitoring modules.

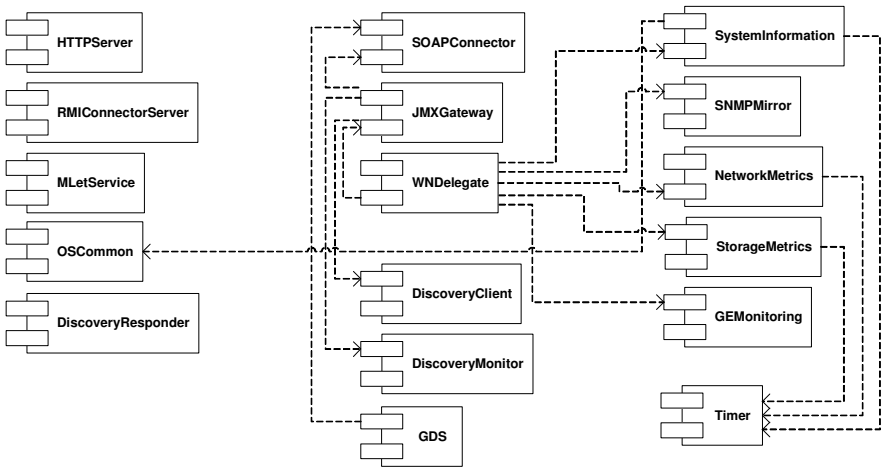


Figure 5.7 Diagram of dependencies between components of the monitoring service.

The HTTPServer module is responsible for serving static files: modules and their XML descriptors. RMIConnectorServer allows users to remotely and safely[226] manage and monitor agent internals and monitored resources. The MLetService module is a standard JMX component equipped with the dynamic loading facility; it enables access to dynamic modules loaded from remote HTTP servers. The next module, OSCommon, provides additional, common, parameters required by other monitoring modules. Among others, the two most important attributes of this module, are the agent's registry value and its IP address (see p. 5.3.1). The last module, DiscoveryResponder, is responsible for replying to active discovery requests and exposing the agent's RMIConnectorServer JMX address. This facility enables discovering agents without a naming service. All other components are loaded dynamically after agent startup, according to the requirements in given environment.

Components presented on the right-hand side of the picture include all viable monitoring modules, such as the SystemInformation MBean for delivery of CPU- and memory-related information, the SNMPMirror MBean for delivery of information received using the SNMP protocol, the NetworkMetrics MBean for obtaining statistics and performing some measurements in the network, the StorageMetrics MBean for monitoring grid storage elements, and the GEMonitoring[227] MBean for monitoring job management systems such as SGE [152] or PBS [167]. Most of these components make use of an additional JMX service –

[226] Using TLS 1.0 secured RMI connections.

[227] The full name: GridEngineMonitoring.

the Timer component, which is responsible for synchronizing statistics about average utilization of particular infrastructure resources: CPU, network interfaces, disks, etc.

The most interesting functionality is provided by components shown between the described elements, i.e. the SOAPConnector, JMXGateway, WNDelegate, DiscoveryClient, DiscoveryMonitor and GDS MBeans. The first component is an open source implementation [178] of the SOAP connector, conforming to the JMX connector architecture. The role of this component is to provide an interoperable interface for consumers of monitored information. Like all JMX connectors, it does not directly depend on any other module because it provides a transparent view of all installed MBeans to remote JMX clients. JMXGateway is a key component for the dynamic discovery featur; it holds a special element responsible for managing connections for all other monitoring agents. Whenever a new agent appears, this component adds it to the list of registered connectors, and, conversely, if an agent does not respond to periodically sent hearbeat requests, it is removed from the list. WNDelegate makes use of the list of connectors managed by JMXGateway and delegates monitoring requests to all connected monitoring agents. It allows clients of the monitoring system to transport monitoring information more efficiently, using one request to retrieve a common parameter from all monitored nodes. DiscoveryClient performs active discovery and dynamically locates all agents running within the scope of a given multicast group. DiscoveryMonitor provides the passive discovery facility, which is currently not used in this monitoring service, but is available for potential future applications[228]. The lastcomponent, GDS, is responsible for performing SG registration with GR operating at a certain point of the wide area network. It communicates with other GDS nodes via WS and delivers lists of currently available monitored clusters, assuming that each cluster has one SOAP Gateway started and running.

5.3.1 Elements of Adaptability

In JIMS, adaptability is understood as the agent's ability to configure itself by adjusting its functionality to the changing environment conditions. As described in Chapter 4 (section 4.5), JIMS offers two types of adaptability, both implemented and tested in real grids. The first type involves the way the agent decides which modules to load during startup. The second type covers adaptability of the functionality inside the monitoring sensors during JIMS runtime[229].

[228] Passive Discovery, as less reliable than Active Discovery, is disabled in this JIMS release.

[229] In addition to the described adaptability features, there is ongoing research in DSRG on components facilitating dynamic interfaces (implemented as Dynamic MBeans) for monitoring dynamically changing Solaris containers.

Startup Time Adaptability

Separation of environment testing and exact agent's adaptability[230] (see p. 4.5.2) was implemented using a shell script, which gathers information about the infrastructure elements, which should be monitored, and the role of the monitoring agent itself. The script then passes these facts as bit registry flags to the agent's process. The proposed structure of such a registry is shown in Figure 5.8.

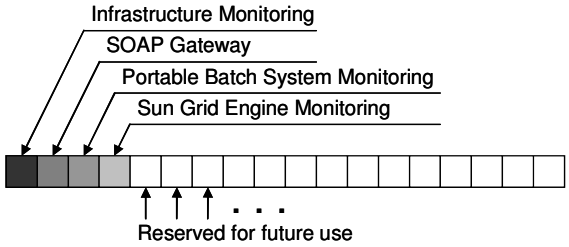


Figure 5.8 Structure of JIMS configuration registry.

All indicators in the registry are mutually independent, i.e. they can occur simultaneously: this applies to the general infrastructure monitoring facility, the gateway facility, the PBS monitoring facility and the SGE monitoring facility. Other important types of information gathered by this shell script include IP address of the host, which the agent is starting on, and the hostname of the module repository. The full list (excluding JVM-related parameters) of base arguments passed to the monitoring agent consists of three values shown in Listing 5.15. The first value is the IP address of the monitoring agent, the second is the IP address of the repository of modules[231], and the last one is the JIMS registry value[232].

```
java -cp $CLASSPATH \
org.crossgrid.wp3.monitoring.jims.JIMSAgent \
10.0.128.1 $REPOSITORY $REGISTRY
```

Listing 5.15 Configuration parameters passed to the agent during startup.

The agent then passes all these values as arguments to the constructor of the OSCommon component, holding this information for further use by other monitoring modules (Listing 5.16).

```
loadModule("jims.agent.mbean.oscommon");
setAttribute("jims.agent.mbean.oscommon", "HostIP", hostIP);
setAttribute("jims.agent.mbean.oscommon", "RepositoryIP", RepositoryIP);
setAttribute("jims.agent.mbean.oscommon", "JIMSRegistry", JIMSRegistry);
```

Listing 5.16 Configuration parameters passed to the agent during startup time[233].

[230] In other words, this type of adaptability is split into two stages. The first relies on learning about some facts in the surrounding environment, while the second is the strict adaptability.

[231] Passed as a value of the $REPOSITORY environment variable.

[232] Passed as a value of the $REGISTRY environment variable.

[233] The value shown in the listing, "jims.agent.mbean.oscommon" is the name of a property describing the real MBean name.

Having properly loaded the registry, the agent than checks these bit indicators, compares them with the internal map of registry flags (Listing 5.17)[234] and loads modules encapsulating the required functionality.

```
/** Constants indicating different functionalities discovered in cluster
/*  Indicate that agent should load monitoring specific modules */
public static int JIMS_CONST_MON=1;
/** Indicate that agent should load gateway specific modules */
public static int JIMS_CONST_GW=2;
/** Indicate that agent should load Grid Engine (PBS or SGE) specific
modules */
public static int JIMS_CONST_PBS=4;
public static int JIMS_CONST_SGE=8;
```

Listing 5.17 Configuration registry indicators known to the monitoring agent.

First, the gateway flag (JIMS_CONST_GW) is checked. If the agent needs to operate as a gateway, it loads the HTTP server module from the local filesystem and starts the HTTP module repository, then downloads the MLet descriptor (XML m-let file[235]) for modules required by the gateway agent. The list of downloaded modules can be customized and by default contains the JMXGateway and the GDS MBean. The agent then instantiates components according to the list in the MLet descriptor (JMXGateway and GDS). Dependencies are resolved by initialization of further components by the already-loaded components. For example, the JMXGateway component creates two other, dependent components: SOAPConnector and WNDelegate.

A second check is performed against the monitoring flag (JIMS_CONST_MON). If detected, operations are performed which are similar to the ones presented above. First, the agent downloads the monitoring descriptor (XML m-let file) from the repository. It should be noted that the selected m-let filename depends on the properties of the operating system and the hardware platform, so a proper descriptor is downloaded for each given monitored infrastructure. For example, if the operating system is called Linux, operating on an IA64 platform, a file called mbeanOSInfoLinux-ia64.mlet is downloaded. Descriptors can be prepared for different operating systems, so that the monitoring agent can operate on almost any available platform, if only the required monitoring modules are present (Listing 5.18). Subsequently, the agent instantiates all the components according to the list included in the MLet descriptor. By default, four monitoring modules are downloaded and instantiated: SystemInformation, SNMPMirror, NetworkMetrics and StorageMetrics MBeans.

[234] The semantics describe which functionality should be loaded by the monitoring agent if a particular bit is set.

[235] Here: mbeanSoapGateway.mlet.

```
this.loadModuleRemote(ceHost, "mbeanOSInfo"
            + System.getProperty("os.name") // typical value: os.name=Linux
            + "_"
            + System.getProperty("os.arch") // typical value: os.arch=i686
            + ".mlet");
```

Listing 5.18 Process of selection of proper monitoring module descriptor to download from the repository.

The last registry flags checked by the agent concern monitoring of job management systems (flags JIMS_CONST_PBS and JIMS_CONST_SGE), which involve loading of a related module descriptor[236] and instantiating components according to its content. By default, separate modules are deployed for SGE and PBS monitoring, which handle APIs for gathering information about job queues and other job-related parameters.

Runtime Adaptability

Runtime adaptability of the monitoring system is achieved thanks to the logic implemented withing the monitoring modules. For example, this logic allows us to distinguish between different versions of the IP protocol announced by the discovered remote monitoring agents. It also allows the system to apply special procedures to correctly handle IPv4 and IPv6 protocols.

```
// Test the type of the IP address (Is it IPv6?)
if((start = this.jmxConnectorAddress.indexOf("["))!=-1 &&
   (stop  = this.jmxConnectorAddress.indexOf("]"))!=-1)) {
      ipv6 = true;
}
// Handle IPv6-specific functionality
if(ipv6) {
      ...
}
// Handle IPv4-specific functionality
else {
      ...
}
```

Listing 5.19 Runtime adaptability logic inside the JIMS module; here: adaptability to IPv4 and IPv6 addresses of JMX connectors returned by the active discovery process.

Listing 5.19 is a fragment of the SG code responsible for registration of the discovered monitoring agents. If the incoming response is a JMX address of the IPv6 type, the SG agent can perform some IPv6-specific operations. In this case, the feature is coded inside the SG module and is an example of code responsible for JIMS runtime adaptability.

[236] Here: mbeanGE.mlet.

5.3.2 M-Let Service and the Dynamic Loading Facility

JIMS places significant emphasis on the installation and configuration process, which allows automatic deployment and upgrading of management modules, operating as helper services, such as the SOAP gateway[237] or the HTML adaptor[238]. Dynamic module loading is based on the JMX m-let service. It is a class included in a recent version of Java[239], which extends the classic remote class loader (see Figure 5.9) and makes use of a downloadable classes' descriptor, called the m-let file (see Listing 5.20). This technique allows preparation of modules which can be deployed (downloaded, installed and started) automatically, during startup or later. The m-let service also enables upgrading or installing newly developed modules as well as removing existing ones without restarting the whole monitoring system.

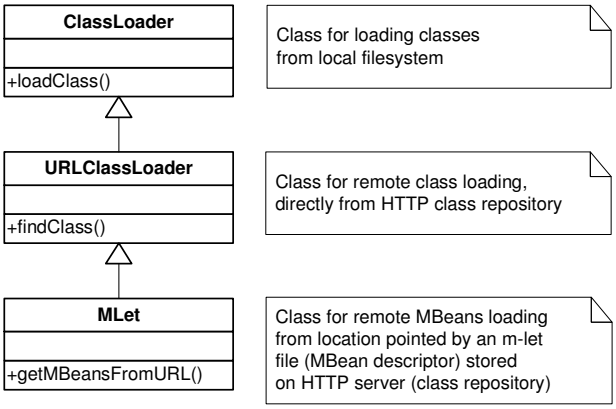


Figure 5.9 Simplified MLet class hierarchy[240].

By using the m-let service, the monitoring agent can instantiate and register one or several MBeans, downloaded from a remote URL, in the running MBean server. The m-let service does this by loading the m-let text file, which specifies information about the MBeans to be obtained. An URL specifies the location of the m-let text file. In the m-let file, the information about each MBean is specified by a single *mlet* tag (see Listing 5.20). The *code* attribute specifies the full Java class name, including the package name, of the MBean to be obtained. The compiled class file of the MBean must be contained in one of the **JAR** (Java Archive) files specified by the *archive* attribute. *Name* is an optional attribute and specifies the object name to be assigned to the MBean instance when the m-let service registers it. Another optional element is the *version* attribute. It specifies the version number of the MBean and associated JAR files to be obtained. This version number can be used to specify

[237] The SOAP gateway is a Web Service, thus fitting the definition of a helper service.

[238] The HTML adaptor is in fact a HTTP server which renders a HTML view of the installed management modules. The HTML adaptor also suits the definition of a JIMS helper service.

[239] Java TM 2 Platform Standard Edition 5.0.

[240] Actual inheritance hierarchy of the MLet class includes java.lang.Object as a base class for java.lang.ClassLoader. In the figure we also omitted the java.security.SecureClassLoader class, extending the standard ClassLoader and acting as a base class for java.net.URLClassLoader.

that the JAR loaded from the server should overwrite those stored locally in the cache the next time the m-let text file is loaded[241].

```
<mlet
code="org.crossgrid.wp3.monitoring.jims.mbeans.Linux.SystemInformation"
    archive="jims-osinfo-linux-i686.jar"
    name="Monitoring:class=SystemInformation"
    version="1.0">
</mlet>
<mlet
    code="org.crossgrid.wp3.monitoring.jims.snmp.mbean.SNMPMirror"
    archive="jims-snmp.jar,snmp.jar"
    name="Monitoring:class=SNMPMirror"
    version="1.0">
</mlet>
<mlet
    code="org.crossgrid.wp3.monitoring.jims.NetworkMetrics"
    archive="jims-networkmetrics.jar"
    name="Monitoring:class=NetworkMetrics"
    version="1.0">
</mlet>
<mlet
    code="org.crossgrid.wp3.monitoring.jims.StorageMetrics"
    archive="jims-storagemetrics.jar"
    name="Monitoring:class=StorageMetrics"
    version="1.0">
</mlet>
```

Listing 5.20 M-let file containing descriptions of four downloadable components.

The main idea behind the m-let service is based on first retrieving descriptions of components to be loaded, and then downloading real module resources (JAR files). By using this technique, the location of MBean archives can differ from the location of MBean descriptors[242], providing a higher level of flexibility when choosing the module repository.

A sample scenario for m-let service usage is shown in Figure 5.10. It shows the loading process of two sample MBean components: Object1 and Object2. Object1 is loaded in a traditional way from the local filesystem using standard class loaders, available in each JVM. A more interesting method of loading MBeans is presented for Object2. First, the MBeanServer invokes the for remote MBean loading (*getMBeansFromURL()*, see Listing 5.21) and downloads the mbean2.mlet file from the HTTP server running at `http://ce-host.org`[243].

[241] As an option, the *codebase* URL can also be specified, providing even more flexibility in specifying the location of the code to download.

[242] Text files which specify component behaviour in terms of well-defined XML tags.

[243] `ce.host` is an example of a node which hosts downloadable modules.

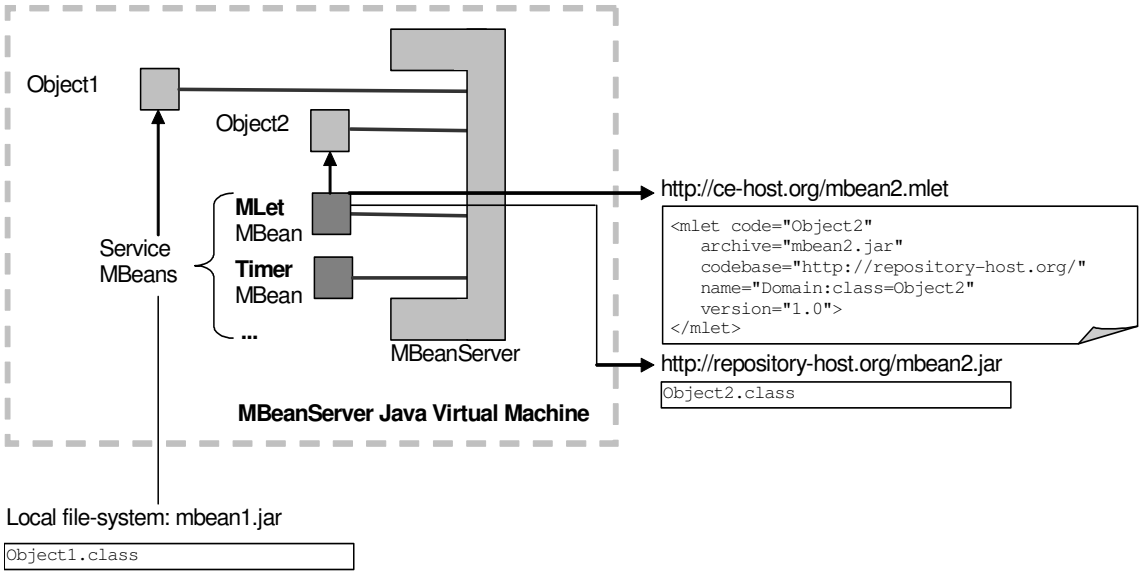


Figure 5.10 Dynamic module loading from a remote repository using JMX m-let facility.

```
MBeanServer server;
server = MBeanServerFactory.createMBeanServer();
server.invoke(
      new ObjectName("DefaultDomain:service=MLet"),
      "getMBeansFromURL",
      new Object[] { "http://" + ceHost + ":"+  cePort
                     + "/" + mletFileName },
      new String[] { "java.lang.String" });
```

Listing 5.21 Dynamic loading of modules using the MLet MBean and the Reflection API.

Next, the m-let component reads the contents of the file, and searches the JAR archive where the packed component is available for instantiation. The location can be either the same HTTP server (if the full codebase URL address is omitted), or another server, as in this case: `http://repository-host.org`. Given the location of the JAR archive, the MLet MBean downloads the archive, loads the Object2 class and instantiates the MBean with parameters included in the mbean2.mlet file (among others, the most important parameter is the name of the MBean object[244]). This ends the process of dynamic module loading. The components instantiated by the m-let service do not differ from other components loaded locally from the filesystem.

The m-let service and its dynamic deployment facility provide a base for the adaptability of the monitored service, described earlier in section 4.5. It is used for loading almost all of the monitoring agent functionality (SG, dynamic discovery component, all monitoring modules, etc.), except for the HTTP server component, which is an inherent part of the JIMS

[244] Typical full object name is "DomainName:{name,type,class}=Object", according to JMX component naming rules.

architecture and acts as a remote modules repository – thus it is typically loaded from the local filesystem.

5.4 Self-configurability

As discussed in p. 5.3.1, elements of adaptability only cover the configuration of the monitoring agent itself. The author clearly distinguishes this type of agent abilities from agent self-configurability, which only concerns a certain type of agents (the SGs) and is strictly connected with the process of their discovery in clusters and the grid.

JIMS self-configurability adheres to the general architecture of modern grids, i.e. it offers a two-level discovery mechanism, taking advantage of features of each of the two types of environments: clusters (usually with private addressing, hidden behind firewalls and NAT) and the grid (public network without multicast support).

5.4.1 Cluster-level Self-configurability

For the purpose of cluster-level self-configurability, two mechanisms were implemented: passive and active discovery. Comparison of these two mechanisms (see Table 4.1) suggests that an uncomplicated, reliable and dynamic method of dynamic discovery is provided by the active discovery mechainsm, and this is why active discovery is the default method in the production version of JIMS.

Passive Discovery

In the current implementation of the JIMS system two types of protocols are leveraged: a SOAP text-based protocol used by clients for interoperability purposes, and a binary RMI protocol for efficient communication inside local clusters. In the process of passive discovery, sent messages carry the addresses of node JMX RMI connectors. Sample RMI connector server (see p. 2.6.4) and discovery responder registration, along with the required configuration in WN, are shown in Listing 5.22.

```
ObjectName rmiServ = new ObjectName("Connector:name=RMIConnectorServer");
ObjectName drRes = new ObjectName("Core:name=DiscoveryResponder");
String jmxAddress = ((JMXServiceURL)server.getAttribute(
      rmiServ, "Address")).toString();
DiscoveryResponder responder = new DiscoveryResponder();
server.registerMBean(responder, drRes);
responder.setUserData(jmxAddress.getBytes());
responder.setMulticastPort(7706);
responder.setMulticastGroup("224.224.224.224");
responder.start();
```

Listing 5.22 Instantiation of Discovery Responder and RMI Connector Server[245].

[245] In the JDMK framework the default settings of the discovery responder include the multicast group 224.224.224.224, on port 9000. The shown listing is simplified for reasons of clarity – for example, in reality all parameters (i.e. the multicast IP address and port) are dynamically configurable (even during system runtime).

The discovery monitor is installed on the SG side (an example of instantiating and configuring the discovery monitor MBean is shown in Listing 5.23). This monitor listens for the discovery responders' responses.

```
ObjectName name = new ObjectName("Core:name=DiscoveryMonitor");
DiscoveryMonitor dm = new DiscoveryMonitor("224.224.224.224", 7706);
server.registerMBean(dm, name);
dm.start();
```

Listing 5.23 Instantiating and initializing a Discovery Monitor component[246].

Active Discovery

In active discovery scenario, the search action is invoked by the discovery client (an example of instantiation and initialization of a discovery client component is shown in Listing 5.24).

```
ObjectName name = new ObjectName("DefaultDomain:name=DiscoveryClient");
DiscoveryClient dc = new DiscoveryClient();
dc.setMulticastGroup("224.224.224.224");
dc.setMulticastPort(7706);
server.registerMBean(dc, name);
dc.start();
```

Listing 5.24 Instantiating and initializing a Discovery Client component.

In a typical scenario, the application triggers a search operation by invoking the *findMBeanServers()* method on an active DiscoveryClient object. Using current settings, it sends a multicast request[247] and then blocks for the timeout period [143]. At the end of the timeout period, the method returns the responses received from remote discovery responders. In the described monitoring service[248], the discovery response only[249] carries the address of the JMX RMI connector, encapsulated as a binary UserData field. The *findMBeanServers()* function is called periodically by the MBSActiveDiscovery thread in JIMS SG, which is responsible for collecting the addresses of all monitoring agents running in the cluster. Next, the *getNodes()* function of SG returns the current list of recently responding monitoring agents (see Figure 4.10).

The listing also assumes that a reference to an already instantiated MBean server is present in the *server* variable.

[246] See above note.

[247] Theoretically, the scope of the discovery request can be adjusted by modification of the time-to-live parameter used by the multicast socket. Nevertheless, this dynamic discovery feature is not applicable in practice, due to the absence of multicast routing in real grids.

[248] In the standard JDMK application, DiscoveryResponse carries remote MBeanServer-related information and contains an entire list of all registered communicator MBeans (the connectors).

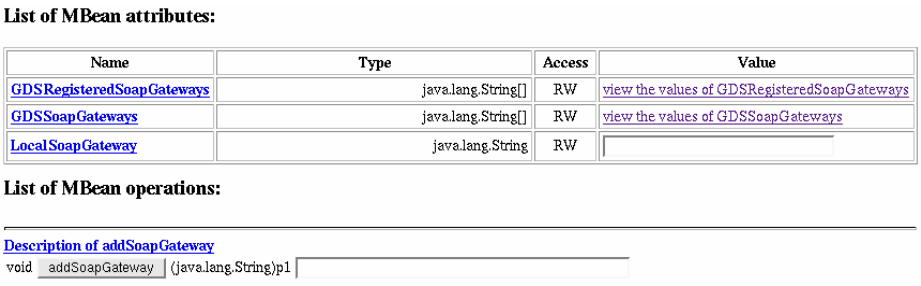
[249] In JDMK, the discovery response carries the full list of connectors.

5.4.2 Grid-Level Self-configuration

Grid-level self-configuration is the ability of the JIMS system to dynamically discover active clusters and to provide the user with a list of all available cluster access nodes. For this purpose, two GDS systems were developed – the first one with a static, arbitrarily chosen repository of global registry nodes, and the second one with a single migrating[250] global registry.

First prototype of GDS

The first version of GDS was implemented and deployed in CrossGrid. All implementation tasks were based on Java version 1.4, Java AXIS version 1.2 alpha and JMX – JSR 160. Just like many other JIMS modules, GDS was implemented as a JMX (MBean) running within an SG agent on a CE host. GDS implementation as an MBean results in high module reusability and does not narrow its functionality in any aspect. Having compiled and packaged it as a standard Java library (JAR), it becomes possible to not only deploy GDS in any MBeanServer requiring global discovery functionality, but in any Java application as well, since MBean is a standard Java class with a special interface exposing some attributes and methods. This interface can also be exposed by the HTML interface rendered by the JMX HTTP adaptor, as depicted in the Figures 5.11-5.13.



List of MBean attributes:

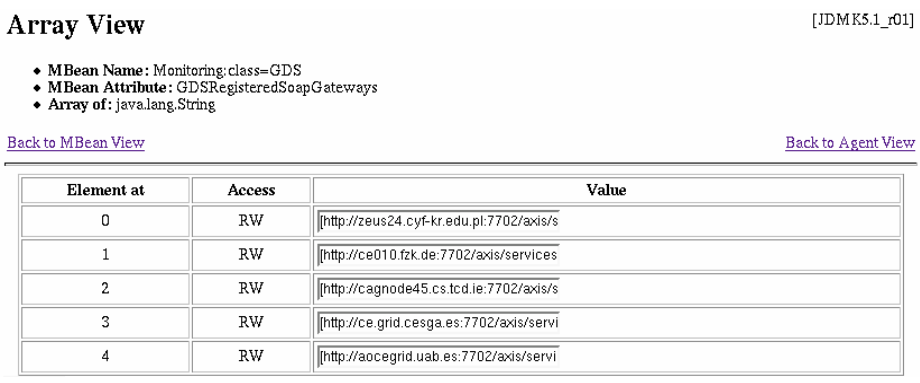
Name	Type	Access	Value
GDSRegisteredSoapGateways	java.lang.String[]	RW	view the values of GDSRegisteredSoapGateways
GDSSoapGateways	java.lang.String[]	RW	view the values of GDSSoapGateways
LocalSoapGateway	java.lang.String	RW	

List of MBean operations:

Description of addSoapGateway

void addSoapGateway (java.lang.String)p1

Figure 5.11 GDS MBean attributes and methods.



Array View [JDMK5.1_r01]

- **MBean Name:** Monitoring:class=GDS
- **MBean Attribute:** GDSRegisteredSoapGateways
- **Array of:** java.lang.String

Back to MBean View Back to Agent View

Element at	Access	Value
0	RW	http://zeus24.cyf-kr.edu.pl:7702/axis/s
1	RW	http://ce010.fzk.de:7702/axis/services
2	RW	http://cagnode45.cs.tcd.ie:7702/axis/s
3	RW	http://ce.grid.cesga.es:7702/axis/servi
4	RW	http://aocegrid.uab.es:7702/axis/servi

Figure 5.12 GDS SOAP Gateway registry and list of registered SOAP Gateway nodes.

[250] In case of failure, re-election of global registry is performed.

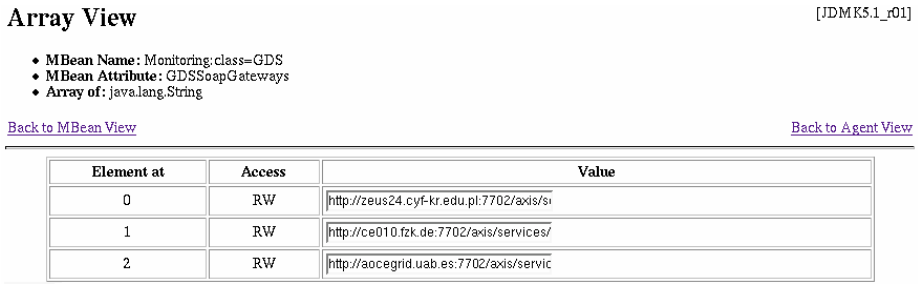


Figure 5.13 List of GDS registry nodes.

In order to dynamically load GDS module into the JIMS agent, the GDS JAR file should first be installed on a HTTP server. Subsequently, an m-let file should be prepared, containing the description of a given JAR file along with its contents: class, module name, etc. In the end, the *getMBeansFromURL()* method should be invoked with the m-let URL as a parameter. Three key components are exposed in this module: GDSRegisteredSoapGateways – a Java String array containing all SOAP gateways registered in GDS (it should contain a full list of all GDSs running the JIMS monitoring system, installed in the grid – see Figure 5.12); GDSSoapGateways – a Java String array containing all SOAP gateways performing the role of GDS nodes. Typically, it contains three SG entries (Figure 5.13) which maintain lists of the registered SGs. The LocalSoapGateway attribute is the URL of the local (from the agent's point of view) SG. This address is periodically registered with *n* (in this case 3) GDS nodes (Figure 5.11). For development purposes, the interval between registrations was set to 30 seconds, but just as the number of GDS nodes and their URLs, it is fully configurable through the GDS configuration file. The presented approach is described in [23].

Second Prototype of GDS

The second prototype of GDS was also implemented as JMX MBeans, registered in JIMS agents (MBeanServer) operating on CE hosts. Communication was based on remote access to the MBeanServer using the JMXRemote API (JMX connectors) and SOAP, available in MX4J [32]. Detailed descriptions of the GDS protocol, the GDS service and its applications can be found in [172, 20, 173, 70, and 98].

5.5 Uniform Infrastructure and Application Monitoring

Usage of JMX in the infrastructure monitoring system stems from relative ease in monitoring and management of resources represented as MBeans. However, since the introduction of Java language version 5.0, new possibilities have appeared for monitoring of Java applications. Because monitoring interfaces in Java 5.0 are designed for the JMX platform, it is possible to connect a Java application to JIMS without any modifications. The only two conditions that should be fulfilled are that the same version of JMX is used in JVM and

JIMS[251], and that the JVM JMX connector address of the running Java application is known. The mechanism of finding the JMX connector address will be presented later on in this section.

This section describes solutions for attaching Java 5.0 applications to the JIMS monitoring system. Possible scenarios of application monitoring for regular Java applications, as well as applications equipped with special discovery responder modules, are described. First, we will describe uniform worker node and application monitoring using JIMS, with monitored parameters available through JMX.

5.5.1 Java Applications and JVM Monitoring

JIMS makes it possible to monitor Java applications running in the grid using the same tools as for monitoring worker nodes. Connecting a Java application to JIMS enables not only monitoring of the application itself, but also monitoring of JVM, since in Java 5.0 JVM contains another built-in JMX monitoring agent[252].

Monitoring of Standard JVM MBeans

By using the JIMS client application we have the possibility to connect to a JIMS SOAP Gateway and list all MBeans available for monitoring of JVM. Once connected to JIMS, we can manage and monitor all available MBeans' attributes, operations and other parameters. JVM provides detailed information about threads and memory, especially in the context of the garbage collector, which is a crucial element of JVM. MBean details are shown in Table 5.2.

Using these components, JIMS can deliver information about many aspects of a running application, such as the type of compiler used for its compilation[253], garbage collector information[254], application memory statistics[255], operations[256], JMX notifications, memory pool statistics[257], operating system details[258], runtime system details[259], thread details[260] and

[251] Java 5.0 includes a reference implementation (RI) of JMX compatible with JSR-003 and JSR-160 (Java Specification Request no. 3 and 160) which is also available as a separate package for use with older versions of the Java platform: Sun Microsystems JMX RI v. 1.2.1 and Sun Microsystems RI of the JMX Remote API v. 1.0.1.

[252] The build-in JVM JMX monitoring agent is disabled by default but it can be enabled by JVM options, as shown in Listing 5.25.

[253] For example the HotSpot Client Compiler.

[254] Collectors types, collection count.

[255] Heap and non-heap memory usage: committed, init, max, used.

[256] Such as gc() – perform garbage collection.

[257] Young Generation: Eden and Survivor Space, Permanent Generation, Old (Tenured) Generation: committed, init, max, used [81].

operations[261], logger information[262], etc. All this information is available through a common JIMS WS interface.

Table 5.2 MBeans available in JVM version 5.0.

Java Interface	MBean Name	Monitored Component
CompilationMXBean	java.lang:type=Compilation	Compilation system
ClassLoadingMXBean	java.lang:type=ClassLoading	Class loading system
GarbageCollectorMXBean	java.lang:type=GarbageCollector, name=collectorName	Garbage collector
MemoryManagerMXBean	java.lang:type=MemoryManager, name=managerName	Memory pool
MemoryPoolMXBean	java.lang:type=MemoryPool, name=poolName	Memory
MemoryMXBean	java.lang:type=Memory	Memory system
OperatingSystemMXBean	java.lang:type=OperatingSystem	Underlying operating system
RuntimeMXBean	java.lang:type=Runtime	Runtime system
ThreadMXBean	java.lang:type=Threading	Thread system

In addition to JVM MBeans, the JMX management interface also enables monitoring of other specialized enterprise-level Java applications such as J2EE servers, elements of enterprise middleware, databases, etc. Many of these applications expose JMX management interfaces and were successfully tested by the author, including, among others, the JBoss Application Server [124], the Hibernate engine[263] [25] and the Sun Application Server 8.0 [151].

[258] Name, version, and architecture of operating system, number of processors, process CPU time, total and free physical memory size, committed virtual memory size, total and free swap space, max file descriptor count, open file descriptor count.

[259] Virtual machine name, version, vendor, specification name, specification version, specification vendor, input arguments, management specification version, library path, start time, uptime, boot class path, class path.

[260] Identifiers of all threads, daemon thread count, peak thread count, total started thread count, thread count, current thread user time, current thread CPU time.

[261] Find monitor deadlocked threads, reset peak thread count.

[262] Logger names and their operations (get and set logger level, get parent logger name).

[263] A high-performance object/relational persistence mapping and query service.

Further added value of using JIMS for monitoring of Java applications lies in the ability to discover such applications being executed someplace in the grid. JIMS SG functionality provides the possibility to use the interoperable SOAP protocol to access such applications. The management application can connect to a Java application started inside a cluster located behind a firewall and/or NAT, handling the private addressing of computational nodes on which the application is being executed.

5.5.2 Static Registration of Application in the JIMS Monitoring System

The first method of connecting the JIMS system to an already running Java application is to statically (administratively) register it with the JIMS SG. The MBeanServer of the registered application becomes available for monitoring by means of JIMS SG, exposed in a manner similar to other monitored worker nodes. Figure 5.14 presents information retrieved from the monitored application (in this case a new version of MVM 5.0), registered with the JIMS system.

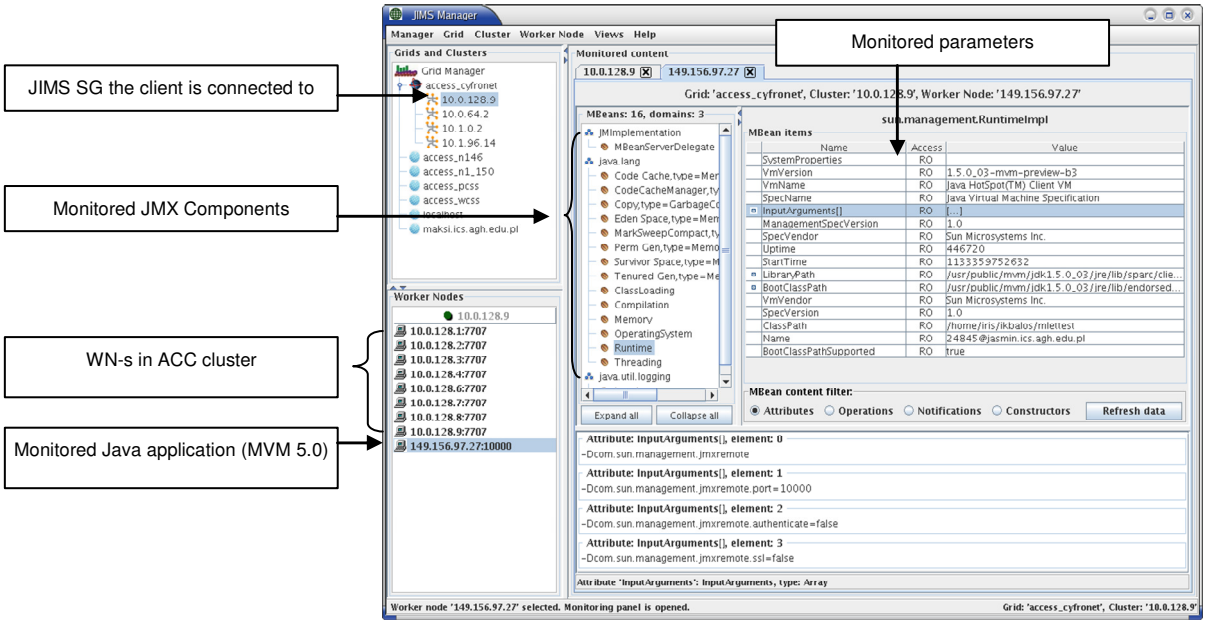


Figure 5.14 Uniform monitoring of Java applications and WNs with the JIMS GUI.

5.5.3 Dynamic Discovery of Java Applications in JIMS

Dynamic discovery of Java Applications in JIMS is a problem that has ben solved and practically proven in many aspects. Tests were performed on modified Java applications (with installed discovery responders) running in the grid. For these purposes the discovery responder can be created and started by the application developer using a specialized API. It can also be loaded and started by using Java configuration parameters (Java agent, dynamic class loading and pre-main function facility). Monitoring Java applications requires fulfilling two conditions: the application has to be running under JVM version 5.0 or higher and a JMX agent must be enabled in JVM for remote monitoring. The application does not need to be compiled using JSDK 1.5 unless it makes use of specific management functions of JVM

(internal agent). Moreover, the monitored application can be any standard Java application, agnostic of the fact that it is being monitored. Listing 5.25 shows the way a Java application should be started in order to enable its internal management and monitoring agent, assuming that JVM 5.0 is used and that SSL [72] encryption and authentication is switched off.

The application started in this way is available for further remote monitoring using its RMI connector server address in the form shown in Listing 5.26. Though it is physically possible to connect to a Java application running within JVM 5.0, locating it in the grid still poses a problem because the user does not know the IP address of the node on which the application is actually running.

```
java \
-Dcom.sun.management.jmxremote \
-Dcom.sun.management.jmxremote.port=10000 \
-Dcom.sun.management.jmxremote.authenticate=false \
-Dcom.sun.management.jmxremote.ssl=false \
-cp jims-discovery.jar:jjob.jar \
jjob.Hello
```

Listing 5.25 Running Java application with JMX monitoring agent enabled.

A simple solution to this issue is to check queues in the job management systems or to g-login [125] to worker nodes and find the application and its JMX address. Another way to locate an application started somewhere in the grid is to use the discovery techniques. In the course of a study of possible discovery services, two techniques were discussed: passive discovery (Java application registers itself in a central registry such as SG) and active discovery (requiring the application to respond to certain types of messages). The adopted solution relies on active discovery and was designed for easy integration with the existing monitoring infrastructure. Tests focused on an application with a standard JIMS discovery module, responding to the JIMS SOAP gateway in the same way as WNs do. The discovery scenario and client interaction with the monitored application are shown in Figure 5.15.

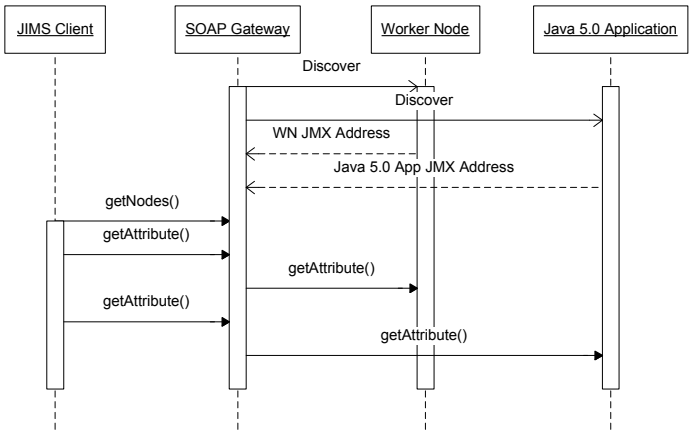


Figure 5.15 Accessing a Java 5.0 application.

Applications with the JMX agent and discovery service enabled are discovered by JIMS SG and made available for external clients using WS. Following successful discovery, each client can obtain a list of all registered MBeanServers and connect to the selected one by means of

the SOAP gateway. Such an approach provides a common interface for WNs and application attributes. The only difference is the JMX address form – Java 5.0 applications use JNDI for delivering interfaces of remote objects, while JIMS uses serialized objects as part of the JMX address (Listing 5.26)[264]. Addresses collected during the discovery process are generated by remote discovery responders, which simply return the JMX addresses of monitoring agent connectors[265].

```
service:jmx:rmi:///jndi/rmi://149.156.97.159:10000/jmxrmi
service:jmx:rmi://149.156.97.159:7707/stub/rO0ABXNyAC5qYX…gAAAeA==
```

Listing 5.26 Two types of JMX connector addresses (Java 5.0 application and JIMS agent, respectively).

Applying the discovery service to Java applications requires an additional thread to respond to multicast discovery messages, which entails modifications in the application's source code. These modifications include:

- creation of a DiscoverResponder object,
- preparation of a JMX address to pass to the DiscoveryResponder,
- creation of a new thread with the DiscoveryResponder,
- starting the DiscoveryResponder thread.

It is also possible to start another application, which would return the address of the monitored application. However, such a solution involves more overhead because running another JVM is far more resource-consuming than starting a simple, lightweight thread inside the current JVM. To verify the solution and to see whether JIMS is able to discover and monitor Java applications running in the grid, a simple JVM 5.0 application was prepared and executed as a standard job using Globus [76] middleware.

```
[    VirtualOrganisation = "cg";
     Executable = "jjob.sh";
     Arguments = "";
     StdOutput = "std.out";
     StdError = "std.err";
     OutputSandbox = {"std.out", "std.err"};
     InputSandbox = { "./jjob.sh", "./jjob.jar",
                      "./jre1.5.0_01.tgz", "./jims-discovery.jar"
     };
     Requirements = other.GlueCEUniqueID ==
     "zeus24.cyf-kr.edu.pl:2119/jobmanager-pbs-short";
]
```

Listing 5.27 Job description file for the Java test application.

[264] Update: in the most recent version (i.e. since 2.0.92), JIMS uses the same short, JNDI-based, form of the JMX RMI connector address. Such an address can be easily entered by administrators or constructed/configured automatically, with no need to send the entire string representing the serialized stub of the RMI connector server object.

[265] In JDMK, responders return addresses of all connectors. In JIMS, responders only return RMI connector addresses.

Listing 5.27 shows the contents of the jjob.jdl file used for submission of the Java test application. In addition to the batch script, jjob.sh and jjob.jar with the compiled application, two files were also submitted: jims-discovery.jar (containing the Discovery Responder module) and jre1.5.0_01.tgz, containing JVM version 5.0 with JMX monitoring capabilities. The experiment proved that the application can be properly discovered and that the JIMS SG correctly connects to the discovered application.

5.5.4 Runtime Instrumentation of Java Applications

Applications which are not originally accessible by the JIMS system can be equipped with the discovery responder without modifications in their compiled code. Thanks to the JVM feature called *java agent*, the user can prepare a non-modified Java application to be discovered dynamically by the JIMS monitoring system. This ability relies on the *premain()* method, which can be implemented in this the java agent and then called before the main method of the application class is invoked. This special implementation initializes the JIMS discovery mechanism, allowing JVM and the running application to be dynamically discovered.

5.5.5 Runtime Remote Instrumentation of Java Applications

The author foresees further possibilities for runtime instrumentation of Java applications, using the embedded MLet class and its dynamic loading facilities. Through the m-let service, the discovery responder code could be dynamically downloaded and then automatically discovered by JIMS[266]. However, this method of application instrumentation concerns only JVM 5.0 or above, with internal monitoring and the management JMX agent initially enabled. Moreover, this approach has not yet been tested.

5.6 Examples of Sensor Modules

This section presents examples of sensor modules developed for JIMS. Applicability of these modules ranges from benchmarking systems which need to know whether the tested system is really free and not used by jobs, through job management systems (GRMS) making use of grid engine monitoring modules, to data management systems which make use of the JIMS information layer for hard disk performance and network throughput monitoring[267].

5.6.1 Infrastructure Monitoring

All the instrumentation-level resources are implemented as MBeans (Managed Beans), where each monitored parameter (of a monitored resource) corresponds to an MBean attribute. Such an MBean is registered with the JMX MBeanServer, which makes it visible to management applications and exposes its management interface. In other words, management requests to

[266] The concept is based on the fact that the MLet class is available in JVM since Java version 5.0 and can be instantiated using JVM JMX management facilities.

[267] Actual bandwidth available for transport of user data.

an MBean are handled by the MBeanServer, which dispatches them to the appropriate MBeans [100].

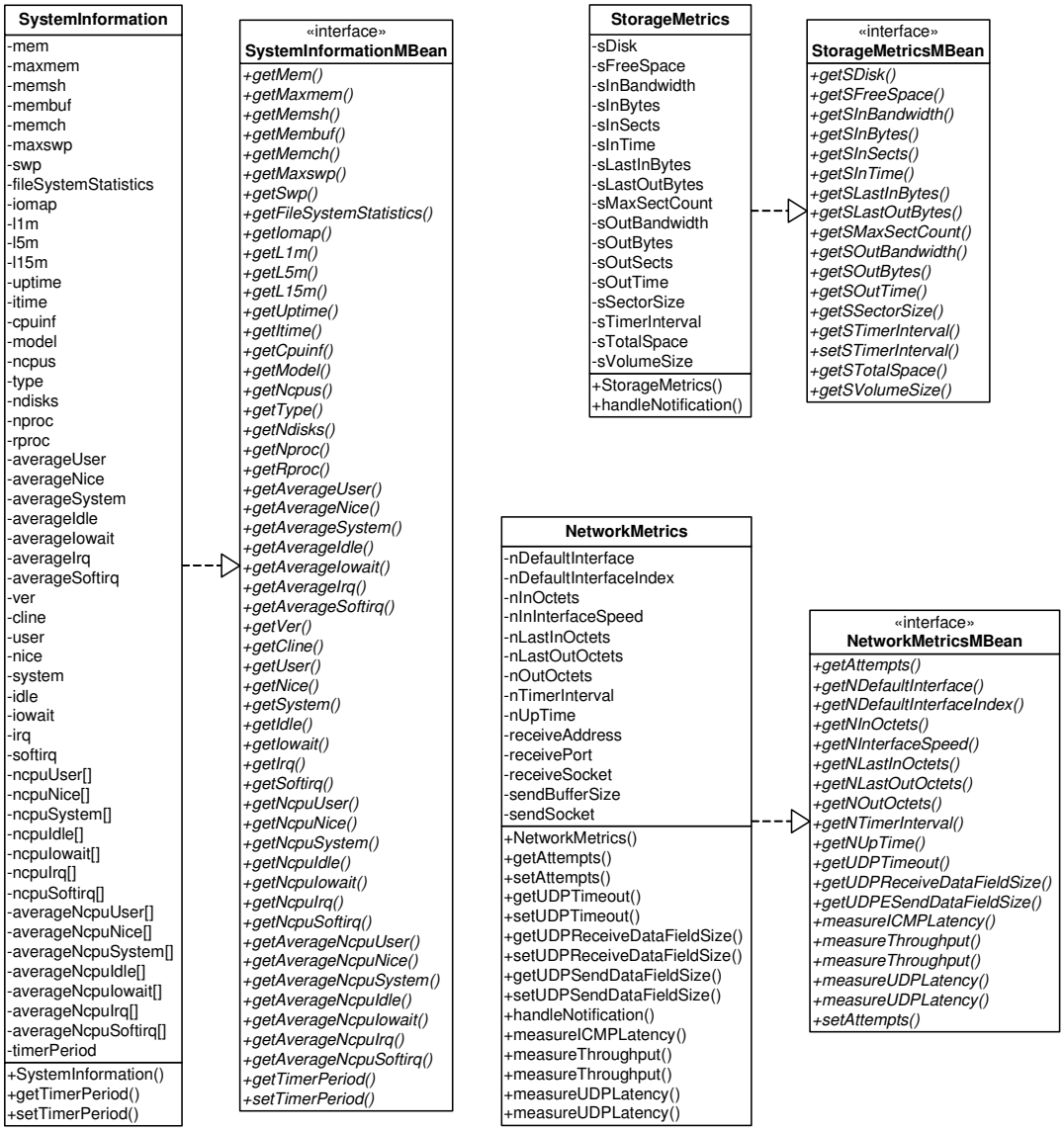


Figure 5.16 SystemInformation[268], StorageMetrics and NetworkMetrics MBeans.

An example of such a monitoring MBean is shown in Figure 5.16, which exposes (among others) information delivered by the operating system kernel interface and identifies some CPU statistics[269] per node or per CPU[270], memory usage statistics[271], number of running

[268] In addition to the attributes shown in the figure, there are also complex attributes, such as FileSystemStatistics which encapsulates information concerning disk partitioning, free space and usage.

[269] The amount of time[269] the CPU has spent performing different types of work: user (normal processes executing in user mode), nice (niced[269] processes executing in user mode), system (processes executing in kernel mode), idle (free CPU cycles), iowait (waiting for I/O to complete), irq (servicing interrupts), and softirq (servicing softirqs) [35].

processes, installed filesystem statistics and other additional information (version of installed kernel, version of operating system, etc.) A diagram of the SystemInformation component which exposes all these monitored parameters is shown in Figure 5.16[272].

5.6.2 Storage Monitoring

Another example of a monitoring MBean is the StorageMetrics MBean which instruments grid networked storage. Deployed on a Storage Node, the StorageMetrics MBean delivers information about hard disk performance and its current utilization. Performance information is obtained offline[273] by specially designed benchmark scripts and then delivered from the configuration file by means of attribute getter methods. Disk utilization information is periodically collected using the kernel API[274] and then presented in a similar way, as shown in Figure 5.16.

5.6.3 Network Performance Monitoring

Figure 5.16 also shows the class diagram of the NetworkMetrics MBean[275]. This MBean delivers information about network condition and provides two types of network-related information. First, some statistics are computed periodically based on the SNMP Proxy MBean described above. The most frequently used metrics cover incoming and outgoing packets transmitted through network interfaces over a given, preconfigured period. The module is also able to measure the latency and throughput between any pair of nodes inside or outside of the computational cluster. The methods can measure latency using UDP and ICMP protocols[276]. However, grid users should be aware that such throughput measurements are more of an estimation of the bandwidth available during the strictly specified period and can change along with network conditions[277].

---

[270] In Figure 5.16 the parameters which provide detailed per-CPU information are called getAverageNcpu{Idle,User,…}. They return average values of CPU time spent on different tasks during a given interval (configured dynamically by the timerPeriod parameter, typically 2 seconds). Per-CPU monitoring information is not commonly available in many monitoring systems.

[271] Total usable ram, free memory, buffers, in-memory cache for files read from the disk, swap cached, and many other parameters [35].

[272] More details can be found in appendix B, in Table B.8.1.

[273] In some cases the benchmark measuring disk performance can also be invoked during normal system operation. However, disk performance is a fairly constant parameter, and therefore this benchmark should not be invoked too frequently, especially as it induces high disk load and takes a long time to complete.

[274] Typically using the */proc* VFS filesystem in Linux/Unix systems.

[275] More details can be found in appendix B, in Table B.8.3.

[276] Measurements are performed by means of *measureICMPLatency()* and *measureUDPLatency()* methods, with variable sent/received packet lengtsh and a configurable number of measurements per method call.

[277] This problem is thoroughly discussed by authors of other monitoring tools such as IPerf and OWAMP [162,132].

5.6.4 SNMP Network Monitoring

The SNMP Proxy MBean provides instrumentation of devices equipped with SNMP agents. SNMP agents are an important part of the managed resources since SNMP is widely used to monitor network devices. Though JDMK provides tools for generating MBean interfaces from SNMP MIB descriptors, the presented monitoring service uses a specialized SNMP monitoring MBean (described in [131]) based on the WestHawk SNMP Java framework [11]. With this MBean, an SNMP agent can be represented and accessed through the MBean interface shown in Figure 5.17[278], which allows grid users to read any SNMP variables from the SNMP agent [100] by means of a uniform JIMS JMX interface.

5.6.5 Job Management System Monitoring

Job management system monitoring is required by grid job brokering systems in order to pick a proper (in terms of computational power and other available resources) queue for a newly submitted job. Following this assumption, the better the load of computational nodes is reflected in the monitoring service, the better decisions the brokering system can make. However, in addition to online statistics covering current computational node usage, grid engine-related information is also required – including the total number of computational nodes available, the number of nodes that are currently free, the names and parameters of configured queues in the queuing system, jobs running inside the cluster, their priorities, scheduler policies, etc. A sample monitoring component was developed and tested in relation to the JIMS monitoring service in the CLUSTERIX project, providing information about details of the available queuing systems such as PBS, Torque and SGE. The component was called gridEngineMonitoring and used a C API and Java bindings for the C libraries to access the grid engine monitoring parameters. Some of them, inaccessible through this API, are read using the **DRMAA** API [123][279] and calls to executable tools. A logical view of this component (the gridEngineMonitoring MBean) is shown in Figure 5.17.

[278] More details can be found in appendix B, in Table B.8.2.

[279] DRMAA is in fact a standardized set of command line tools, however in literature it is called an API.

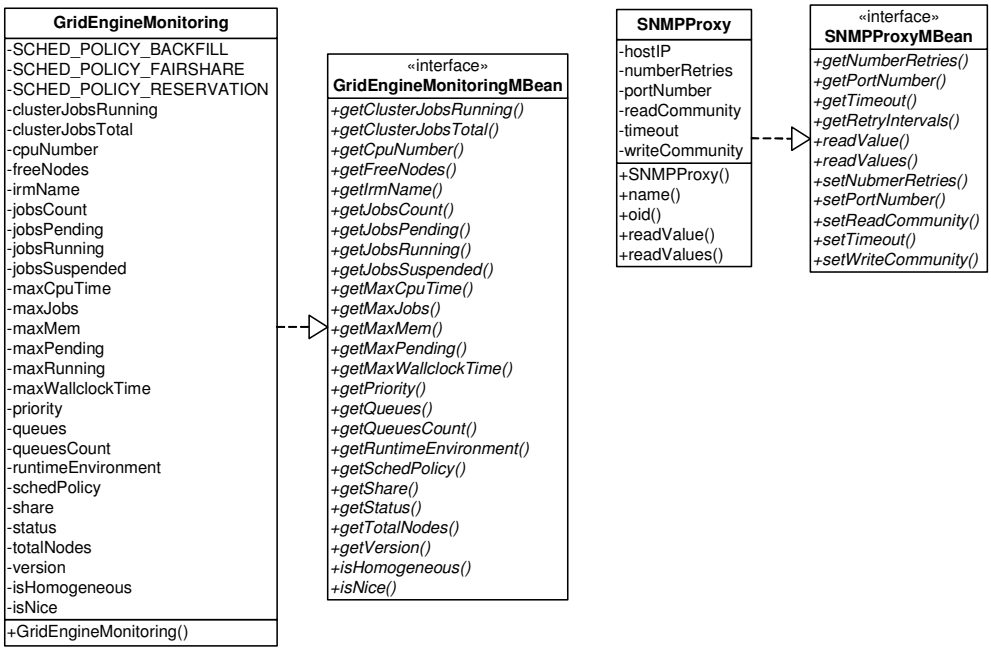


Figure 5.17 GridEngineMonitoring and SNMPProxy MBeans.

5.7 Security Concerns

The authentication mechanism and secure communication in the JIMS system can be achieved by using **JAAS** (Java Authentication and Authorization Service), **GSI** (Grid Security Infrastructure) or specialized security interceptors. The first mechanism, JAAS, is a standard security facility available for the Java platform, which separates security logic from other types of system functionality, aiming to support authentication and authorization in all Java applications. The second mode works by extending GSI in order to use **PKI** and Globus X.509 certificates to authenticate JIMS communication interfaces and their users. For example, by using the JMX SOAP connectors implemented in the MX4J framework, GSI certificates can be mapped onto **SSL/TLS**[280] public keys used by JIMS SOAP JMX connectors, which can then be configured to operate in a secure mode (see Listing 5.28).

```
service:jmx:soap+ssl://149.156.97.159:7702/jmxconnector
service:jmx:soap://149.156.97.159:7702/jmxconnector
```

Listing 5.28 MX4J SOAP connector addresses configured for operation with and withouth SSL support.

The third possibility is to use custom security interceptors, which can encrypt and decrypt the communication messages. An example of such a solution, implemented using the WASP container for WS is described in [177].

[280] TLS 1.0 is the proper name of the current security infrastructure using PKI and SSL version 3.0.

5.8 Chapter Summary

The chapter presented the possibilities of implementing JIMS using Java and JMX, and introduced some implementation details pertaining to the JIMS prototype and its application in real grids. Despite the name given to JIMS, the author does not consider JMX the only technology suitable for implementation of the concept of a self-configurable and adaptable monitoring system. Many other technologies facilitating distributed management and monitoring can be used for this purpose – including Jiro, .NET 2.0, CORBA 3.0 and CCM. Each has its own advantages, bottlenecks and scope of application. However, the author chose the JMX technology to prove that the JIMS concept is realizable and that it can be used to verify the architectural concepts presented in the thesis. It might be interesting to compare implementations of JIMS concepts using various technologies, though this is beyond the scope of this thesis and would require extensive research in its own right.

6 Verification of Results and System Evaluation

This chapter defines the testing methodology and describes non-functional, functional (concerning adaptability and self-configurability), and performance (covering access time and intrusiveness) tests of JIMS.

This chapter shows the results collected while operating JIMS in real grid systems (EU CrossGrid, CLUSTERIX and DSRG projects) and constitutes an evaluation and practical verification of the thesis. The concept and prototype design of the JIMS grid monitoring system were evaluated and verified according to the scheme shown in Table 6.1.

6.1 Tests in CrossGrid T&V Laboratory Environment

An early version of the proposed JIMS system was tested in CrossGrid according to its T&V (Test and Validation) procedure described in [79]. The procedure covered tests of grid components under development, according to their delivered functional specifications. The objectives of these tests covered conformance of middleware with its specification, identifying potential bottlenecks, verification of measurement accuracy, detecting software security issues and testing the middleware components in order to prevent wide deployment of components with critical problems.

The CrossGrid laboratory environment for JIMS tests was composed of a minimal set of computer systems required for building a grid infrastructure, i.e. a CE (Computing Element), a SE (Storage Element) and WNs (Worker Nodes). The first system acted as an interface between remote clients requesting job submission through grid middleware and a local farm using local scheduling software. The SE was a data storage server holding the input and output files used for computations, whereas the WNs are the real computational machines, typically high performance computers with specialized software, such as MPI and MPICH-G2, installed.

Table 6.1 JIMS testing methodology.

	Testbed	Type of Tests	Description
Laboratory Environment	Crossgrid T&V testbed	Non-functional aspects	Tests and validation, quality assurance, software installation and usability, security and networking
		Functionality	Unit tests of sensor modules (SystemInformation, NetworkMetrics, SNMPMirror)
			System tests (interaction with other grid components)
			Stress tests
	N1	N.-f. aspects	Adaptability to heterogeneous hardware (Sparc CPUs)
			Uniform monitoring of infrastructure and applications
		Functionality	Self-configurability and dynamic discovery of applications
Real Grid	CrossGrid development and production testbed	N.-f. aspects	Compatibility
			Performance tests (access time)
			Stress tests
		Functionality	Unit tests (SystemInformation, NetworkMetrics, SNMPMirror, StorageMetrics, GridEngineMonitoring)
			Self-configurability & dynamic discovery of applications
			System tests
	CLUSTERIX testbed	N.-f. aspects	Intrusiveness (CPU and RAM consumption)
		Functionality	Adaptability to heterogeneous hardware (32/64bit i686 CPUs) and network protocols (IPv4 and IPv6)
			Self-configurability at the cluster and grid level

Along with the T&V procedure, a special QA (Quality Assurance [36]) methodology was applied, which helped improve the quality of JIMS source code by periodic measurements of metrics called Quality Indicators (number of lines of code and comments, blank lines, average cyclomatic complexity number per functions and classes, number of classes and functions etc.) JIMS QIs can be found in [134].

Finally, JIMS successfully passed the T&V tests and was installed in the T&V testbed where it operated continuously without any supervision or user intervention. Two ways to retrieve monitoring data were tested: direct requests to monitoring agents using the web user interface (HTML adaptors) in working nodes and calls to the Java API from the user code by means of

the SG. Communcation via SG was tested using only one port, 7702, on the CE. For development and debugging reasons it was also useful to have port 7708 opened to the outside world from each WN, but it was not strictly required (this web interface was disabled in the following releases of JIMS). The JIMS T&V results[281] mostly concern grid administrative problems (such as software packaging, service permissions, etc.) and measurement accuracy (concerning active network measurements and statistics[282]). During the T&V procedure, some unit and system tests were also performed. Unit tests were conducted through the web user interface and calls from a Java program. Altogether, several one-hour tests were carried out using this second method. In each test, all parameters from SystemInformation and NetworkMetrics classes were measured at five-second intervals. The parameters of the SNMPMirror class could not be measured because of the lack of the proper configuration of the SNMP software in the target testbed. System tests were performed using the Performance Prediction Component (PPC) and aimed to test JIMS interoperability with other components using its SG facility.

6.2 Tests in the N1 Laboratory Environment

The N1 architecture is another networking and clustering environment, developed by Sun Microsystems, and formerly called "N1 Sun Fire Blade 1600 System". The main differences of the N1 platform in comparison to typical computer clusters are as follows:

- nodes are packed as compact hot-pluggable elements called *blades*,
- management of the system is performed by specialized software (Sun N1 System Manager) for configuration of the computational fabric used for installation, provisioning, monitoring, and management of Sun Fire x64 and **SPARC**[283]-based nodes. The second part of the N1 architecture, the Sun N1 Service Provisioning System, is also used for configuration of running applications.
- network connections are based on IEEE 802.1q (VLANs), which allows building separated clusters (called *farms*) of nodes, composed of any free nodes available in the entire system,

[281] Four JIMS T&V reports: JIMS v .1.2.3 Test Report (CG-4.4-REP-v1.1-USC001-JIMS_TestReport.pdf), JIMS v. 1.2.4 Test Report (CG-4.4-REP-v2.0-USC001-JIMS_TestReport.pdf), JIMS v. 1.4.20 Test Report: CG-4.4-REP-v3.0-USC001-JIMS_TestReport), JIMS v. 1.5.02 Test Report (CG-4.4-REP-v4.0-USC001-JIMS_TestReport), all available online at the location included in appendix C ([2]).

[282] Network throughput measurements are very complex due to the volatile nature of public network conditions. For more accurate tests of network statistics, many specifications and concepts were published. Among them, the most important ones are **OWAMP** [132] and BING (Bandwidth Ping). Many misunderstandings in network measurements also result from different definitions of such terms as available bandwidth and throughput.

[283] SPARC (Scalable Processor ARChitecture) is a type of **RISC** (Reduced Instruction Set Computer) CPU designed and used by Sun Microsystems.

- the base operating system used in N1 is Solaris 10, and it features container-based structure and virtualization technology[284],

- N1 offers sophisticated mechanisms for management of resources allocated to each instance of the operating system and project (group of applications called *tasks*), as well as single processes.

Tests of JIMS in the N1 environment were performed in order to check whether it is possible to monitor and manage Solaris 10 containers (tasks and projects) and its virtualized instances using JIMS. Initial results of these experiments, published in [178], show that JIMS can be successfully installed as a N1 application and can monitor and manage the allocated resources for each process running under Solaris 10. Based on the N1 infrastructure, tests of uniform monitoring of grid infrastructure and applications were also performed. A special Java class was prepared using the *premain* function, which loaded the dynamic responder for the JMX-enabled JVM. Such an application was then automatically discovered by JIMS and visualized using the JIMS GUI, exposing it as an additional monitored node (results similar to the one shown in Figure 5.14).

6.3 Verification of Results in Real Grid - CrossGrid Development and Production Testbed

CrossGrid (funded by the EU in the 2002-2005 period), successor of DataGrid, was a grid project oriented towards interactivity and compute- and data-intensive applications characterized by interaction with a person in the processing loop. Such applications require support from both application and infrastructure monitoring, enabling execution of experiments in near real time and with a high level of detail.

6.3.1 JIMS Compatibility with Other CrossGrid Components

The main components of CrossGrid are depicted in Figure 6.1. In CrossGrid, JIMS [17] was integrated with many other grid components such as gridBench (a tool for evaluating the performance of the grid [163]), **PPC** (Performance Prediction Component, [34]), SANTA-G [46], and the Post-Processing Component [118] for analysis of monitoring data, developed at **ICM**.

[284] I.e. any single hardware node can run many independent and separate instances of Solaris 10 or other operating systems.

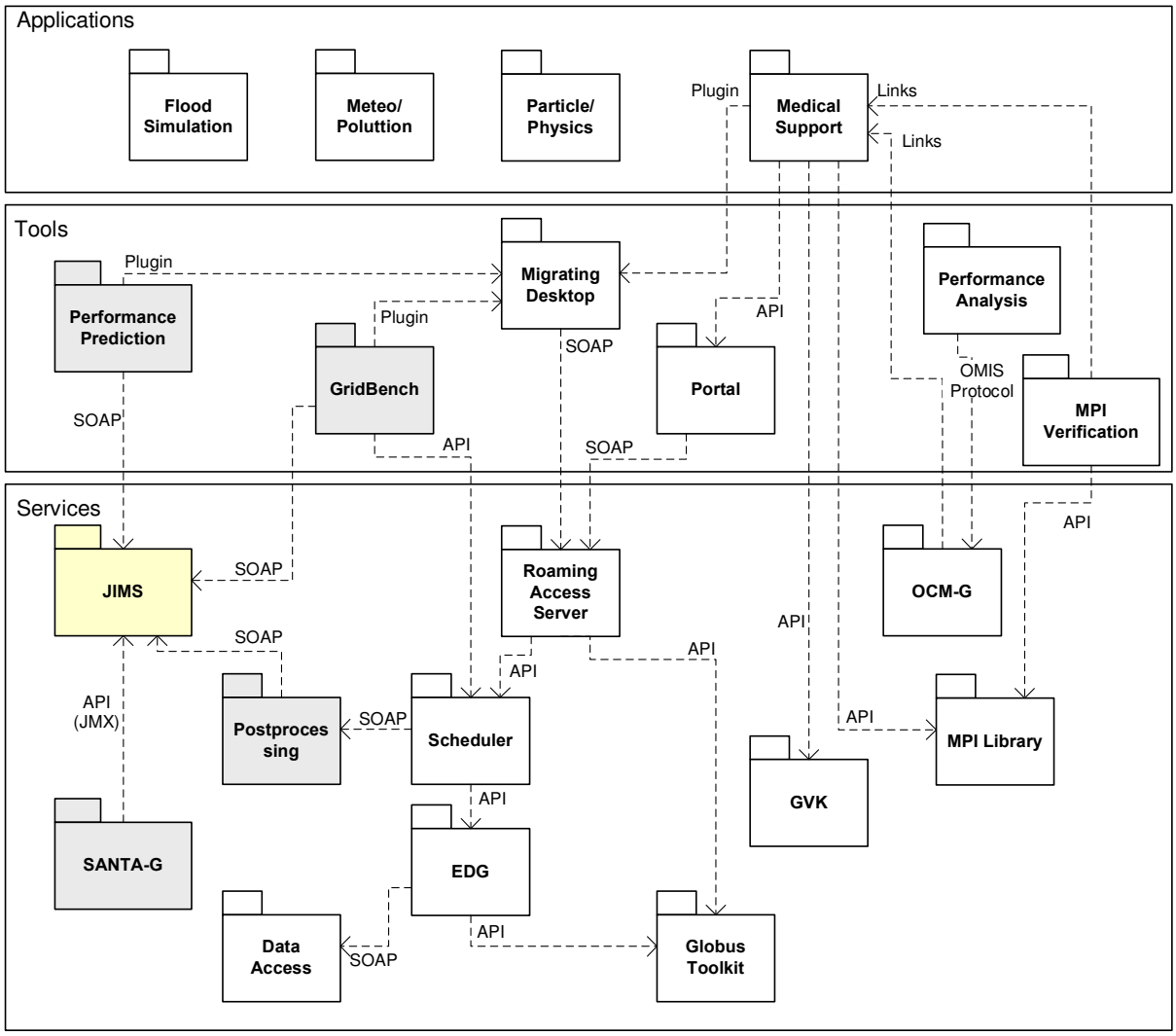


Figure 6.1 The three-tier architecture of CrossGrid and integration of JIMS with its components [126].

CrossGrid, with its two testbeds (one for development and one for production purposes)[285], was the first real grid environement where JIMS was applied and tested.

6.3.2 JIMS Access Time Measurements and Stress Tests

Performance tests conducted in real grid environments enabled us to evaluate the JIMS architecture and measure its overheads. Based on the grid characteristics described in section 3.1, the tests aimed to prove that the prototype grid monitoring service, built according to the guidelines proposed in this thesis and within the selected technology domain, can be used for monitoring of real grid infrastructure (i.e. assuming that JIMS is deployed in a real grid cluster and that many users can communicate with this service simultaneously).

[285] The first testbed was composed of clusters in Ireland, Greece, Germany, and Spain. The second one consisted of clusters in Poland (Kraków, Poznań, Warszawa), Spain, Greece, the Netherlands, Slovakia and Portugal.

Access time measurements were performed in order to evaluate whether SG strongly affected JIMS performance. The tests, already described in [18], show the overhead introduced by JIMS SOAP communication, how SG scales with increasing amounts of data and how it deals with a growing number of concurrent threads. Configuration of the testbed used for JIMS performance tests is presented in Table 6.2.

Table 6.2 Testbed used for JIMS performance (access time and SG overhead) tests.

Node	CPUs, [GHz]	RAM [MB]	Operating System (GNU/Linux)
CE (SG)	Dual Intel® Xeon™ 2.40	1024	2.4.18-686-smp
WN1..14	Dual Intel® Xeon™ 2.40	1024	2.4.18-686-smp
Client	Single Intel® Celeron® 1.7	640	2.4.18

The first scenario, depicted in Figure 6.2, shows access time for all monitored stations, sequentially polled and providing a fixed set of monitoring parameters[286]. Measured times (average of 100 performed tests) are on the order of 60ms (the client accesses JIMS using SOAP and JIMS SG) and 40ms (direct RMI communication with WNs), which indicates that overhead introduced by SG (blue bars) is around 50% higher than for direct communication via RMI, which seems to be acceptable. Moreover, recent installations of the JIMS system show that performance is not the only factor which should be taken into consideration when communicating with JIMS monitoring agents. Many cluster installations use private IP addressing and there is no direct access from the outside to the computational nodes, hence the use of a gateway is not a matter of performance, but becomes mandatory.

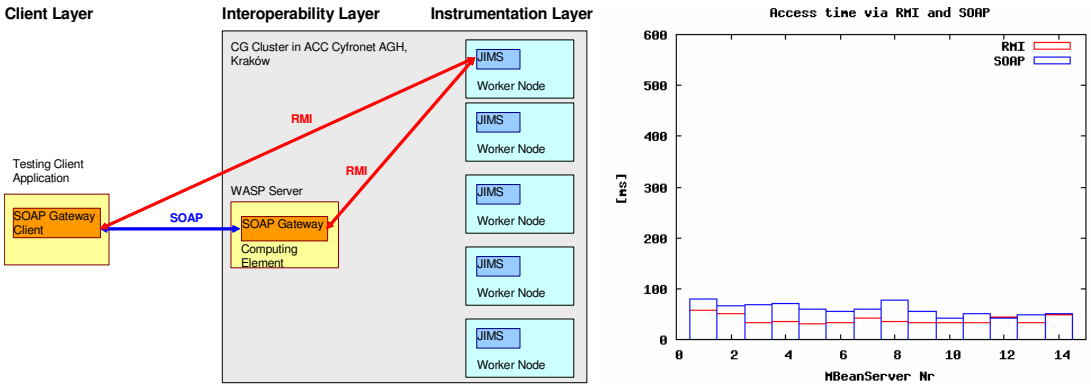


Figure 6.2 Scenario no. 1: sequential access time to each MBean server in the cluster.

In scenarios 2 and 3 (Figures 6.4 and 6.5 respectively) tests measured JIMS access time against the number of requests[287] and attributes[288] delivered in one request. Average access

[286] Three attributes of the SystemInformation MBean wre used: L1m, L5m and L15m (of type float).

[287] One attribute of type float was transferred per request.

[288] Attributes were taken from the SystemInformation MBean – see Table B.8.1 (the table shows parameters available in the current version of JIMS, which slightly differs from the one used in the test).

time from these two tests normalized for one request and one transferred attribute equals 29ms for SOAP and 14ms for direct RMI communication.

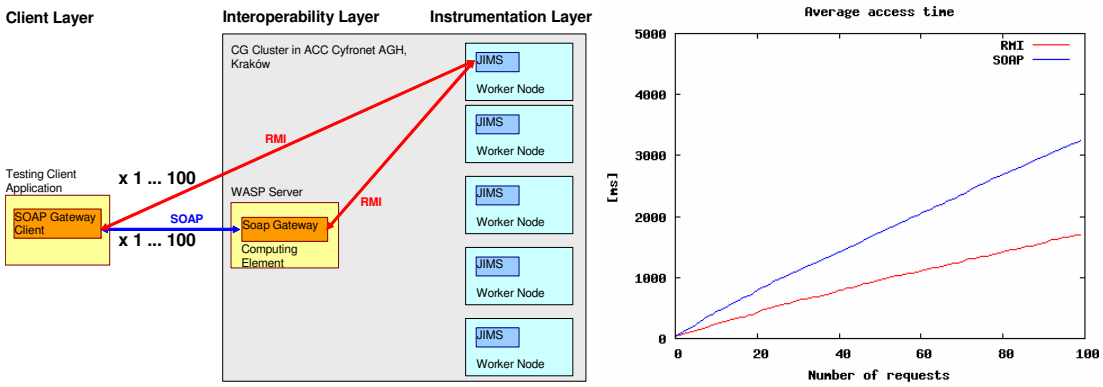


Figure 6.3 Scenario no. 2: repeating tests from 0 to 100 times with a fixed amount of data.

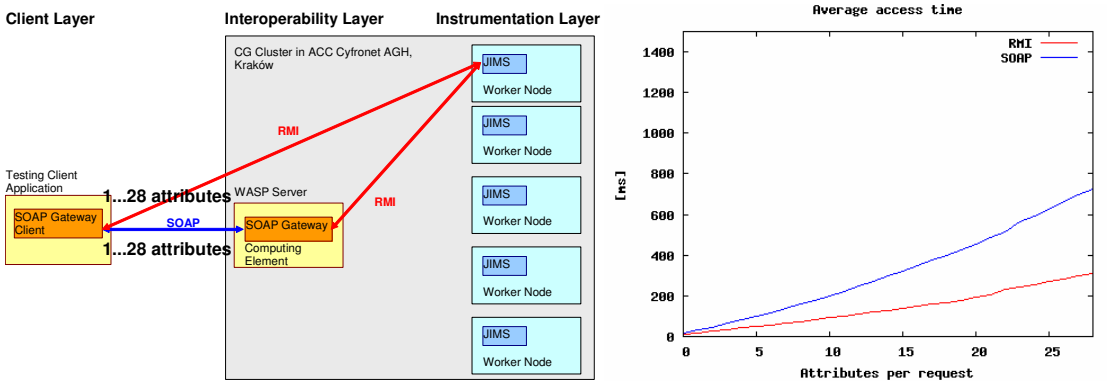


Figure 6.4 Scenario no. 3: number of attributes ranging from 1 to 28 per request.

More interesting tests were performed to gauge SG operation when many clients simultaneously request monitoring data[289] from different MBeanServers (scenario 4) or the same MBeanServer (the worst case, shown in scenario 5). Experimental results presented in Figure 6.5 and Figure 6.6 illustrate that SG does not introduce significant overhead (even when accessing the same monitored station), nor does it become a bottleneck.

In addition to performance tests, JIMS and SG also passed a long-term test. They have been successfully exploited in different ACC clusters since 2003 and many other computational institutions, both in Poland and throughout Europe. This proves that the presented solution, along with the early Sun JMX Reference Implementation and its Remote API[290], is reliable enough to be used for building grid monitoring services.

[289] The same set of attributes as the one specified in the first scenario.

[290] At that time (late 2003), the JMX was at an early stage of specification, and an "early access" version of the JMX **RI** (Reference Implementation) was available.

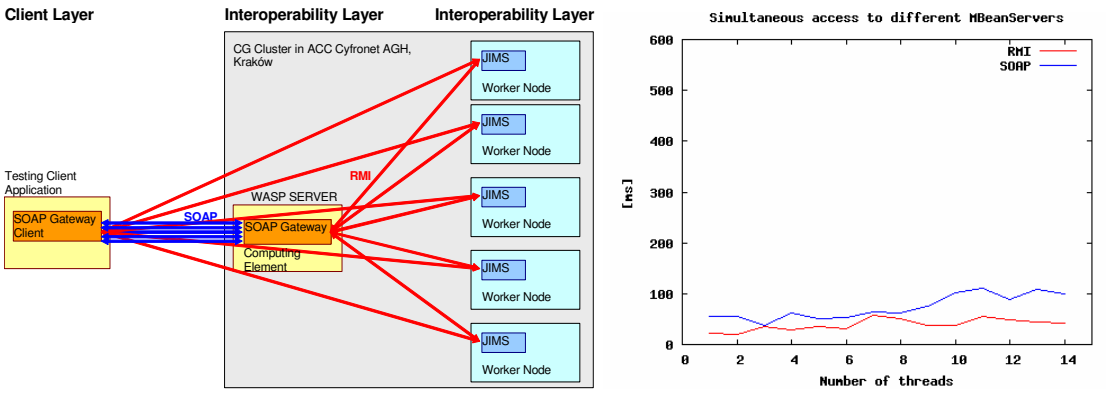


Figure 6.5 Scenario no. 4: simultaneous access to consecutive MBeanServers.

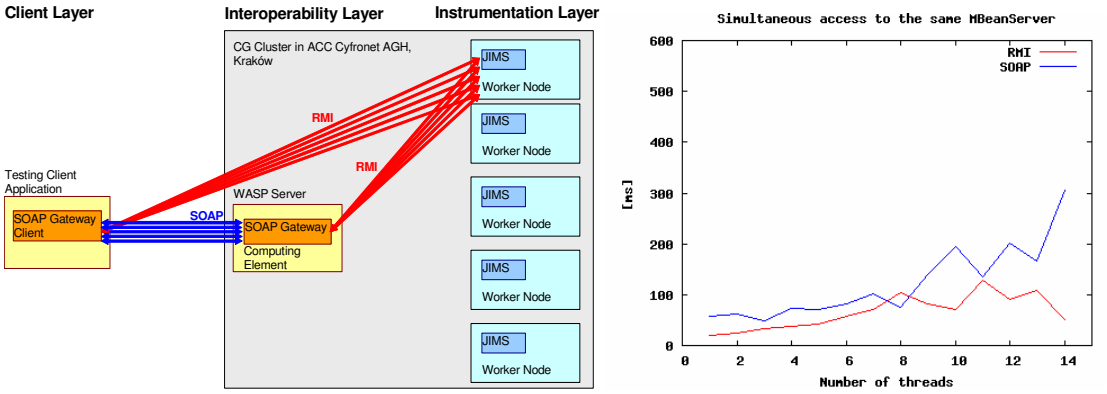


Figure 6.6 Scenario no. 5: simultaneous access to the same MBeanServer.

6.3.3 Sample JIMS Functional Test – Network Statistics

A sample of JIMS application for grid network statistics measurements is the measurement system created by M. Hardt from the CrossGrid Integration Team. A specially designed job with the JIMS CLI batch client was started periodically[291] on a given CE and it measured network latency to all remaining CEs. Such a system produced a constant map of site availability and showed the latency between each pair of CEs. Tables 6.1 and 6.2 show values of the latency[292] (in milliseconds), obtained by means of the JIMS NetworkMetrics module, using the UDP protocol and packets exchanged between CE nodes. The amber cells show the measurement results from one CE to the same CE, and contain lower values (due to local network calls) than the values marked in green. Red colouring denotes sites which could not be accessed to perform measurements *from* (red row), or were inaccessible *from* other sites (red column). Such visualization allows grid administrators to easily monitor the state of the

[291] The test ran automatically and was scheduled by a CrossGrid batch system which periodically executed the jims-cli application.

[292] Latency in a real grid environment is subject to frequent changes and depends on many factors, such as current grid computational power utilization, network utilization and many others (changes in layer III device routing tables, packets queuing algorithms, etc.)

whole grid and identifies possible bottlenecks (high latency between any pair of CEs) as well as site failures[293].

Table 6.3 JIMS UDP connectivity status page in the development testbed (latency in milliseconds)[294].

From\To	cagnode45 .cs .tcd .ie	grid14 .physics .auth .gr	ce010 .fzk .de	gtbcg01 .ifca .unican .es	aocegrid .uab .es
cagnode45.cs.tcd.ie	0.35	39.45	17.35	39.8	41.1
grid14.physics.auth.gr	34.8	1.1	20.25	49.15	51.85
ce010.fzk.de	17.35	25.55	0.35	25.85	27.1
gtbcg01.ifca.unican.es	39.8	49.25	25.75	0.35	14.95
aocegrid.uab.es	41.15	50.6	27.05	12.05	0.05

Table 6.4 JIMS UDP connectivity status page in the production testbed (latency in milliseconds)[295].

From\To	zeus24 .cyf-kr .edu .pl	ce .grid .cesga .es	cgce .ifca .org .es	cedar .crossgrid .man .poznan .pl	ce02 .lip .pt	cms .fuw .edu .pl	cgnode00 .di .uoa .gr	grid01 .physics .auth .gr	loki01 .ific .uv .es	xgrid .icm .edu .pl	ce2 .das2 .nikhef .nl	cluster .ui .sav .sk
zeus24.cyf-kr.edu.pl	0.25	52.2	55.55	9.85	56.4	-1.0	59.75	54.95	51.35	12.05	42.65	17.1
ce.grid.cesga.es	52.45	0.15	8.35	50.2	15.75	-1.0	38.5	40.45	8.4	51.35	24.25	27.65
cgce.ifca.org.es	55.6	8.35	0.3	46.75	19.0	-1.0	41.9	43.95	11.85	54.7	27.6	35.7
cedar.crossgrid.man.poznan.pl	9.65	43.3	46.65	0.25	46.0	-1.0	43.85	46.2	42.05	6.9	33.65	8.15
ce02.lip.pt	55.0	15.55	18.9	46.05	0.5	-1.0	44.3	46.4	14.4	53.8	20.75	33.7
cms.fuw.edu.pl	-1.0	-1.0	-1.0	-1.0	-1.0	1.6	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
cgnode00.di.uoa.gr	53.2	38.1	41.45	43.75	44.1	-1.0	0.25	5.3	37.3	51.8	24.5	28.1
grid01.physics.auth.gr	54.9	40.35	43.85	46.05	54.7	-1.0	6.1	0.35	39.15	50.05	22.9	26.4
loki01.ific.uv.es	50.95	7.95	11.4	42.05	14.35	-1.0	37.1	39.15	0.35	54.35	26.9	30.35
xgrid.icm.edu.pl	12.05	51.2	54.55	6.45	53.7	-1.0	51.75	49.95	53.85	1.0	41.65	16.6
ce2.das2.nikhef.nl	42.7	24.2	27.5	33.75	25.1	-1.0	24.65	23.0	26.85	42.1	0.15	10.9
cluster.ui.sav.sk	17.1	27.65	35.7	8.15	33.7	-1.0	28.1	26.4	30.35	16.6	10.9	0.3

[293] In Table 6.4, the `cms.fuw.edu.pl` site fails doubly: the testing application cannot start the JIMS job from this site and neither can it reach this site from any other location.

[294] Status captured in February 2005, using the test application developed by M. Hardt at FZK, Germany.

[295] See above note. A value of -1 means that there is no connectivity. i.e. there no possibility to send measurement packets from and to the affected node.

6.4 Verification of Results in Real Grid - CLUSTERIX Testbed

The CLUSTERIX project is a Polish National Linux Cluster [95], focused on building a production grid environment from local PC clusters with 32- and 64-bit architectures, operated by many independent institutions. The management software, supporting dynamically changing growth and configuration of hardware infrastructure, is based on the FLOSS concept (GNU/Linux) and extends the functionality of existing solutions with additional custom tools. The project is implemented by twelve KDM and MAN units, with ICIS performing the role of coordinator.

6.4.1 Intrusiveness - Monitoring Overhead and Resource Utilization

Intrusiveness tests show how monitoring data traffic affects monitored stations, their CPUs and access nodes where SG is typically deployed. Intrusiveness is measured as the amount of time spent by the CPU on performing tasks associated the operation of the monitoring system itself[296]. These tests were performed in the CLUSTERIX testbed with the hardware configuration presented in Table 6.5. Results are shown in Figure 6.7.

Table 6.5 Testbed used for cluster-level performance and self-configuration tests.

Node	CPUs, [GHz]	RAM [MB]	Operating System (GNU/Linux)
access	Dual Intel® Xeon™ 2.66	1024	2.6.8-2-686-smp
fw[297]	Dual Intel® Xeon™ 2.66	1024	2.6.8-2-686-smp
n1...n8	Dual Intel® IA-64 Itanium, 2 1.3	4096	2.6.8-1-mckinley-smp
Client	Dual-core Intel® Pentium® D, 3.2	1024	2.6.12-12mdksmp

Figure 6.7 shows mean CPU utilization time in USER mode, expressed in milliseconds, for the *access* node and the available[298] computational nodes (*n1...n8*). We can notice that CPU utilization induced by the JIMS system depends on the number of requests (for monitoring parameters) per second. This utilization is greatest on the access node, which is natural, considering the fact that each call to SG results in many subsequent calls to all WNs. The second factor, which makes access node utilization higher than in WNs, is the SG translation process from SOAP to RMI and vice versa. The observed results show that CPU utilization of all nodes (the access node and all WNs) increases linearly, and at around 65 requests per second is equal to 150‰ ([ms/s]) on the access node and a few permilles on each WN. The linear growth of CPU utilization shows that JIMS and SG scale against the growing number of requests, and that WN utilization is almost imperceptible for typical request rates in the order of less than 10 requests per second.

[296] It was difficult to test the JIMS CPU utilization due to some bugs in Linux monitoring tools (like *top*). For this reason, the JIMS CPU utilization was read directly from */proc* VFS by means of the JIMS monitoring system. This is why the measured utilization covers the overal CPU time spent in USER mode for all OS processes. However, the author ensured that no other CPU-heavy applications were running during tests.

[297] A firewall host, used in the next scenario (p. 6.4.1).

[298] The missing *n5* node was not available for tests due to a hardware failure.

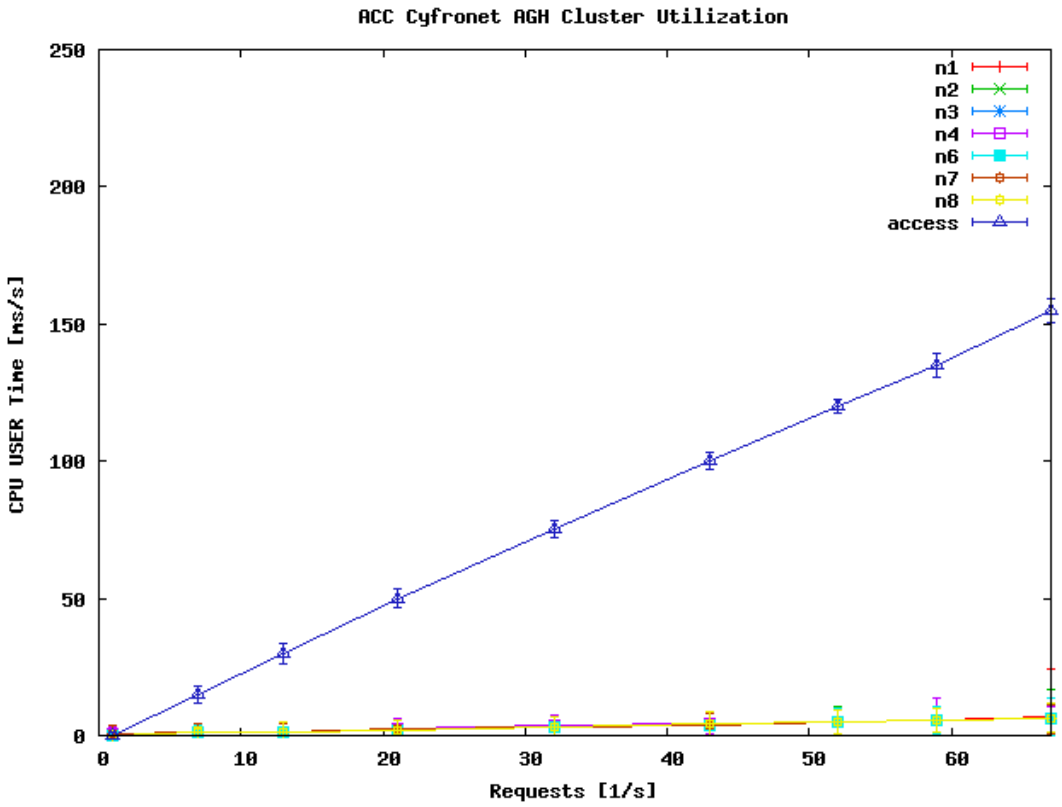


Figure 6.7 Cluster utilization against the number of requests per seconds[299].

The second metric, showing JIMS system intrusiveness, covers memory utilization. In the case of Java applications (including JIMS) memory utilization depends on activation of the internal monitoring agent (feature provided by JVM 5.0 and higher versions). By default, it is unnecessary to monitor the JVM itself, and thus internal JVM monitoring is disabled. Interested parties can enable this function by changing a configuration parameter in the JIMS startup script. On the access node, the measured memory usage by the monitoring agent was in the order of 32 MB, and on computational nodes – 96 MB[300]. Following activation of internal JVM monitoring, memory utilization increased to the level of 41 and 122 MB on the access note and computational nodes respectively[301]. Unfortunately, this is far more than for the competitive Ganglia monitoring framework written in C. For comparison, Ganglia only uses 2 * 4 MB of RAM on the access node (the *gmond* and *gmetad*), and only 4 MB on each

[299] The missing n6 computational node was not available due to a hardware failure and was not used throughout the tests.

[300] In Clusterix.

[301] A misunderstanding concerning the JIMS memory usage can result from incorrect interpretation of the memory used by JVM by such tools as the Linux top tool. It probably displays memory usage multiplied by the number of JVM threads (so-called light threads), which, in the case of kernel-level thread implementation (in practice all contemporary Linux and Unix systems) creates the impression, that – for example – 20 processes of for 30 MB each are started, while in fact there is only one process running, consuming the mentioned 30 MB. In order to prove this, we can check the amount of memory used before and after the JIMS system is started.

WN (only the *gmond* agent). This significant JIMS memory utilization is the main cost of the flexibility offered by the Java platform.

6.4.2 Functional Tests - JIMS Adaptability

The main goals of the projects focused on development of grid services using the new IA64 architecture (64-bit Intel Itanium architecture) and IPv6, with high security of the managed tasks. Infrastructure monitoring in CLUSTERIX was based on JIMS, which was ported from CrossGrid and enhanced to support new network protocols (IPv6), new Linux kernels (2.6 series), new hardware architectures (IA-64)[302] and many new software components for storage, network and queuing system monitoring.

The mixed IPv4/v6 network protocol environment and mixed 32- and 64-bit node hardware architectures[303] imposed further requirements for testing of the JIMS system, with both 32- and 64-bit JVMs, and required a level of monitoring system adaptability. Results of JIMS system evaluation in this testbed showed some drawbacks resulting from the use of early versions of JVM[304]. These drawbacks included higher memory usage than in ordinary grids (CrossGrid testbed was based on Intel Xeon 32-bit CPUs)[305]. However, the overall results of JIMS deployment in the CLUSTERIX testbed are very promising: following recompilation of native code for JIMS sensor modules and applying the adaptability mechanisms described earlier, it became possible to install and successfully run the JIMS monitoring service with an acceptable level of intrusiveness (see p. 6.4.1).

Cluster-Level Self-Configuration Tests

Cluster-level self-configuration tests aim to show how the multicast active discovery mechanism performs in a real network, assuming given discovery configuration parameters (heartbeat period and number of retries). Tests were performed in a cluster configured as shown in Figure 6.8.

[302] Clusterix is formed of about 800 IA64 CPUs (Dual Itanium 2).

[303] Access nodes featured 32-bit CPUs, while all computational nodes were equipped with dual 64-bit CPUs.

[304] For example, Sun Microsystems does not offer a JVM for Intel Itanium 2 64-bit architectures, which is why the author had to use another JVM implemented by BEA, called JRockit. Although the tests were successful, the results show high memory usage due to unoptimized native libraries used by JVM and heavy usage of 32-bit system emulation encountered, in the operating system libraries.

[305] This topic is discussed in sections 6.3.2 and 6.4.1.

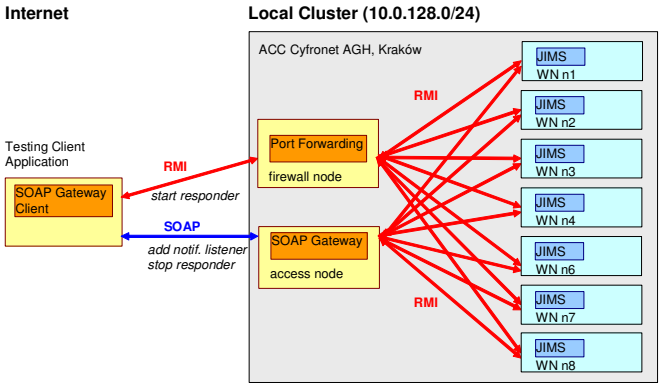


Figure 6.8 Scenario for cluster-level self-configuration tests.

Measurements covered t_r and t_a intervals, which express the time needed to react (sending the notification) on removed or added nodes. t_r and t_a were measured as shown in the sequence diagram in Figure 6.9. First, an application was started and then subscribed in the SG for notifications of the `WorkerNodeNotification` type. Second, the application stopped the DiscoveryResponder on node *x*, and then measured the time elapsed until a notification of the `WorkerNodeRemovedNotification` type was handled. Following this, the application entered a random wait period and then connected directly (through the firewall, due to the unreachability of the node via the SG) to node *x*, started the DiscoveryResponder module, and then measured the time elapsed until the second notification of the `WorkerNodeAddedNotification` type was handled. Usage of a firewall with port-forwarding is a consequence of the private addressing space inside the local cluster. Such addressing requires either gatewaying (in the case of a JIMS SOAP Gateway) or port forwarding, in order to connect to the monitoring system from external public networks.

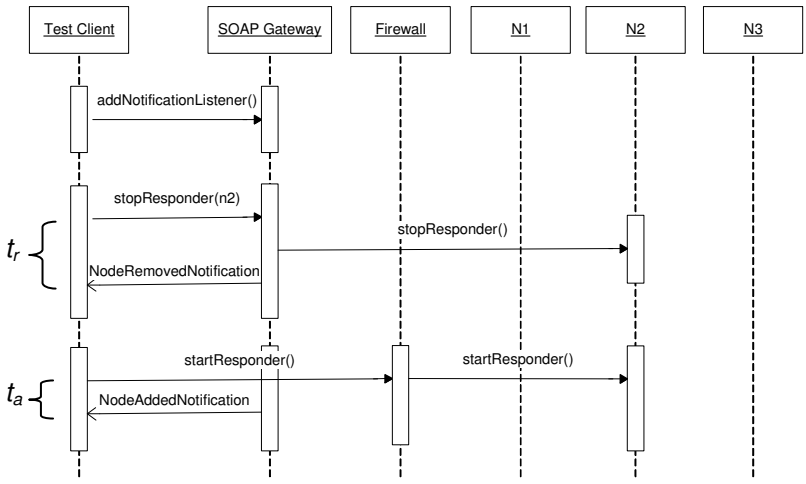


Figure 6.9 Sequence diagram of cluster-level self-configuration tests.

Measurement operations were performed numerous times for different nodes. Sample results for the value of maximum failed discovery attempts set to 3 and heartbeat time set to 10000 milliseconds are shown in Table 6.6.

Table 6.6 Results of cluster-level self-configuration tests for a given number of retries and heartbeat period.

Nr	Node IP	Retries	Heartbeat [ms]	Notification delay [s]	
				Node removed	Node added
1	10.0.128.1	3	10000	28.5	10.9
2	10.0.128.2			32.0	4.9
3	10.0.128.3			24.3	6.9
4	10.0.128.4			24.9	6.9
5	10.0.128.6			29.5	4.9
6	10.0.128.7			24.3	9.9
7	10.0.128.8			30.3	3.9
8	10.0.128.1			33.5	3.9
9	10.0.128.2			30.3	3.9
10	10.0.128.3			31.3	11.9

The configured parameters cause notifications of node removal (from the list of active nodes) to appear after time t_r (between 10 and 20 seconds), and notifications of new node discovery to appear after time t_a (between the 0 and 10 seconds). The computed average value for t_r equals 18.7 seconds, with minimum and maximum values equal to 13.2 and 22.9 seconds respectively. For t_a, the computed average value equals 8.6 seconds, with minimum and maximum values equal to 4.9 and 11.9 seconds respectively. Summarizing, the average values fit the proper ranges of allowed values, with some individual results exceeding the expected boundaries by about 2-3 seconds. Results, which show how the number of retries affects the notification time, are depicted in Figure 6.10.

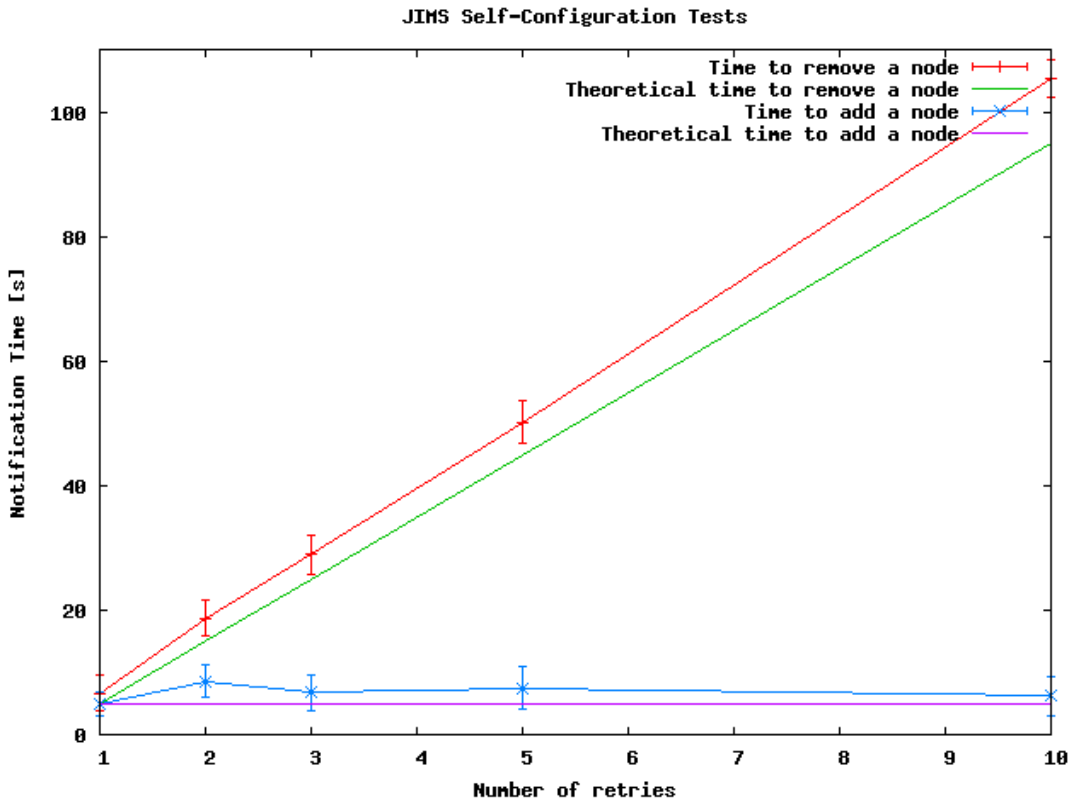


Figure 6.10 Notification time against number of retries.

Results plotted with blue and red lines show the time of notification for node addition and removal respectively. The theoretical lines for statistical[306] t_a and t_r can be defined as follows:

$$t_a^t = \frac{1}{2} t_{heartbeat} = 5[s],\ t_r^t = \left(n_{retries} - \frac{1}{2} \right) t_{heartbeat} = 10\left(n_{retries} - \frac{1}{2} \right)[s].$$

Time t_a^t is the average expected time of notification about the addition of a new node and it is equal to half the heartbeat time. Similarly, the time t_r^t of notification about node removal is, on average, equal to the heartbeat time multiplied by half the number of retries (see above).

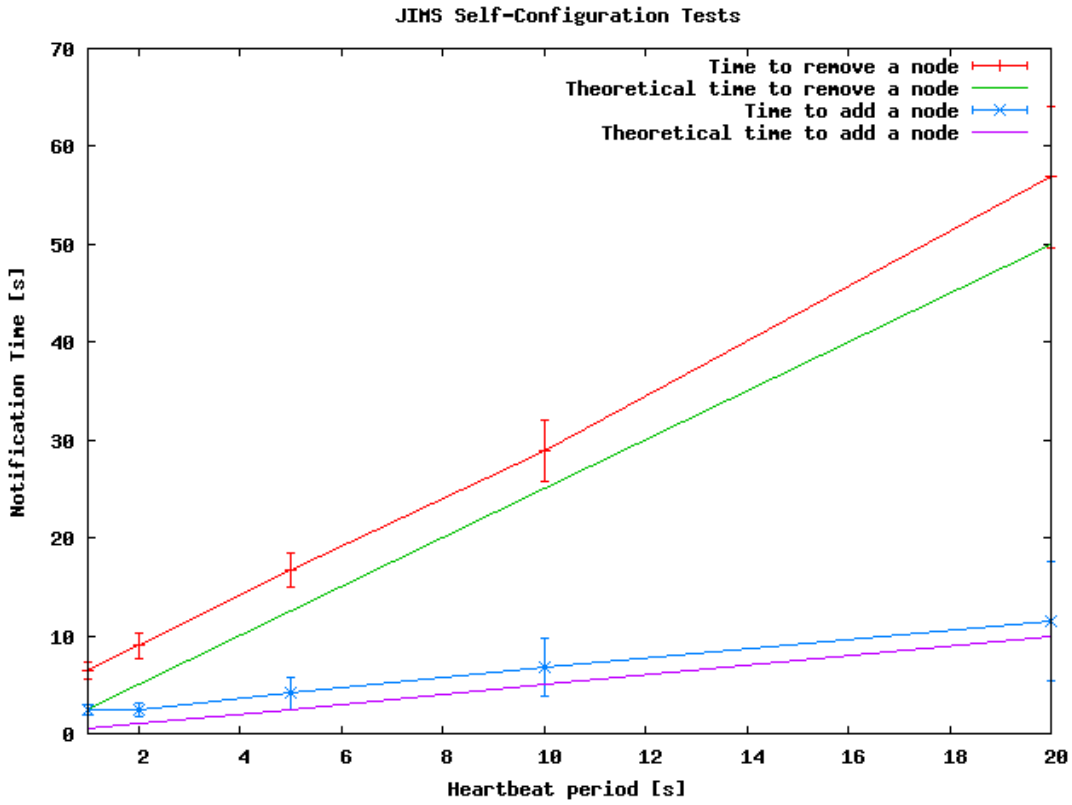


Figure 6.11 Notification time against heartbeat period.

The second diagram (Figure 6.11) shows notification time for removal and addition of computational nodes depending on the variable heartbeat time. In this case, the number of retries was set to 3 while the heartbeat time was subject to changes from 1 to 20 seconds[307]. Theoretical average times for notifications of node addition and removal against the heartbeat period, can be expressed as follows:

$$t_a^t = \frac{1}{2} t_{heartbeat}\,[\mathrm{s}],\ t_r^t = \left(n_{retries} - \frac{1}{2} \right) t_{heartbeat} = 2\frac{1}{2} t_{heartbeat}[s].$$

[306] Average from many measurements.

[307] Tests were performed with heartbeat periods equal to 1, 2, 5, 10 and 20 seconds.

The obtained practical results generally match the theoretical values, but are always several seconds longer, which can be attributed to additional delays introduced by network latency, delays when sending multiple notifications to many remote clients[308] and delays resulting from the operation of distributed agents in the cluster. The author is convinced that the obtained results are correct and sufficient for self-configuration purposes. In no case was notification omitted and no communication with a connected agent failed. These facts ensure that the cluster-level self-configuration mechanism of JIMS is operating correctly and can be successfully applied in real grid environments.

Grid-Level Self-Configuration Tests

Grid-level self-configuration mechanisms were already described in section 4.6.2. Two types of grid-level self-configuration systems can be distinguished. The first is based on a statically chosen and preconfigured (in all clusters) set of global registry nodes, used as a repository for periodic registration by each access node. The second system spans a statically preconfigured set of so-called Initial Sites, which are then used for initial integration of newly added clusters into the set of working access nodes. This second mechanism uses a dynamically elected single Global Registry, which can be changed in case of failure or unreachability. Due to the fact that the tests for this dissertation were performed using the latest version of JIMS, the presented results only concern only the most up-to-date, second prototype of GDS[309]. All these tests are based on GDS configuration parameters (see p. 5.4.2) shown in Table 6.7.

Table 6.7 Initial parameters for GDS System testing configuration.

Parameter	Value [s]	Description
Connection Timeout	10	Time to decide that the remote host is unreachable.
WaitTimeAYA	60	Time used by the *Are You Alive* procedure.
RefreshPeriodHeartbeat	60	Time used by clients that test the connection to GR.
RestartTime	300	Time to sleep in idle mode when no GR is elected or available.

All tests were performed using clusters with private the addressing scheme shown in Table 6.8.

[308] During tests, in addition to the application which actually tested the sent notifications, other JIMS client applications were also started (JIMS GUI – the JMX Manager and JIMS CLI – interactive command-line interpreter for JIMS). This was done for diagnostic purposes.

[309] Results from implementation and operation of the first prototype of the Global Discovery mechanism are presented in section 4.6.2, in the paragraph called “Static Global Discovery Service”. We present an example with three clusters set as Global Registries and five clusters (including the above mentioned three) registered with them.

Table 6.8 Clusters used in the tests.

Cluster Location	Institute	Cluster Address[310]	Access Node
Kraków	ACC[311]	10.0.128.0/24	access.cyf.clusterix.pl
Częstochowa	ICIS[312]	10.0.64.0/24	access.pcz.clusterix.pl
Poznań	PSNC[313]	10.1.0.0/24	clusterix-2.man.poznan.pl
Wrocław	WCNS[314]	10.1.96.0/24	access.wcss.clusterix.pl

Since GDS service operation depends only on access nodes available in each cluster, physical configuration concerns only these nodes and is shown in Table 6.9.

Table 6.9 Testbed used for grid-level self-configuration tests.

Node	CPUs [GHz]	RAM [MB]	Version of GNU/Linux OS
ACC	Dual Intel® Xeon,™ 2.66	1024	2.6.8-2-686-smp
ICIS	Dual Intel® Xeon,™ 2.40	2024	2.6.15.5-686-smp
PSNC	Quad Intel® Xeon,™ 2.40	2024	2.6.8.1-686-smp
WCNS	Single Intel® Xeon,™ 2.40	1024	2.6.8-2-686
UST	Dual-core Intel® Pentium® D, 3.2	1024	2.6.12-12-smp

All tests were performed using the client machine and JIMS client application deployed at UST. The client was connected to the JIMS agent with the GDS module deployed, installed on an access node at ACC, with a public IP address[315]. The GDS module was installed at ACC and acted as a HTTP repository for all other clusters. Thanks to such an architecture all clusters used the same version of JIMS components, with identical configuration parameters. During tests, all clusters used the GDS module with an attached properties file[316], with two configured Initial Sites: 10.0.64.2 (ICIS) and 10.1.96.1 (WCNS). The tests were split into two stages: observation of the GR election process and observation of the reaction of the GDS service to addition or removal of clusters from the grid.

GR Election, GR Failure and GR Re-election

The first part of the tests focused on observing the time for GR election following JIMS agent startup on selected access nodes, then simulating GR failure (stopping the agent which became the GR) and observing the process of GR re-election. Sample results obtained during

[310] Addresses of clusters use 24-bit netmasks, while access nodes use 8-bit netmasks in order to be reachable from other clusters within the 10.0.0.0/8 network.

[311] Academic Computer Centre CYFRONET of the University of Science and Technology in Kraków.

[312] Institute of Computer and Information Sciences, Częstochowa University of Technology.

[313] Poznań Supercomputing and Networking Centre.

[314] Wrocław Centre for Networking and Supercomputing, Wrocław University of Technology.

[315] In order to be publicly accessible, access nodes have both public and private IP addresses. In this case, the access node at ACC has (among others) two IP addresses (10.0.128.9 and 149.156.9.73).

[316] Included as a Java properties file in the JAR file.

thes two experiments performed in a three- and four-cluster environment are shown in Tables 6.9 and 6.10, respectively. We can observe the following events and the time[317] of their occurrence:

- GR election (4, 5[318]) after starting all access nodes (events no. 1-3, 1-4) and notification about completion of the list of active clusters (events no. 5-7, 6-9),
- simulated GR failure (events no. 8, 10),
- notification about GDS critical failure (failure of the GR), deregistration of all clusters (events no. 9-11, 11-14) and GR re-election (events no. 12, 15),
- notification about completion of the list of active clusters following election of a new GR (events no. 13-14, 16-18).

Table 6.10 GR election, GR failure and GR re-election (3 GDS nodes).

No.	Event	Notes	Location	Time	Delay[319]
1	JIMS agent started		ACC	00:00:00	
2	JIMS agent started		WCNS	00:00:20	
3	JIMS agent started		ICIS	00:00:35	
4	ICIS becomes GR		ICIS	00:01:42	00:01:42
5	Notification	ACC added	UST	00:05:25	00:05:25
6	Notification	ICIS added	UST	00:05:25	00:05:25
7	Notification	WCNS added	UST	00:06:25	00:06:25
8	JIMS agent stopped		ICIS	00:13:37	
9	Notification	ACC removed	UST	00:14:27	00:00:50
10	Notification	ICIS removed	UST	00:14:27	00:00:50
11	Notification	WCNS removed	UST	00:14:27	00:00:50
12	WCNS becomes GR		WCNS	00:15:34	00:01:57
13	Notification	ACC added	UST	00:19:32	00:05:55
14	Notification	WCNS added	UST	00:19:32	00:05:55

[317] The time of events, such as starting JIMS monitoring agents, was measured on each node separately and then normalized to the time of the JIMS client operating on the machine at UST. The normalization of these times was achieved thanks to deltas measured for all nodes and the machine at UST. The measurement was performed using the Linux *date* command executed simultaneously on two nodes, using the SSH protocol and passwordless authentication (with specially-configured RSA keys). Subsequently, timestamps were recomputed in relation to the UST clock. Such a normalized and relative time starts from 0 s on all nodes.

[318] In these and subsequent parentheses, the first number concerns Table 6.10 (3 clusters), while the second pertains to Table 6.11 (4 clusters).

[319] Relative time since the event causing the operation was carried out (starting the first agent or stopping the chosen agent).

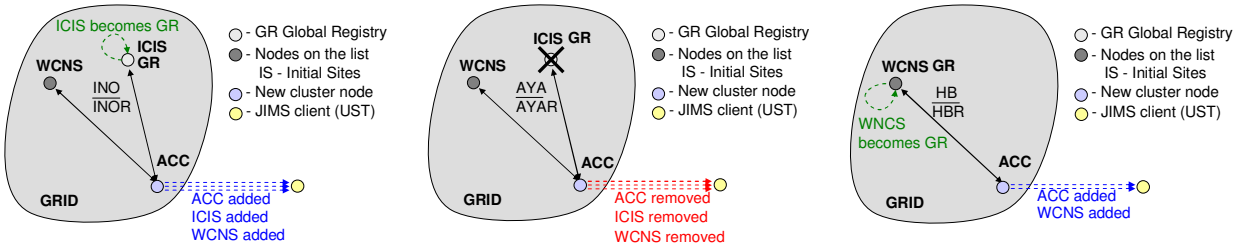
Table 6.11 GR election, GR failure and GR re-election (4 GDS nodes).

No.	Event	Notes	Location	Time	Delay[320]
1	JIMS agent started		ACC	00:00:00	
2	JIMS agent started		WCNS	00:03:14	
3	JIMS agent started		PCSS	00:03:55	
4	JIMS agent started		ICIS	00:05:13	
5	ACC becomes GR		ACC	00:06:25	00:06:25
6	Notification	ACC added	UST	00:06:04	00:06:04
7	Notification	ICIS added	UST	00:08:04	00:08:04
8	Notification	PCSS added	UST	00:10:04	00:10:04
9	Notification	WCNS added	UST	00:10:04	00:10:04
10	JIMS agent stopped		ACC	00:13:43	
11	Notification	ACC removed	ICIS[321]	00:14:24	00:00:41
12	Notification	ICIS removed	ICIS	00:14:24	00:00:41
13	Notification	PCSS removed	ICIS	00:14:24	00:00:41
14	Notification	WCNS removed	ICIS	00:14:24	00:00:41
15	ICIS becomes GR		ICIS	00:15:35	00:01:52
16	Notification	ICIS added	ICIS	00:15:35	00:01:52
17	Notification	PCSS added	ICIS	00:19:35	00:05:52
18	Notification	WCNS added	ICIS	00:19:35	00:05:52

Results show that the GDS service operates in accordance with the designed algorithm [172] and performs election once all JIMS agents are started (1 min. 42 sec.) However, a stable list of all active clusters is established near *RestartTime* (300 seconds), which in this case was equal to 6 min. 25 sec. (the final notification about addition of the WCSS node, event no. 7). Removal of a GR node resulted in notifications about disruption in the GDS registry 50 seconds after the GR agent was stopped. Following this event, a new GR was elected at WCNS (1 m. 57 s.) and notifications about the establishment of a new list of active clusters were sent after 5 m. 55 s. (once the GR agent at ICIS was stopped). In the case of four clusters, this process took longer to stabilize; however proper configuration was achieved after about ten minutes (events 6-9, Table 6.11). The system also correctly reactsed to GR failure and elected a new GR (in this case – at ICIS).

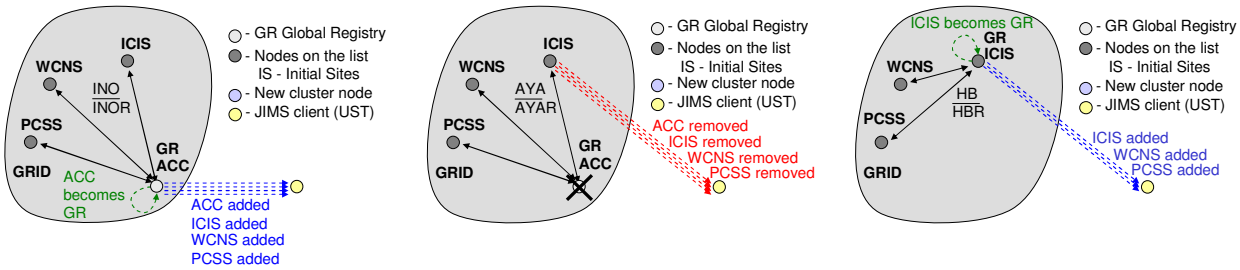
[320] See above note.

[321] In this case information was obtained from the log file of the JIMS agent deployed at ICIS due to unreachability of the JIMS agent switched off at ACC.



(a) election of GR (b) GR failure (c) GR re-election

Figure 6.12 Global Discovery Service test with 3 GDS nodes.



(a) election of GR (b) GR failure (c) GR re-election

Figure 6.13 Global Discovery Service test with 4 GDS nodes.

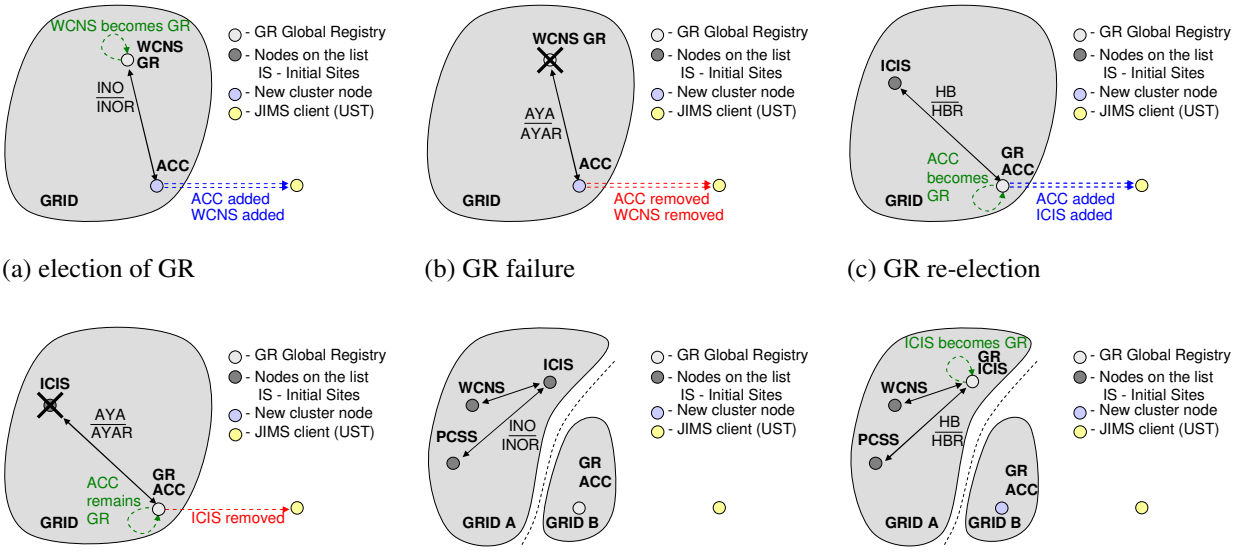
Addition and Removal of a Cluster and Network Partitioning Problem

The results of tests presented in Table 6.10 and Table 6.11 show a critical case when a failure of GR occurs and the GDS system has to elect a new GR. Since this is a rare case, we should rather measured the time for addition or removal of a cluster that is not a GR. For such a case, another experiment was performed, with the following events observed (measured times are shown in Table 6.12):

- GR election (event no. 3) and notification about completion of the list of active clusters (events no.4 and 5),
- simulated failure of a GR-cluster (event no. 6, critical failure), notification about this critical failure (GR failure, event no. 6), and deregistration of all clusters (events no. 7 and 8),
- addition of a new cluster at ICIS (event no. 9), election of a new GR at ACC (event no. 10) and notification about new clusters added (events no. 11 and 12),
- simulated failure of a normal cluster (event no. 13) and notification about this fact (event no. 14).

Table 6.12 Addition and removal of a cluster and the network partitioning problem.

No.	Event	Notes	Location	Time	Delay[322]
1	JIMS agent started		ACC	00:00:00	
2	JIMS agent started		WCNS	00:00:34	
3	WCNS becomes GR		WCNS	00:06:25	00:06:25
4	Notification	ACC added	UST	00:06:25	00:06:25
5	Notification	WCNS added as GR	UST	00:06:25	00:06:25
6	JIMS agent stopped		WCNS	00:11:04	
7	Notification	ACC removed	UST	00:11:27	00:00:23
8	Notification	WCNS removed	UST	00:11:27	00:00:23
9	JIMS agent started		ICIS	00:18:22	
10	ACC becomes GR		ACC	00:22:44	00:04:22
11	Notification	ACC added as GR	UST	00:22:44	00:04:22
12	Notification	ICIS added	UST	00:24:44	00:06:22
13	JIMS agent stopped		ICIS	00:25:55	
14	Notification	ICIS removed (ACC remains the GR)	UST	00:30:44	00:04:49



(a) election of GR (b) GR failure (c) GR re-election

(d) failure of non-GR node (e) network partitioning[323] (f) possible election of a second GR

Figure 6.14 Global Discovery Service test and the network partitioning problem.

Summarizing, the GDS system deployed in the JIMS monitoring service operates according to the requirements specified before, althouth its detailed analysis and comparison with theoretical assumptions is beyond the scope of this thesis. It should be noted that operation of the GDS highly depends on starting conditions. Moreover, statistical measurements of the

[322] See above note.

[323] Figure 6.14 (e) and (f) show possible scenario, assuming hypothetical network partitioning. The scenario in grid A following network partitioning is analogous to the one shown in Figure 6.12 (a).

operation of GDS are not the main goal of this dissertation[324]. In order to prove the stated thesis, the author instead prefers to show the possibility of deployment and application of different mechanisms for global discovery in the JIMS system.

6.5 Chapter Summary

This chapter describes the testing methodology and results of JIMS prototype application in a real grid environment. Results of verious tests were presented, ranging from tests in a laboratory environment, to tests in real grids, covering many aspects such as performance, intrusiveness and self-configurability.

The presented architecture and experiments prove that the idea of exposing the functionality of a grid infrastructure monitoring system through WS can be successfully applied in real grids and it provides the required level of interoperability. The presented solutions follow the WS-RF and WS-DM paradigms[325] and allow building a grid monitoring service, which is extensible enough to be used for monitoring highly-differentiated grid environments, ranging from typical GT-based grid installations and ending with N1 systems, with Solaris 10 containers and virtualization technologies, both highly exploited over the recent years. Tests confirm that WS do not degrade system functionality or introduce visible performance bottlenecks. They also prove that JIMS scales along with growing numbers of requests and responds quickly to incoming requests. The system can be successfully used in clusters with highly dynamic configurations of computational nodes and clusters.

Results of JIMS intrusiveness tests show that the most affected element of grid infrastructure is the access node (with SG installed). JIMS does not affect computational nodes in terms of CPU utilization. Unfortunately, a side effect of applying the Java-based JIMS implementation is the noticeable memory usage.

Adaptability of JIMS was proven by deployment of JIMS monitoring agents in a heterogeneous environment – in this case simultaneously on 32-bit (CE) and 64-bit (WN-s) nodes. In this case, proper monitoring modules were automatically loaded and JIMS monitoring agents were discovered by SG without any obstacles.

Tests of JIMS self-configuration prove that JIMS can handle dynamic addition and removal of computational nodes and can also be used as a global registry to hold lists of all active clusters. Cluster-level self-configuration test results were compared to theoretical expected values and are sufficient for proper operation of the whole monitoring service. The investigated mechanism gives administrators freedom to customize parameters in order to

[324] The author's view is based on the fact that many similar protocols, based on election and fault-tolerant mechanisms, implemented for instance in routing protocols (such as **OSPF** or **EIGRP)**, are typically described in terms of algorithm and main timers, and rather rarely in terms of time performance statistics. In the case of these algorithms, logical correctness and reliability are arguably more important than performance.

[325] JIMS reach functionality such as WS notifications, WS fault handling, and WS addressing, covers many aspects of WS-RF, WS-DM and WS-Management specifications.

achieve the required dynamic characteristics. Grid-level self-configuration was tested against proper functionality. Due to the complication of the algorithm, only functional tests were performed, showing that the mechanism deployed in the JIMS system works correctly and informs interested parties about addition or removal of clusters as well as about failures of the Global Registry itself.

Finally, tests show that the author's choice to base the proposed monitoring system implementation on JMX was correct, leading to a fully operable grid monitoring system with all the expected features of adaptability, self-configurability and uniform approach to infrastructure and application monitoring.

7 Conclusions

This chapter summarizes the dissertation, verifies the thesis statement, presents advantages, limitations and scope of applicability of the presented solution and, finally, discusses directions of future work.

This short chapter is a summary of the dissertation and it verifies the conformance of the thesis stated at the beginning with the results obtained during the study.

7.1 Thesis Verification and Confirmation

The author of the dissertation puts forth the thesis that there exists a component-based and coherent approach to resource representation, which allows building a self-configurable and scalable service for monitoring of infrastructure and applications in a grid environment, equipped with elements of adaptability. This adaptability results from the flexible structure of the system, capable of altering its functionality during runtime, without need for shutting down, reconfiguring and restarting the system. This was accomplished thanks to splitting the functionality of the system into smaller, independent elements, called components, which are transferred over the network from a central repository to particular nodes, and then deployed, depending on the current monitoring requirements. The aspect of monitoring system functional adaptation encompasses a wide spectrum of grid environment properties, including the type of hardware platform, types of network protocols, operating system kernel versions, queuing batch systems installed, etc. Moreover, mechanisms were described for discovery, self-configurability and uniform access to monitored nodes and applications. Basing on these facts, the author is convinced that the goal of the disseratation was achieved, i.e. it was confirmed that a system, build according to assumptions included in the thesis (involving a component-based, coherent and hierarchical approach to resource representation), can possess all the desired properties, such as self-configurability, scalability, and adaptability, needed for uniform monitoring of grid infrastructure and applications.

7.2 Introduced Novel Concepts

The system introduces a new approach to monitoring and management system design, which benefits from the powerful component-based, connector-oriented architecture, and which, from the client's point of view, provides excellent location transparency of the managed resources, allowing dynamic discovery of monitored resources, self-configurability and adaptability.

The JIMS concept is novel in two respects. First, it deals with the domain of grid monitoring systems, featuring elements of adaptability and self-configuration. The proposed JIMS system is built on top of manageable components and, unlike early research work on grid monitoring, which was restricted to the underlying operating systems and hardware platforms and typically oriented on object and procedural approaches to resource representation, shows that modern operating system abstraction layers can play a major role in providing the user with high flexibility in obtaining grid monitoring information. The proposed scalable, grid-oriented, component architecture allows automatic software installation and upgrading without system restarts, due to centralizing the module repository. Furthermore, it is grid middleware-independent and offers a far higher level of dynamism, self-configurability and adaptability than other frameworks such as MDS or GT/IS.

Second, the JIMS concept deals with the application of component and connector architectures in the domain of grid monitoring. As a case study, for the implementation of the system to possess all the expected features in terms of described requirements, the Java platform was chosen, along with its JMX extensions. In-depth JIMS prototype analysis and its practical verification in real grids shows that this technology was appropriate, resulting in a system with the expected properties. Thus far, JMX has not played an important role in grids, and its applicability was more focused on the enterprise application market (EIS-class systems). Presented test results prove that usage of the Java platform does not affect the underlying network and computational resources. The author's decision to use JMX is a result of practical system verification, encompassing a full development cycle, staring with a concept phase, then proceeding through design and implementation, and, finally, ending with practical evaluation of the prototype application in a real grid environment, which constitutes the author's key contribution to this area. The performed thesis verification proves that the JIMS concept can be practically applied to systems where dynamic resource discovery, self-configuration and adaptability are considered priorities. All these new properties of the grid monitoring service form the main result of this dissertation and the practical work carried out.

7.3 Critical Reflection on Presented Thesis

The JIMS monitoring system was developed in place of existing systems for infrastructure monitoring, previously available in the DataGrid project. Many other systems, such as MDS2, EDG NMA or MapCenter, provide similar functionality, but their collaboration is highly

dependent on the operating system used[326] and requires a homogeneous grid. JIMS delivers better flexibility and was successfully deployed on machines running different versions of Linux and Unix systems.

For example, MDS2 strictly follows the GMA architecture and is typically only installed on CEs and SEs. It uses LDAP[327] as a directory service and only provides general cluster information such as the number of all processors in a cluster, the number of jobs waiting in the queue, the number of free WNs or the configuration of the cluster itself. The static character of monitoring data obtained by means of MDS2 results from the fact that it reflects configuration parameters entered by the cluster administrator or lists of detected cluster capabilities, services and protocols, which do not change very often. The MDS2 LDAP database is also better suited for read-only operations.

On the other hand, JIMS is designed for more detailed monitoring and is installed on all WNs. Its auto-configuration mechanisms allow automatic discovery of WNs and clusters. JIMS provides its own instrumentation layer, which can be extended with regular applications equipped with the JMX interface. The access time to monitored data is on the order of several milliseconds and the system also provides low intrusiveness, which increases slowly and linearly with the growing number of monitoring requests, and a scalable architecture which proves that it can be used in an environment where many users want to obtain monitoring data simultaneously. The universal JIMS interface can be easily extended to support future grid resources thanks to the introspection mechanism, also used by WS-based clients, written in C, C++, Java, and Perl. Tools offered by JIMS allow monitoring of many clusters at the same time and enable visual monitoring of many grid aspects[328]. JIMS follows the concept of SOA specifications: OGSI/OGSA, WS-RF and WSDM, and offers reach functionality (i.e. advanced WS addressing and WS notifications) which can be found in the newest WS 2.0 specification. JIMS also has its own instrumentation layer (OS information, SNMP, DRMAA, network active measurements, hard disk benchmarks, etc.) Its lightweight architecture, based on connection pooling, does not affect the monitored system when it is not active and when there are no incoming requests. The JIMS implementation based on Java and JMX is an example of a portable system, which is at the same time open and freely available (in terms of FLOSS software such as JMX, AXIS, Net SNMP, without any dependencies on proprietary solutions). Finally, JIMS should be perceived rather as a framework, which can be refined and enhanced according to user needs.

Nonetheless, the flexibility of the JIMS architecture comes at a cost. First, the memory usage is noticeable in systems with small amounts of physical memory. Second, the usage of

[326] In CrossGrid, the entire testbed was designed strictly for Red Hat Linux v. 7.2-7.3 and a chosen set of common programming libraries for C, C++, and Java.

[327] MDSv2 is LDAP-based, but the future implementations of MDS (v3) are already based on the Java platform. The latest version, MDSv4, uses the WS-RF framework.

[328] Graphical charts can be dynamically composed of any parameters available in many different clusters.

reflection mechanisms results in larger computational power consumption in comparison to systems with statically defined interfaces. Furthermore, the system does not address standard information models such as CIM. Standardized connectors (JSR 262 [155], conforming to the WS-DM specifications to a certain extent), should be used instead of current MX4J SOAP connectors[329]. JIMS could take better advantage of other monitoring systems in terms of reusability of sensor modules and since it lacks a persistent store, it should be integrated with another specialized system for collection of monitoring data [122]. If there is a need to adapt JIMS to GT-based systems, JIMS can be easily equipped with a unified security solution based on GSI. In order to further minimize JIMS resource utilization in GT-based grids, we should also consider porting JIMS to the GT service architecture.

There are also some objective factors, which hinder the design of such a system. These factors result from the fact that during development, numerous differentiated grid monitoring frameworks, specifications (sometimes conflicting[330]) and organizations standardizing various aspects of grid services should be taken into account. We can also notice slight reluctance to the use of the Java platform in the grid monitoring area, though this dissertation shows that this is no longer an issue. Finally, heterogeneity of hardware infrastructures requires shifting some work below the layer of environmental abstraction (in this case – below JVM), which necessarily limits portability of the solution.

7.4 Applicability of Proposed Solution

JIMS applicability ranges from homogenous HPC clusters to heterogeneous clusters of dynamically changing workstations used as computational nodes, and, finally, to mixed hardware and software infrastructure, including any JMX-enabled applications. However, JIMS applications are more applicable to grids with highly a changeable environment. In HPC clusters with constant configurations, JIMS can introduce unnecessary performance overhead (see section 6.4.1 on intrusiveness)[331]. JIMS can also be applied for JVM monitoring to allow administrators to monitor application servers running on single machines or in cluster environments (for example clustered installations of JBoss 4.0 AS [124]). The performed tests prove the applicability of the presented solution in mid-size grids.

7.5 Future Work

Directions for future work encompass many aspects of JIMS and include design of object-oriented, efficient storage for historical monitoring data, development of a suitable

[329] MX4J SOAP Connector is a supplementary extension to JSR-160 but is not part of the JMX Remote API specification.

[330] Vide WS-DM and WS-Management specifications.

[331] A similar situation occurs when using dynamic routing protocols in stub networks. In this case, even the most advanced dynamic routing protocols are unsuitable due to the overhead they produce in networks which are not changing or change very rarely. Despite their unquestionable advantages, they generate unnecessary communication overhead in static networks.

information model and further standardization in terms of current grid specifications. While the problem of storage for JIMS has already been addressed by related ongoing research described in [122], the problem of the information model remains the subject of further intensive work within the scope of investigations encompassing ontological and semantic aspects of monitored data. The proposed hierarchical model, reflecting three levels of monitored resources, may prove insufficient and require the development of a better information model for JIMS, interoperable with CIM or GLUE. In the future, the author plans to conduct further research on grid monitoring systems based on the described principles and integrate the achieved results with the features and possibilities offered by modern operating systems, autonomous systems (in terms of **AC**[332]) and systems offering resource management and virtualization technologies. Other directions of future work can cover further JIMS extensions, accommodating the limitations caused by current cluster address schemas with private IP addressing, firewalls and specialized security architectures. One of the last problems, which require further investigation, is the problem of dependency between numerous monitoring modules. At the time of JIMS design this was not a major issue, however nowadays, in the age of new operating systems with sophisticated resource management facilities and virtualization technologies, it becomes a more and more important and interesting topic. Another promising area of further research concerns the AC. Systems built according to this concept use a closed loop of control, with sensors and effectors. Such a configuration represents a modern approach to achieving given business goals and JIMS seems to be one of the key enablers of such an infrastructure, offering both monitoring (sensors) and management facilities (effectors).

[332] Autonomic Computing

8 Terminology

This supplementary chapter contains selected abbreviations, definitions as well as an index of more relevant terms used in the thesis.

8.1 Abbreviations

AC Autonomic Computing

ACC Academic Computer Centre (CYFRONET)

AGH Akademia Górniczo – Hutnicza

ANL Argonne National Laboratory

APART Automatic Performance Analysis: Real Tools

API Application Programming Interface

ASN.1 Abstract Syntax Notation 1

BER Basic Encoding Rules

BLOB Binary Large Object

BOA Basic Object Adapter

CCA Common Component Architecture

CCS Compute Cluster Server

CCLRC Council for the Central Laboratory of the Research Councils

CCM CORBA Component Model

CE Computing Element

CG CrossGrid

CIM Common Information Model

CLR Common Language Runtime

CLI Command Line Interface

CLUSTERIX National CLUSTER of LInuX Systems

CMIP Common Management Information Protocol

CMIS Common Management Information Services
CMOT CMIP/CMIS over TCP/IP
COBOL COmmon Business Oriented Language
COM Component Object Model
CORBA Common Object Request Broker Architecture
CPU Central Processing Unit
CG EU CrossGrid Project IST-2001-32243
CVS Concurrent Versions System
DARPA Defense Advanced Research Projects Agency
DBMS Database Management System
DCE Distributed Computing Environment
DCOM Distributed Component Object Model
DMTF Distributed Management Task Force
DII Dynamic Invocation Interface
DMTF Distributed Management Task Force
DSI Dynamic Skeleton Interface
DSRG Distributed Systems Research Group
DTF Distributed Terascale Facility
DWDM Dense Wavelength Division Multiplexing
EAR Enterprise Archive
EDA Event Driven Architecture
EDG EU DataGrid Project IST-2000-25182
EGEE Enabling Grids for E-sciencE
EIS Enterprise Information Systems
EJB Enterprise Java Beans
EPR Web Service Endpoint
DataTAG Data TransAtlantic Grid
FMA Federated Management Architecture
FSM Finite State Machine
FZK Forschungszentrum Karlsruhe
GAR Globus Archive
GDS Global Discovery Service
GEMINI Generic Monitoring and Instrumentation Infrastructure
GGF Global Grid Forum
GIOP General Inter-ORB Protocol

GIS Grid Information Service

GLUE Grid Laboratory Uniform Environment

GMA Grid Monitoring Architecture

GOC Grid Operations Center

GPM Grid Performance Measurement

GPL General Public License

GR Global Registry

GRAAP Grid Resource Allocation Agreement Protocol

GSI Grid Security Infrastructure[333], Grid Service Interface[334]

GSH Grid Service Handle

GSR Grid Service Reference

GT Globus Toolkit

GUI Graphical User Interface

HPC High Performance Computing

HTTP HyperText Transfer Protocol

ICIS Institute of Computer and Information Sciences[335]

ICM Interdisciplinary Centre for Mathematical and Computational Modelling

IDL Interface Definition Language

IEC International Electrotechnical Commission

IETF Internet Engineering Task Force

IIOP Internet Inter-ORB Protocol

INFN Istituto Nazionale di Fisica Nucleare

IOR Inter-Operable Reference

IP Internet Protocol

ISI Information Sciences Institute

ISO International Organization for Standardization

IST Information Society Technologies

ITU-T Telecommunication Standardization Sector of ITU

JAAS Java Authentication and Authorization Service

JANOS Java-oriented Active Network Operating System

[333] GT nomenclature.

[334] OGSI nomenclature.

[335] Częstochowa University of Technology.

JAR Java Archive
J2EE Java 2 Enterprise Edition
J2SE Java 2 Standard Edition
JCP Java Community Process
JDK Java Development Kit
JDMK Java Dynamic Management Kit
JIMS JMX-based Infrastructure Monitoring System
JNI Java Native Interface
JMX Java Management Extensions
JVM Java Virtual Machine
JRMP Java Remote Message Passing
JSP Java Server Pages
JSR Java Specification Request
JXTA Juxtapose
K-WfGrid Knowledge-based Workflow System for grid Applications
LAN Local Area Networks
LCFG Local Configuration
LCG LHC Computing Grid
LDAP Lightweight Directory Access Protocol
LHC Large Hadron Collider
LNCS Lecture Notes in Computer Science
MIB Management Information Base
MIT Massachusetts Institute of Technology
MDS Monitoring and Discovery System
MLet, M-Let Management Applet (the Java class and the JMX service, respectively)
MPI Message Passing Interface
MVM Multi-tasking Virtual Machine
NAT Network Address Translation
NCES Network Cost Estimation Service
NGOSS New Generation Operations Systems and Software
NMA Network Management Architecture[336], Network Monitor Architecture[337]

[336] SNMP nomenclature.

[337] EDG nomenclature.

NMS Network Management Station
NWG Network Working Group
NWS Network Weather Service
OASIS Organization for the Advancement of Structured Information Standards
OCM-G OMIS-Compliant Monitor for the Grid
OGF Open Grid Forum
OGSA Open Grid Services Architecture
OGSI Open Grid Services Infrastructure
OID Object Identifier
OMG Object Management Group
OMIS Online Monitoring Interface Specification
OMG Object Management Group
OO Object Oriented
ORB Object Request Broker
ORPC Object Remote Procedure Call
OS Operating System
OSI Open Systems Interconnection
OWAMP One-way Active Measurement Protocol
P2P Peer To Peer
PBS Portable Batch System
PDA Personal Digital Assistant
PDU Protocol Data Unit
PKI Public Key Infrastructure
POA Portable Object Adapter
POJO Plain Old Java Object
PPC Performance Prediction Component
PSNC Poznań Supercomputing and Networking Centre
QA Quality Assurance
RAS Roaming Access Server
RFC Request for Comments
RI Reference Implementation
RISC Reduced Instruction Set Computer
RMI Remote Method Invocation
RM-ODP Reference Model for Open Distributed Processing
RMON Remote Monitoring

REST	Representational State Transfer
R-GMA	Relational Grid Monitoring Architecture
RPC	Remote Procedure Call
SANTA-G	Grid-enabled System Area Networks Trace Analysis
SCEA	Sun Certified Enterprise Architect
SE	Storage Element
SG	SOAP Gateway
SGE	Sun Grid Engine
SGML	Standard Generalized Mark-up Language
SID	Shared Information/Data Model
SMI	Structure of Management Information
SNMP	Simple Network Management Protocol
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SoC	Separation of Concerns
SPARC	Scalable Processor ARChitecture
SSL	Secure Sockets Layer
STD	State-Transition Diagram
TCP	Transmission Control Protocol
TLS	Transport Layer Security
T&V	Testing and Validation
UDDI	Universal Description, Discovery, and Integration
UDP	User Datagram Protocol
UI	User Interface
UML	Unified Modelling Language
UNICORE	UNiform Interface to COmputing REsources
USC	University of Southern California
UST	University of Science and Technology
US-iVDGL	United States International Virtual Data Grid Laboratory
UTF	Unicode Transformation Format
WAN	Wide Area Network
WASP	Web Applications and Services Platform
WBEM	Web-Based Enterprise Management
WCNS	Wrocław Centre for Networking and Supercomputing
WG	Working Group

W3C	World Wide Web Consortium
WS	Web Services
WN	Worker Node
WP	Work Package
WSDL	Web Services Description Language
WS-DM	Web Services Distributed Management
WS-RF	Web Services Resource Framework
WWW	World Wide Web
VFS	Virtual File System
VO	Virtual Organization
XML	eXtensible Mark-up Language
XDR	eXternal Data Representation

8.2 Definitions

Adaptability	Variability in respect to, or under the influence of, external conditions. Susceptibility of a system to that variation whereby it becomes suited to or fitted for its conditions of environment[338].
Autonomous	A system that is being controlled by one organisational unit (university or a company) or being a subordinate to one administrative unit.
Component	A reusable software building block: a pre-built piece of encapsulated application code that can be combined with other components and with handwritten code to rapidly produce a custom application [160]. Here: Java MBean [147].
Computer cluster	A group of loosely coupled computers that work together closely so that in many respects it can be viewed as though it were a single computer.
Computing Element	According to Globus Toolkit/MDS terminology, a computing cluster with many computational nodes, seen by end user as a software unit with computational resources. Its main functionality is job management: submission, control and termination.
Configurability	The ability of a system showing how much the system can be changed or customized.
Deployment	Deployment is a common term used in the context of grid and enterprise applications. It's used to express both installation and startup of a module and concerns specialized, hot-deployable modules, like MBean/JAR (Java Archive), J2EE application packed as EAR (Enterprise Archive) or GT service, packaged as GAR (Globus Archive).
Discovery	Observing or finding networked objects in order to register the entities once they appear or to de-register them once they are inaccessible or the interconnecting network fails.
GDS node	Registry for SOAP gateways, used by client applications during initialization. Uses WS-based GDS module as registry for SOAP Gateways. Typically operates on chosen CEs (Computing Elements).
GMA	Grid Monitoring Architecture, as defined by the Open Grid Forum Performance Working Group, a general architecture for building grid monitoring systems. It consists of a producer, a consumer and a directory

[338] Definition derived from the biology domain, from "The Century Dictionary", The Century Co., New York, 1911.

	service.
Grid	The computing and data management infrastructure that provides the electronic underpinning for a global society in business, government, research, science and entertainment. Grid infrastructure provides users with the ability to dynamically link together resources as an ensemble to support the execution of large-scale, resource-intensive, and distributed applications [26].
Grid computing	Uses the resources of many separate computers connected by a network (usually the internet) to solve large-scale computation problems.
JCP	The Java Community Process program – procedure of creating a new standard in the area of Java technology (called JSR) in the Java Community [154]. See also: JSR.
JDMK	The Java Dynamic Management Kit (Java DMK), a set of Java classes and tools allowing easy development of secure monitoring and management solutions based on the Java Management Extensions (JMX) specifications and on the SNMP standard.
JIMS	Grid monitoring system, based on Java and JMX. See also: JMX.
JMX	An extension of the Java language, supporting local and remote resource management and monitoring. Defined by JSR 003[147], JSR 160[145], and JSR 255 [146], was primarily intended as an extension to Java 1.4. Finally it became a standard part of Java 5.0 (1.5) and seems to be an industry standard for resource representation, management and monitoring[339].
JSR	Java extension specification defined by the JCP process.
Management Station, Management Application	A host with specialized client software (management application), which allows the administrator to manage the monitoring system. Management station typically connects to one, chosen cluster access node (sometimes called CE) and is responsible for the presentation and visualization of monitoring data.
Module	A part of a program. Such a program is composed of one or more independently developed modules, which are combined when the program is linked during compile time or later, during runtime. In the presented monitoring system the term *module* denotes an online installable element such as a Java archive (JAR file), containing packed

[339] In the scope of Java technology.

	software components (MBeans) with corresponding helper configuration files (XML m-let text file) and compiled binary code (Java classes).
Monitoring data, Monitoring station	Data that is the result of monitoring. Accordingly, station (computational node, access node, or any networked host) that is being monitored is often called the monitoring station.
Node	A physical element object that represents a processing resource, generally having memory and processing capability – such as a server. Obviously, nodes include computers and other devices, but they can also be human resources or any processing resources [4], or any running applications.
Simulation	An imitation of a real situation, environment, state of affairs, or process. It generally entails representing certain key characteristics or behaviours of the target physical system.
SOAP Gateway	One of JIMS components, which acts as a proxy between monitored stations and client applications, translating requests from SOAP to JMX/RMI.
Storage Element	A computer system configured as a grid data storage server. SE is designed to stores data files to be processed and finally, the results.
Testbed	An experimentation platform for large development projects, allowing practical tests of scientific theories and new technologies. In the context of grids, this means the hardware and software infrastructure which allows execution of computational jobs across many geographically distributed nodes and clusters.
Virtual Organization	A mechanism for grouping users according to affiliation with a particular organization such as a research institute, a performed experiment or a field of research. It allows grid administrators to grant different authorization permissions for access to common grid resources [79].
Worker Node	A computer system designed for execution of grid jobs and computations [79].

8.3 Index

References

1. AdventNet, Inc., *SNMP Adaptor for JMX. Release 5*, Software documentation, available at: http://www.adventnet.com/products/snmpadaptor/AdventNetSNMPAdaptorForJMX.pdf, Pleasanton, CA, 2003, retrieved: Dec. 2006.
2. Alfonso C., Cabaler M., Hernandez V., "DIDA, a Distributed Discovery Architecture for grid Environments", *CGW'05 Workshop Proceedings*, ACC Cyfronet AGH, Kraków, 2006, pp. 106–113.
3. Allan B. A., Armstrong R. C., Wolfe A. P., Ray J., Bernholdt D. E., Kohl J. A., "The CCA core specification in a distributed memory SPMD framework", *Concurrency and Computation: Practice and Experience*, Vol. 14, Issue 5, John Wiley & Sons, Ltd., 2002, pp. 323–345.
4. Allen P. R., Bambara J. J., *Sun Certified Enterprise Architect for J2EE Study Guide (Exam 310-051)*, McGraw-Hill, Emeryville, CA, 2003, pp. 54–58.
5. Aloisio G., Cafaro M., Lezzi D., Engelen R., "Secure Web Services with Globus GSI and gSOAP, *Euro-Par 2003 Parallel Processing*, LNCS 0302, Springer-Verlag, Berlin/Heidelberg, 2003, pp. 421–426.
6. Anderson P., "Towards a High-Level Machine Configuration System", *System Administration Conference Proceedings of the 8th USENIX conference on System administration*, USENIX Association, Berkeley, CA, 1994, pp. 19–26.
7. Anderson P., Scobie A., "Large Scale Linux Configuration with LCFG", *Proceedings of the Atlanta Linux Showcase*, The USENIX Association, Berkeley, CA, 2000, pp. 363–372.
8. Andreozzi S., Bortoli N., Fantinel S., Ghiselli A., Tortone G., Vistoli C., "GridICE: a monitoring service for the grid", *CGW'03 Workshop Proceedings*, ACC Cyfronet AGH, Kraków, pp. 220–226.
9. Andreozzi S., *GLUE Schema Specification, Version 1.3, Draft 1*, available at: http://glueschema.forge.cnaf.infn.it/Spec/V13 (GLUESchema-1.3-v1.pdf), 2006, retrieved: Dec. 2006.
10. Apache Software Foundation, *Apache Synapse*, project homepage, http://incubator.apache.org/synapse, 2006, visited: Dec. 2006.
11. Arkesteijn B., Panton T., "Westhawk's SNMP Stack in Java", *The Simple Times*, Vol. 9, No. 1, online journal, available at: http://www.simple-times.org/pub/simple-times/issues/9-1.html, 2001, retrieved: Dec 2006.
12. Avery P., Foster I., *iVDGL Annual Report for 2004.2005*, Technical Report: GriPhyN/iVDGL 2005 - 14, available at: http://www.ivdgl.org/documents/ (doc--1563--ivdgl_report2005_v8-1.pdf), 2005, retrieved: Dec. 2006.
13. Baliś B., Bubak M., Dziwisz J., Rozkwitalski K., Hurnik S., "GEMINI - A Universal Monitoring Framework: Concept, Design, and First Prototype", *CGW'05 Workshop Proceedings*, ACC Cyfronet AGH, Kraków, 2005, pp. 122–123.
14. Baliś B., Bubak M., Dziwisz J., Truong H. L., Fahringer T., "Integrated Monitoring Framework for grid Infrastructure and Applications", *Innovation and the Knowledge Economy. Issues, Applications, Case Studies*, IOS Press, Ljubljana, 2005, pp. 269–276.
15. Baliś B., Bubak M., Rząsa W., Szepieniec T., "Efficiency of the GSI Secured Network Transmission", *Computational Science - ICCS 2004*, LNCS 3036, Springer-Verlag, Berlin/Heidelberg, 2004, pp. 107–115.
16. Ballinger K., Box D., Curbera F., *WS-MetadataExchange Specification*, available at: http://specs.xmlsoap.org/ws/2004/09/mex/WS-MetadataExchange0904.pdf, BEA Systems Inc., Computer Associates International, Inc., International Business Machines Corporation, Microsoft Corporation, Inc., SAP AG, Sun Microsystems, webMethods, 2004, retrieved: Dec. 2006.
17. Bałos K. et al., "JIMS", *Report on the Results of the WP3 2nd and 3rd Prototype*, ed. Mayer N., CrossGrid Deliverable D3.5, available at: http://www.eu-crossgrid.org/Deliverables/M24pdf/CG3.0-D3.5-v1.2-PSNC010-Proto2Status.pdf, ALGO, CSIC, CYFRONET, DATAMAT, ICM, PSNC, TCD, UAB, UCY, Poznań, 2004, pp. 203–228.
18. Bałos K., Bizoń L., Rozenau M., Zieliński K., "Interoperability Architecture for grid Networks Monitoring Systems", *CGW'03 Workshop Proceedings*, ACC Cyfronet AGH, Kraków, 2004, pp. 245–253.
19. Bałos K., Radziszowski D., Rzepa P., Zieliński K., Zieliński S., "Monitoring grid Resources - JMX in Action", *TASK Quarterly: Scientific Bulletin of Academic Computer Centre in Gdansk*, Vol. 8, No. 4, CI TASK, Gdańsk, 2004, pp. 487–501.
20. Bałos K., Wasilewski L., Wojtas K., Zieliński K., "Software Tool Construction for Deployment of JMX Services in Distributed Testbeds", *Testbeds and Research Infrastructures for the Development of Networks and Communities, 2006. TRIDENTCOM 2006.*, IEEE Xplore, 2006, pp. 460–470.
21. Bałos K., Zieliński K., "JIMS - the Uniform Approach to grid Infrastructure and Application Monitoring", *CGW'04 Workshop Proceedings*, ACC Cyfronet AGH, Kraków, 2005, pp. 160–167.
22. Bałos K., Zieliński K., "JMX-based grid Management Services", *Broadnets 2004: Workshop on Networks for grid Applications*, IEEE Computer Society, Los Alamitos, California, 2004, pp. 103–111.

23. Bałos K., Zieliński K., "WS-based Discovery Service for grid Computing Elements", *Advances in grid computing, EGC 2005*, LNCS 3470, Springer-Verlag, Berlin/Heidelberg, 2005, pp. 711–720.
24. Ban B., *Reliable Multicasting with the JGroups Toolkit*, Programmer's and User's Guide for JGroups, available at: http://www.jgroups.org/javagroupsnew/docs/manual/pdf/JGroups.pdf, 2006, pp. 3–5.
25. Bauer Ch., King G., *Hibernate in Action*, Manning Publications Co., Bruce Park Avenue, Greenwich, 2005.
26. Berman F., Fox G. C., Hey A. J. G., grid *Computing*, John Wiley & Sons, Ltd., West Sussex, 2003, pp. 9–16.
27. Bizoń A., Adamczyk R., *JXTA communication extension for JIMS*, Project homepage, http://student.agh.edu.pl/~abizon/pp/, visited: Dec. 2006.
28. Blumenthal U., Wijnen B., *User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3)*, Network Working Group: IBM T. J. Watson Research, RFC 2574[340], 1999, retrieved: Dec. 2006.
29. Boden N., Cohen D., Felderman R., Kulawik A., Seitz C., Seizovic J., Su W., "Myrinet: A gigabit per second local area network", *IEEE Micro*, Vol. 15, No. 1, IEEE Computer Society, 1995, pp. 29–36.
30. Bonnassieux F. et al., *DataGrid, Final Report on Network Infrastructure and Services*, DataGrid technical report, available at: http://www.hep.man.ac.uk/u/rich/DataGrid/WP7_Reports/DataGrid-07-D7-4-0206-2.0.pdf, 2004, pp. 35–36.
31. Bonnassieux F., Harakaly R., Primet P., Fernadez Rivera F., Bubak M., Gomez Tato A., Doallo R., "Automatic services discovery, monitoring and visualization of grid environments: The MapCenter approach", grid *Computing*, LNCS 2970, Springer-Verlag, Berlin, 2004, pp. 222–229.
32. Bordet S., Quiroz C. (Proj. admins), *MX4J - Open Source Java Management Extensions*, project homepage, http://mx4j.sourceforge.net, 2006, visited: Dec. 2006.
33. Borland Software Corporation, *Borland® SilkCentral® Test Manager*, Datasheet, available at: http://www.borland.com/resources/en/pdf/products/silk/silkcentral_performance_datasheet.pdf, Cupertino, CA, 2006, retrieved: Dec. 2006.
34. Boullon M., Rivera F. F., *The CrossGrid Performance Prediction Component*, User's manual, available at: https://savannah.fzk.de/distribution/crossgrid/crossgrid/wp2/wp2_4-perf/wp2_4_2-perfpred-PPC/docs/user-manual.pdf, 2005, retrieved: Dec. 2006.
35. Bowden T., Bauer B., J. Nerin, *The /proc filesystem*, Chapter 1.2 of Linux Kernel 2.6 documentation, 2006.
36. Bubak M., Malawski M., Młynarczyk G., Nowakowski P., Pająk R., Rycerz K., Turała M., "Quality assurance aspects in the EU CrossGrid Project", *CGW'03 Workshop Proceedings*, ACC Cyfronet AGH, Kraków, 2004, pp. 52–58.
37. Bubak M., Malawski M., Rycerz K., Turała M., "Crossgrid - tools and services for interactive grid applications", *TASK Quarterly: Scientific Bulletin of Academic Computer Centre in Gdansk*, Vol. 8, No. 4, CI TASK, Gdańsk, 2004, pp. 503–512.
38. Bubak M., Malawski M., Zając K., "Architecture of the grid for Interactive Applications", *Computational Science – ICCS 2003*, LNCS 2657, Springer-Verlag, Berlin/Heidelberg, 2003, pp. 207–213.
39. Bubak M., Malawski M., Zając K., "The CrossGrid Architecture: Applications, Tools, and grid Services", grid *Computing*, LNCS 2970, Springer-Verlag, Berlin/Heidelberg, 2004, pp. 309–316.
40. Cameron D. et al., "Replica Management in the European DataGrid Project, Springer Netherlands", *Journal of grid Computing*, Vol. 2, No. 4, Springer-Verlag, Netherlands, 2004, pp. 341–351.
41. Cancio G., Kleinwort T., Tomlin W. et al., "CERN uses ELFms to manage enormous range of systems", CERN Computer Newsletter, Vol. XL, Issue 3, CERN IT Department, CERN, 2005, pp. 6–8.
42. Case J., McCloghrie K., Rose M., Waldbusser S., *SNMPv2 Working Group, Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)*, Network Working Group: SNMP Research, Inc., Cisco Systems, Inc., Dover Beach Consulting, Inc., International Network Services, RFC 1905, 1996, retrieved: Dec. 2006.
43. Case J., McCloghrie K., Rose M., Waldbusser S., *Structure of Management Information for version 2 of the Simple Network Management Protocol (SNMPv2)*, Network Working Group: SNMP Research, Inc., Hughes LAN Systems, Dover Beach Consulting, Inc., Carnegie Mellon University, RFC 1902, 1993, retrieved: Dec. 2006.
44. Case J., McCloghrie K., Rose M., Waldbusser S., *Textual Conventions for Version 2 of SNMP*, Network Working Group: SNMPv2 Working Group, SNMP Research, Inc., Cisco Systems, Inc., Dover Beach Consulting, Inc., International Network Services, RFC 1903, 1996, retrieved: Dec. 2006.

[340] All RFCs are available online from the EITF web page at http://www.ietf.org/rfc/rfc*n*.txt, where *n* is the number of the specification.

45. Cisco Systems, Inc., “Remote Monitoring”, *Internetworking Technologies Handbook*, available at: http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/rmon.pdf, Chapter 5, 2006, pp. 1–3.
46. Coghlan B., Kenny S. et al., "SANTA-G", *Report on the Results of the WP3 2nd and 3rd Prototype*, ed. Mayer N., CrossGrid Deliverable D3.5, available at: http://www.eu-crossgrid.org/Deliverables/M24pdf/CG3.0-D3.5-v1.2-PSNC010-Proto2Status.pdf (retrieved: Dec. 2006), ALGO, CSIC, CYFRONET, DATAMAT, ICM, PSNC, TCD, UAB, UCY, Poznań, 2004, pp. 182–202.
47. Cooke A. W. et al., “The Relational grid Monitoring Architecture: Mediating Information about the grid”, *Journal of grid Computing*, Vol. 2, No. 4/2004, Springer-Verlag, Netherlands, 2005, pp. 323–339.
48. Czajkowski K., Ferguson D. F., Foster I., Frey J., Graham S., Sedukhin I., Snelling D., Tuecke S., Vambenepe W., *The WS-Resource Framework, Version 1.0*, Computer Associates International, Inc., Fujitsu Limited, Hewlett-Packard Development Company, International Business Machines Corporation and The University of Chicago, White Paper, available at: http://www-106.ibm.com/developerworks/library/ws-resource/ws-wsrf.pdf, 2004, retrieved: Dec. 2006.
49. Czajkowski K., Fitzgerald S., Foster I., Kesselman C., “Grid Information Services for Distributed Resource Sharing”, *10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10 2001)*, IEEE Xplore, San Francisco, CA, 2001, pp. 181–194.
50. Data TransAtlantic grid, EU IST-2001-32459 DataTag Project homepage, http://datatag.web.cern.ch/datatag, 2006, visited: Dec. 2006.
51. David Baum, “Get Control”, *Oracle Magazine*, Issue July/August 2006, Oracle, Santa Barbara, CA, 2006.
52. DeSchon A. L et al., *A survey of data representation standards*, Network Working Group, RFC 971, 1986, retrieved: Dec. 2006.
53. Distributed Management Task Force, Inc. (DMTF), *Common Information Model (CIM) Core Model, Version 2.4*, White Paper, available at: http://www.dmtf.org/standards/documents/CIM/DSP0111.pdf, 2000, retrieved: Dec. 2006.
54. Distributed Management Task Force, Inc., DMTF homepage, http://www.dmtf.org, 2006, visited: Dec. 2006.
55. Distributed Management Task Force, Inc., *WS-Management*, available at: http://xml.coverpages.org/WS-Management200506.pdf, 2005, retrieved: Dec. 2006.
56. Donno F., "DataTAG Contributing to LCG-0 Pilot Startup", *CERN Computer Newsletter*, Vol. XXXVIII, Issue 1, CERN IT Department, CERN, 2005, p. 2.
57. Dutka Ł., Korcyl K., Zieliński K., Kitowski J., Słota R., Funika W., Bałos K., Skitał Ł., Kryza B., “Interactive European grid Environment for HEP Application with Real Time Requirements”, *CGW’06 Workshop Proceedings*, ACC Cyfronet AGH, Kraków, 2007, in press.
58. Dutka Ł., Kryza B., Krawczyk K., Majewska M., Slota R., Hluchy L., Kitowski J., “Component-expert Architecture for Supporting grid Workflow Construction Based on Knowledge”, *Innovation and the Knowledge Economy, Issues, Applications, Case Studies*, Vol. 2, IOS Press, 2005, pp. 239–246.
59. Dutka Ł., Slota R., Nikolow D., Kitowski J., “Optimization of Data Access for grid Environment”, grid *Computing*, LNCS 2970, Springer-Verlag, Berlin/Heidelberg, 2004, pp. 93–102.
60. Edwards J., Keith W., *Core Jini*, Prentice Hall PTR, Upper Saddle River, NJ, 1999, pp. 197–205.
61. Englander R., *Developing Java Beans*, O'Reilly, Sebastopol, CA, 1997.
62. Erwin D. W., Snelling D. F., “UNICORE: A grid Computing Environment”, *Euro-Par 2001: Parallel Processing: 7th International Euro-Par Conference Manchester, UK August 28-31, 2001, Proceedings*, LNCS 2150, Springer-Verlag, Berlin/Heidelberg, 2001, pp. 825–839.
63. Feldkuhn L., Erickson J., “Event Management as a Common Functional Area of Open Systems Management”, *Integrated Network Management: IFIP TC 6/WG 6.6 Symposium on Integrated Network Management*, North-Holland, Boston, Massachusetts, 1989, pp. 365–376.
64. Foster I. et al., *Modeling Stateful Resources with Web Services (WS-Resource)*, Specification, available at: http://www-106.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf, Computer Associates International, Inc., Fujitsu Limited, Hewlett-Packard Development Company, International Business Machines Corporation and The University of Chicago, 2004, retrieved: Dec. 2006.
65. Foster I., “Globus Toolkit Version 4: Software for Service-Oriented Systems”, *Network and Parallel Computing*, LNCS 3779, Springer-Verlag, Berlin/Heidelberg, 2005, pp. 2–13.
66. Foster I., “What Is The grid? A Three Point Checklist”, *Daily News And Information For The Global grid Community*, Issue July 22/2002, Vol. 1, No. 6, Tabor Communications Inc., San Diego, CA, 2002.
67. Foster I., Kesselman C., *The GRID: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers Inc., San Francisco, CA, 1998.
68. Foster I., Kishimoto H., Savva A. et al., *The Open grid Services Architecture, Version 1.0*, GGF OGSA-WG Specification, available at: http://www.gridforum.org/documents/GWD-I-E/GFD-I.030.pdf, 2003, retrieved: Dec. 2006.

69. Foster I., Tuecke S., "The Different Faces of IT as Service", *Enterprise distributed computing*, Vol. 3, Issue 6, ACM Press, New York, NY, July/August 2005, pp. 26–29.
70. Frankowski G., Bałos K., Wyrzykowski R., Karczewski K., Krzywania R., Kosiński J., "Integrating Dynamic Clusters in CLUSTERIX Environment", *CGW'05 Workshop Proceedings*, ACC Cyfronet AGH, Kraków, 2006, pp. 280–292.
71. Freeman E., Hupfer S., Arnold K., *JavaSpaces*, Addison-Wesley Longman Ltd., UK, 1999, pp. 3–18.
72. Freier A. O., Karlton P., Kocher P. C., *The SSL Protocol, Version 3.0*, Transport Layer Security Working Group, available at: http://wp.netscape.com/eng/ssl3/draft302.txt, 1996, retrieved: Dec. 2006.
73. Frey J. et al., *WS-ResourceLifetime, Version 1.1*, Specification, available at: http://www-106.ibm.com/developerworks/library/ws-resource/ws-resourcelifetime.pdf, Computer Associates International Inc., Fujitsu Limited, Hewlett-Packard Development Company, International Business Machines Corporation and The University of Chicago, 2004, retrieved: Dec. 2006.
74. Gagliardi F., Jones B., Reale M., Burke S., "European DataGrid Project: Experiences of Deploying a Large Scale Testbed for E-science Applications", *Performance Evaluation of Complex Systems: Techniques and Tools: Performance 2002. Tutorial Lectures*, LNCS 2459, Springer Berlin/Heidelberg, 2002, pp. 480–504.
75. Gerndt M., Wismüller R. et al., *Performance Tools for the grid: State of the Art and Future*, APART White Paper, Version 1.0, available at: http://www.lpds.sztaki.hu/~zsnemeth/apart/repository/gridtools.pdf, 2006, retrieved: Dec. 2006.
76. Globus Alliance, *Globus Toolkit 4.0 Release Manuals*, Online documentation, http://www.globus.org/toolkit/docs/4.0/, 2006, visited: Dec. 2006.
77. Globus Alliance, *GT 4.0: Information Services: Index*, Online document, available at: http://www.globus.org/toolkit/docs/4.0/info/index/index.pdf, 2005, retrieved: Dec. 2006.
78. Gombás G., Marosi C. A., Balaton Z., "Grid Application Monitoring and Debugging Using the Mercury Monitoring System", *Advances in grid Computing - EGC 2005*, LNCS 3470, Springer-Verlag, Berlin/Heidelberg, 2005, pp. 193–199.
79. Gomes J., Marco J., *Middleware Test Procedure*, CrossGrid Deliverable D4.1, Appendix D, available at: http://www.lip.pt/computing/projects/crossgrid/doc/deliverables/TestProcedure.pdf, 2006, retrieved: Dec. 2006.
80. Gong L., "JXTA: A Network Programming Environment", *Internet Computing*, IEEE Inc., Issue May/June 2001, Vol. 5, No. 3, Los Alamitos, CA, 2001, pp. 88–95.
81. Gosling J., Joy B., Steele G., Bracha G., *The Java™ Language Specification*, Addison-Wesley, Boston, 2005.
82. Govindaraju M., Slominski A., Chiu K., Liu P., Engelen R., Lewis M.J., "Toward characterizing the performance of SOAP toolkits", grid *Computing*, IEEE Computer Society, Washington, 2004, pp. 365–372.
83. Graham S. et al., *WS-ResourceProperties*, Specification, available at: http://www-106.ibm.com/developerworks/library/ws-resource/ws-resourceproperties.pdf, Computer Associates International Inc., Fujitsu Limited, Hewlett-Packard Development Company, International Business Machines Corporation and The University of Chicago, 2003, retrieved: Dec. 2006.
84. Graham S. et al., *WS-Topics, Version 1.0*, Specification, available at: ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-Topics.pdf, Akamai Technologies, Computer Associates International, Inc., Fujitsu Limited, Hewlett-Packard Development Company, International Business Machines Corporation, SAP AG, Sonic Software Corporation, The University of Chicago and Tibco Software Inc., 2004, retrieved: Dec. 2006.
85. Hamilton G. et al., *JavaBeans (TM),* Specification, available at: http://java.sun.com/products/javabeans/docs/spec.html, Sun Microsystems, Inc., Mountain View, CA, 1997, retrieved: Dec. 2006.
86. Harrington D., Presuhn R., Wijnen B., *An Architecture for Describing SNMP Management Frameworks*, Network Working Group: Cabletron Systems, Inc., BMC Software, Inc., IBM T. J. Watson Research, RFC 2571, 1999, retrieved: Dec. 2006.
87. Head M. R., Govindaraju M., Slominski A., Liu P. , Abu-Ghazaleh N., Engelen R., Chiu K., Lewis M. J., "A Benchmark Suite for SOAP-based Communication in grid Web Services", *Conference on High Performance Networking and Computing Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, IEEE Computer Society, Washington, pp. 19–32.
88. Hodges J., Morgan R. et al., *Lightweight Directory Access Protocol (v3): Technical Specification*, Network Working Group: Sun Microsystems Inc., University of Washington, RFC 3377, 2002, retrieved: Dec. 2006.
89. Huber V., "UNICORE: A grid Computing Environment for Distributed and Parallel Computing", *Parallel Computing Technologies: 6th International Conference, PaCT 2001, Novosibirsk, Russia, September 3-7, 2001, Proceedings*, LNCS 2127, Springer-Verlag, Berlin/Heidelberg, 2001, pp. 258–265.
90. ISO/IEC, *Information technology – Open Distributed Processing – Reference model*, International Standard ISO/IEC 10746-1, 1998.

91. ITU-T, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*, ITU-T Recommendation X.680, available at: http://www.itu.int/ITU-T/studygroups/com17/languages/X.680-0207.pdf, International Standard ISO/IEC 8824-1, 2002, retrieved: Dec. 2006.
92. Jones B., "An Overview of the EGEE Project", *Peer-to-Peer, grid, and Service-Orientation in Digital Library Architectures*, LNCS 3664, Springer-Verlag, Berlin/Heidelberg, 2005, pp. 1–8.
93. Joyce J., Lomow G., Slind K., Unger B., "Monitoring Distributed Systems", *ACM Transactions on Computing Systems*, Vol. 5, No. 2, ACM, New York, NY, 1987, pp. 121–150.
94. Kenny S., Coghlan B., "Towards a grid-wide Intrusion Detection System", *Advances in grid Computing - EGC 2005*, Springer-Verlag, Berlin/Heidelberg, 2005, pp. 275–284.
95. Kopta P., Kuczyński T., Wyrzykowski R., "Grid Access and User Interface in CLUSTERIX Project", *Parallel Processing and Applied Mathematics*, LNCS 3911, Springer-Verlag, Berlin/Heidelberg, 2006, pp. 249–257.
96. Kosińska J., "Jiro Platform", original title in polish: "Platforma Jiro", *Telenet Forum*, No. 04/2002, LUPUS, Warszawa, pp. 44–48.
97. Kouřil D. et al., *Logging and Bookkeeping Service for the DataGrid*, EU DataGrid Deliverable: DataGrid-01-TEN-0109-1_0, available at: http://server11.infn.it/workload-grid/docs/DataGrid-01-TEN-0109-1_0.pdf, 2001, pp. 5–8.
98. Kwiatkowski J., Pawlik M., Frankowski G., Bałos K., Wyrzykowski R., Karczewski K., "Dynamic clusters available under Clusterix grid", *PARA'06 State-of-the-Art in Scientific and Parallel Computing*, To appear in LNCS, Springer-Verlag, Berlin/Heidelberg, 2006, in press.
99. Laurentowski A., *A Distributed Monitoring Service for Software Objects*, Ph.D. thesis, AGH-UST, Department of Computer Science, Kraków, 2001, pp. 13–17.
100. Ławniczek B., Majka G., *Jiro Based grid Infrastructure Monitoring*, M.Sc. thesis, AGH-UST, Department of Computer Science, Kraków, 2002.
101. Ławniczek B., Majka G., Zieliński K., Zieliński S., "Jiro-Based grid Infrastructure Monitoring System - State of Development Report", grid *Computing*, LNCS 2970, Springer-Verlag, Berlin/Heidelberg, 2004, pp. 249–256.
102. Leese M., Tasker R., "GridMon: grid Network Performance Monitoring for e-Science", *Proceedings of the UK e-Science All Hands Meeting 2004*, EPSRC, Nottingham UK, 2004, pp. 907–914.
103. Liang S., *The Java™ Native Interface*, Addison-Wesley, Reading, Massachusetts, 1999, pp. 11–16.
104. Livshits B., Whaley J., Lam M., "Reflection Analysis for Java", *Programming Languages and Systems*, LNCS 3780, Springer-Verlag, Berlin/Heidelberg, 2005, pp. 139–160.
105. Maguire T., Snelling D., *Web Services Service Group 1.2 (WS-ServiceGroup)*, OASIS Working Draft, available at: http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ServiceGroup-1.2-draft-02.pdf, OASIS Open, 2004, retrieved: Dec. 2006.
106. Massie M. L., Chun B. N., Culler D. E., "The ganglia distributed monitoring system: design, implementation, and experience", *Parallel Computing*, Vol. 30/2004, Elsevier B.V., Holland, 2004, pp. 817–840.
107. McGovern J. et al., *A Practical Guide to Enterprise Architecture*, Prentice Hall PTR, Upper Saddle River, NJ, 2003, pp. 63–89.
108. Microsoft Corporation, Inc., *Service Oriented Architecture*, Online article in Microsoft Developer Network (MSDN), available at: http://msdn2.microsoft.com/en-us/architecture/aa948857.aspx, 2006, retrieved: Dec. 2006.
109. Midura J., Bałos K., Zieliński K., "Global Discovery Service for JMX Architecture", *Computational Science - ICCS 2004*, LNCS 3038, Springer-Verlag, Berlin/Heidelberg, 2004, pp. 114–118.
110. Moerdijk A. J., Klostermann L., "Opening the networks with Parlay/OSA: standards and aspects behind the APIs", *IEEE Network*, Vol. 17, Issue 3, IEEE Xplore, Netherlands, 2003, pp. 58–64.
111. Mort Bay Consulting, *jetty6 - Jetty WebServer*, Jetty Project homepage, http://jetty.mortbay.org, 2006, visited: Dec. 2006.
112. Object Management Group, Inc., *CORBA Component Model Specification, Version 4.0*, OMG Specification, available at: http://www.omg.org (06-04-01.pdf), 2006, retrieved: Dec. 2006.
113. Object Management Group, Inc., *CORBA: Core Specification,Version 3.0.3*, Specification, available at: http://www.omg.org (04-03-12.pdf), 2004, retrieved: Dec. 2006.
114. Object Management Group, Inc., *Event Service Specification, Version 1.2*, Specification, available at: http://www.omg.org (04-10-02.pdf), 2004, retrieved: Dec. 2006.
115. Object Management Group, Inc., *Notification Service Specification, Version 1.1*, Specfication, available at: http://www.omg.org (04-10-11.pdf), 2004, retrieved: Dec. 2006.
116. Open Grid Forum, OGF homepage, http://www.ogf.org, 2006, visited: Dec. 2006.

117. Organization for the Advancement of Structured Information Standards, OASIS homepage, http://www.oasis-open.org, 2006, visited: Dec. 2006.
118. Padee A., Nawrocki K. et al., "Postprocessing Monitoring System", *Report on the Results of the WP3 2nd and 3rd Prototype*, ed. Mayer N., CrossGrid Deliverable D3.5, available at: http://www.eu-crossgrid.org/Deliverables/M24pdf/CG3.0-D3.5-v1.2-PSNC010-Proto2Status.pdf, ALGO, CSIC, CYFRONET, DATAMAT, ICM, PSNC, TCD, UAB, UCY, Poznań, 2004, pp. 148–167.
119. Peterson L., Gottlieb Y., Hibler M., Tullmann P., Lepreau J., Schwab S., Dandekar H., Purtell A., Hartman J., "An OS Interface for Active Routers", *Selected Areas in Communications*, IEEE Xplore, 2001, pp. 473–487.
120. Poznański P., *Framework for Managing grid-enabled Large Scale Computing Fabrics*, Ph.D. thesis, AGH-UST, Department of Computer Science, Kraków, 2005.
121. Primet Vicat-Blanc P., Harakaly R., Bonnassieux F., "Grid Network Monitoring In The European Datagrid Project", *The International Journal of High Performance Computing Applications*, Vol. 18, No. 3, Sage Publications, Thousand Oaks, CA, 2004, pp. 293–304.
122. Radziszowski D., *Scalable, component-based system for acquisition and storage of data resulting from distributed systems monitoring*, original title in polish: *Skalowalny, komponentowy system zbierania i przechowywania danych pochodzacych z monitorowania systemów rozproszonych*, Ph.D thesis, AGH-UST, Department of Computer Science, Kraków, 2006.
123. Rajic H. et al., *Distributed Resource Management Application API Specification 1.0*, Global Grid Forum, available at: http://www.ggf.org/documents/GWD-R/GFD-R.022.pdf, 2004, retrieved: Dec. 2006.
124. Red Hat, *Red Hat JEMS: The Open Source Platform for SOA, The platform for flexibility, interoperability, and choice*, JBoss White Paper, available at: http://www.jboss.com/pdf/JEMS_Soa_white_paper_08_06.pdf, 2006, retrieved: Dec. 2006.
125. Rosmanith H., Kranzlmuller D., "glogin - A Multifunctional, Interactive Tunnel into the grid", grid, IEEE Computer Society, Los Alamitos, California, 2004, pp. 266–272.
126. Rycerz K., *Grid-based HLA Simulation Support*, Ph.D. thesis, University of Amsterdam, Amsterdam, 2006, p. 66.
127. S. Graham et al., *WS-Base Notification Specification*, OASIS Working Draft, available at: ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-BaseN.pdf, 2004, retrieved: Dec. 2006.
128. Schopf J. M., Nitzberg B., "Grids: Top Ten Questions", *Scientific Programming*, Vol. 10, Issue 2, IOS Press, Amsterdam, 2002, pp. 103–111.
129. Schopf J. M., Pearlman L., Miller N., Kesselman C., Foster I., D'Arcy M., Chervenak A., "Monitoring the grid with the Globus Toolkit MDS4", *Journal of Physics: Conference Series 46 (2006), SciDAC 2006*, Institute of Physics Publishing, London, 2006, pp. 521–525.
130. Schulte R. W., Natis Y. V., *Service Oriented Architecture*, Gartner Report, available at: http://www.gartner.com, document id.: SPA-00-7425, 1996, verified: Dec. 2006.
131. Sekman T., Smęt M., *Dynamically configurable system for grid resource monitoring*, original title in polish: *Dynamicznie konfigurowalny system monitorowania zasobów gridowych*, M.Sc. thesis, AGH-UST, Department of Computer Science, Kraków, 2004.
132. Shalunov S., Teitelbaum B., Karp A., Boote J., Zekauskas M. (Internet2), *A One-way Active Measurement Protocol (OWAMP)*, Network Working Group, RFC 4656, 2006, retrieved: Dec. 2006.
133. Siddiqi Z., *Using JMX and J2SE 5.0 to Securely Manage Web Applications*, White Paper, available at: http://today.java.net/pub/a/today/2005/11/15/using-jmx-to-manage-web-applications.html, 2005, retrieved: Dec. 2006.
134. Software Mind, *Partial Quality Assurance Report, Code Quality Indicators - static source code metrics and quality indicators*, available at: http://www.eu-crossgrid.org/wp5-1-login/QA_reports/February_2005/code_QI_static_March2005.pdf, Software Mind Sp. z o. o., 2005, retrieved: Dec. 2006.
135. Soley R. M., *Object Management Architecture Guide*, John Wiley & Sons, Inc., Framingham, MA, 1995.
136. Srinivas D., *Axis2 – Performance Testing Round #1*, Online article, available at: http://www.wso2.net/2006/05/axis2_performance_testing_round_1, WSO2, Boston, USA, 2006, retrieved: Dec. 2006.
137. Stallings R., SNMP, SNMP v2.0 and CMIP, *A Practical Guide to Network Management Standards*, Addison Wesley, Reading, Massachusetts, 1993.
138. Steve Graham et al., *Publish-Subscribe Notification for Web Services*, Specification, available at: http://www-106.ibm.com/developerworks/library/ws-pubsub/WS-PubSub.pdf, Akamai Technologies, Computer Associates International, Inc., Fujitsu Limited, Hewlett-Packard Development Company, International Business Machines Corporation, SAP AG, Sonic Software Corporation, Tibco Software Inc. and The University of Chicago, 2004, retrieved: Dec. 2006.

139. Steve Graham, Niblett P., et al., *Web Services Brokered Notification (WS-BrokeredNotification) Version 1.0*, Specification, available at: ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-BrokeredN.pdf, Akamai Technologies, Computer Associates International, Inc., Fujitsu Limited, Hewlett-Packard Development Company, International Business Machines Corporation, SAP AG, Sonic Software Corporation, The University of Chicago and Tibco Software Inc., 2004, retrieved: Dec. 2006.
140. Stewart B. et al., *Definitions of Managed Objects for Character Stream Devices*, Network Working Group: Xyplex, Inc., RFC 1316, 1992, retrieved: Dec. 2006.
141. Sun Microsystems, "VisualGC", *Trouble-Shooting and Diagnostic Guide*, Online documentation, available at: http://java.sun.com/j2se/1.5/pdf/jdk50_ts_guide.pdf, Santa Clara, CA, 2006, retrieved: Dec. 2006, p. 30.
142. Sun Microsystems, *Federated Management Architecture (FMA) Specification, Version 1.0*, JSR 009, available at: http://jcp.org/en/jsr/detail?id=9 (fma_specification_1_0_0.pdf)[341], Palo Alto, CA, 2000, retrieved: Dec. 2006.
143. Sun Microsystems, *Java Dynamic Management Kit 5.1 Tutorial*, available at: http://java.sun.com/products/jdmk/docs/JDMK51TUTOR.pdf, Santa Clara, CA, 2004, retrieved: Dec. 2006, pp. 245–259.
144. Sun Microsystems, *Java Spaces. Service Specification, v1.2.1*, available at: http://www.sun.com/software/jini/specs/js1_2_1.pdf, Palo Alto, CA, 2002, retrieved: Dec. 2006.
145. Sun Microsystems, *Java™ Management Extensions (JMX™) Specification, version 1.4*, JSR 160, available at: http://jcp.org/en/jsr/detail?id=160 (jmx-1_4-mrel3-spec.pdf), Canta Clara, CA, 2006, retrieved: Dec. 2006.
146. Sun Microsystems, *Java™ Management Extensions (JMX™) Specification, version 2.0*, JSR 255, available at: http://jcp.org/en/jsr/detail?id=255, 2006, retrieved: Dec. 2006.
147. Sun Microsystems, *Java™ Management Extensions Instrumentation and Agent Specification v. 1.2*, JSR 003, available at: http://jcp.org/en/jsr/detail?id=3 (jmx_1.2_spec.pdf), Santa Clara, CA, 2002, retrieved: Dec. 2006.
148. Sun Microsystems, *Java™ RemoteMethodInvocation Specification*, available at: http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf, Santa Clara, CA, 2004, retrieved: Dec. 2006, p. 87.
149. Sun Microsystems, *Jini™ Architecture Specification, v1.2*, available at: http://www.sun.com/software/jini/specs/jini1_2_1specs.zip (jini.pdf), Palo Alto, CA, 2001, retrieved: Dec. 2006.
150. Sun Microsystems, *Jini™ Technology Core Platform Specification, v1.2*, available at: http://www.sun.com/software/jini/specs/jini1_2_1specs.zip (core.pdf), Palo Alto, CA, 2001, retrieved: Dec. 2006.
151. Sun Microsystems, *Migrating to Sun Java, System Application Server 8*, available at: http://www.sun.com/software/products/appsrvr/wp_as8.pdf, White Paper, Santa Clara, CA, 2005, retrieved: Dec. 2006.
152. Sun Microsystems, *Sun™ ONE grid Engine Administration and User's Guide*, available at: http://gridengine.sunsource.net/download/Manuals53/SGE53AdminUserDoc.pdf, Santa Clara, CA, 2002, retrieved: Dec. 2006, pp. 4–9.
153. Sun Microsystems, *The BeanShell Scripting Language*, JSR-274, available at: http://jcp.org/en/jsr/detail?id=274, 2005, retrieved: Dec. 2006.
154. Sun Microsystems, *The Java Community Process Program*, JCP homepage, http://jcp.org, 2006, retrieved: Dec. 2006.
155. Sun Microsystems, *Web Services Connector for Java, Management Extensions (JMX™) Agents*, JSR 262 Specification Early Draft, available at: http://jcp.org/en/jsr/detail?id=262 (ws_connector_jmx_agents-1_0-edr-spec.pdf), 2006, retrieved: Dec. 2006.
156. Systinet, *WASP Server for Java, 5.0, Product Documentation*, available at: http://www.systinet.com/doc/wasp_jserver/pdf/waspj.pdf, 2004, retrieved: Dec. 2006.
157. TeleManagement Forum, TM Forum homepage, http://www.tmforum.org, 2006, visited: Dec. 2006.
158. Tennenhouse D. L., Smith J. M., Sincoskie W. D., Wetherall D. J., Minden G. J., "A Survey of Active Network Research", *IEEE Communications Magazine*, Vol. 35, No. 1, IEEE Communications Society, 1997, pp. 80–86.

[341] This and other JSR related documents are available online at: http://jcp.org/en/jsr/detail?id=*n*, where *n* is the number of specification.

159. The Apache Software Foundation, *AXIS, Apache Web Services Project*, project homepage, http://ws.apache.org/axis, 2006, visited: Dec. 2006.
160. Thomas A., *Enterprise JavaBeans Technology – Server Component Model for the Java Platform*, White Paper, Patricia Seybold Group, Boston, MA, 1998.
161. Tierney B., Aydt R., Gunter D., Smith W., Swany M., Taylor V., Wolski R., *A grid Monitoring Architecture*, GGF Specification, available at: http://www.gridforum.org/documents/GFD.7.pdf, 2002, retrieved: Dec. 2006.
162. Tirumala A., Qin F., Dugan J., Ferguson J., Gibbs K., *Iperf version 1.7.0 Iperf 2.0.2,* NLANR Distributed Applications Support Team, Project homepage, available at: http://dast.nlanr.net/Projects/iperf/, 2006, retrieved: Dec. 2006.
163. Tsouloupas G., Dikaiakos M., "Design and Implementation of gridBench", *Advances in grid Computing - EGC 2005*, LNCS 3470, Springer-Verlag, Berlin/Heidelberg, 2005, pp. 211–225.
164. Tuecke S., Czajkowski K., Foster I. et al., *The Open grid Services Infrastructure (OGSI), Version 1.0*, GGF OGSI-WG Specification, available at: http://www.gridforum.org/documents/GFD.15.pdf, 2003, retrieved: Dec. 2006.
165. Tuecke S., Liu L., S. Meder, *Web Services Base Faults 1.2*, OASIS Working Draft, available at: http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-BaseFaults-1.2-draft-02.pdf, 2004, retrieved: Dec. 2006.
166. Tullmann P., Hibler M., Lepreau J., "Janos: A Java-oriented OS for Active Networks", *Selected Areas of Communication*, Vol. 19, No. 3, IEEE Xplore, Piscataway, NJ, 2001, pp. 501–510.
167. Veridian Information Solutions, Inc., *Portable Batch System Administrator Guide*, Online documentation, available at: http://testbed.gridcenter.or.kr/kor/technical_doc/etc/v2.3_admin.pdf, Veridian Systems, Mountain View, CA, 2000, retrieved: Dec. 2006.
168. Waldbusser S., *Remote Network Monitoring Management Information Base*, Network Working Group: Carnegie Mellon University, RFC 1271, 1991, retrieved: Dec. 2006.
169. Waldbusser S., *Structure and Identification of Management Information for TCP/IP-based internets*, Network Working Group: Carnegie Mellon University, RFC 1065, 1991, retrieved: Dec. 2006.
170. Waters G. et al., *User-based Security Model for SNMPv2*, Network Working Group: Bell-Northern Research Ltd., RFC 1910, 1996, retrieved: Dec. 2006.
171. Wegner P., "Dimensions of Object-Based Language Design", *Conference on Object Oriented Programming Systems Languages and Applications*, ACM Press, New York, 1987, pp. 168–182.
172. Wojtas K., Wasilewski L., *Global system for monitoring of computer clusters with dynamically changing configuration*, original title in polish: *Globalny system monitorowania klastrów komputerowych o dynamicznie zmiennej konfiguracji*, M.Sc. thesis, AGH-UST, Department of Computer Science, Kraków, 2005, pp. 52–82.
173. Wojtas K., Wasilewski L., K. Bałos, K. Zieliński, "Discovery Service for JMX-Enabled Monitoring System. JIMS Case Study", *CGW'05 Workshop Proceedings*, ACC Cyfronet AGH, Kraków, 2006, pp. 148–157.
174. World Wide Web Consortium, *WS-Addressing*, W3C Working Draft, available at: http://www.w3.org/TR/2004/WD-ws-addr-core-20041208/ws-addr-core.pdf, 2004, retrieved: Dec. 2006.
175. Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, E. Lear, *Address Allocation for Private Internets*, Network Working Group: Cisco Systems, Chrysler Corp., RIPE NCC, Silicon Graphics, Inc., RFC 1918, 1996, retrieved: Dec. 2006.
176. Yergeau F. et al., *UTF-8, a transformation format of ISO 10646*, Network Working Group: Alis Technologies, RFC 2279, 1998, retrieved: Dec. 2006.
177. Zezula A., *Web Services Security, A WSSec extension for Systinet WASP*, original title in polish: *Bezpieczeństwo Web Serwisów, Rozszerzenie WSSec dla Systinet WASP*, M.Sc. thesis, AGH-UST, Department of Computer Science, Kraków, 2003.
178. Zieliński K., Jarząb M., Wieczorek D., Bałos K., "JIMS Extensions for Resource Monitoring and Management of Solaris 10", *Computational Science – ICCS 2006*, LNCS 3994, Springer-Verlag, Berlin/Heidelber, 2006, pp. 1039–1046.
179. Zimmermann O., Tomlinson M., Peuser S., *Perspectives on Web Services – Applying SOAP, WSDL and UDDI to Real-World Projects*, Springer, Berlin/Heidelberg, 2003, pp. 131–150.

Appendix A – Examples of JIMS API

```
<SOAP-ENV:Envelope>
  <SOAP-ENV:Header>
    <ns1:connectionId></ns1:connectionId>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <ns1:connect></ns1:connect>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing A.8.1 SOAP message: JMX SOAP connector *connect()* method call.

```
<soapenv:Envelope >
  <soapenv:Body>
    <ns1:connectResponse>
      <connectReturn xsi:type="soapenc:string">soap:0x6</connectReturn>
    </ns1:connectResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing A.8.2 SOAP message: JMX SOAP connector *connect()* method response.

```
<SOAP-ENV:Envelope>
  <SOAP-ENV:Header>
    <ns1:connectionId>soap:0x6</ns1:connectionId>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <ns1:getAttributes>
    <name>
      <canonicalName>Monitoring:class=SystemInformation</canonicalName>
    </name>
    <attributes xsi:type="SOAP-ENC:Array"
                SOAP-ENC:arrayType="xsd:string[4]">
      <item>L1m</item>
      <item>L5m</item>
      <item>L15m</item>
      <item>Mem</item>
    </attributes>
    </ns1:getAttributes>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing A.8.3 SOAP message: JMX SOAP connector *getAttributes()* method call[342].

```
<soapenv:Envelope>
  <soapenv:Body>
    <ns1:getAttributesResponse>
    <getAttributesReturn xsi:type="ns1:AttributeList">
      <item xsi:type="ns1:Attribute">
        <name xsi:type="soapenc:string">L1m</name>
```

[342] In Listing A.8.3, the "SystemInformation" string is the name of the monitoring component, and "Monitoring" is the name of the domain of MBean components that the monitoring component belongs to. JMX domains help group MBean components which are logically connected and can be freely nested in each other. Indeed, "Monitoring:class=SystemInformation" is the fully qualified name of the MBean; i.e. the real name of an MBean is composed of its domain name and a list of valued attributes describing the MBean. More information about JMX naming rules can be found in the JMX specification [147].

```
        <value xsi:type="soapenc:float">1.0</value>
      </item>
      <item xsi:type="ns1:Attribute">
        <name xsi:type="soapenc:string">L5m</name>
        <value xsi:type="soapenc:float">1.0</value>
      </item>
      <item xsi:type="ns1:Attribute">
        <name xsi:type="soapenc:string">L15m</name>
        <value xsi:type="soapenc:float">1.0</value>
      </item>
      <item xsi:type="ns1:Attribute">
        <name xsi:type="soapenc:string">Mem</name>
        <value xsi:type="soapenc:long">79</value>
      </item>
    </getAttributesReturn>
    </ns1:getAttributesResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing A.8.4 SOAP message: JMX SOAP connector *getAttributes()* method response.

```
<SOAP-ENV:Envelope>
  <SOAP-ENV:Header>
    <ns1:connectionId>soap:0x6</ns1:connectionId>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <ns1:close></ns1:close>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing A.8.5 SOAP message: JMX SOAP connector *close()* method call.

```
<soapenv:Envelope>
  <soapenv:Body>
    <ns1:closeResponse/>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing A.8.6 SOAP message: JMX SOAP connector *close()* method response.

```
<soapenv:Envelope>
  <soapenv:Header>
    <ns1:connectionId>soap: 0x4F</ns1:connectionId>
  </soapenv:Header>
  <soapenv:Body>
    <ns2:fetchNotifications>
      <sequence href="#id0" />
      <maxNumber href="#id1" />
      <timeout href="#id2" />
    </ns2:fetchNotifications>
    <multiRef id="id0" xsi:type="xsd:long">1</multiRef>
    <multiRef id="id1" xsi:type="xsd:int">25</multiRef>
    <multiRef id="id2" xsi:type="xsd:long">60000</multiRef>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing A.8.7 SOAP message: JMX SOAP connector notification request[343].

[343] Listing simplified.

```
<soapenv:Envelope>
  <soapenv:Body>
    <ns1:fetchNotificationsResponse>
      <fetchNotificationsReturn>
        <earliestSequenceNumber >0</earliestSequenceNumber>
        <nextSequenceNumber >2</nextSequenceNumber>
        <targetedNotifications xsi:type="soapenc:Array">
          <targetedNotifications xsi:type="ns1:TargetedNotification">
            <notification xsi:type="ns2:WorkerNodeAddedNotification">
              <message/>
              <rmiConnectorInfo xsi:type="ns2:RMIConnectorInfo">
                <heartBeatRetries xsi:type="xsd:int">3</heartBeatRetries>
                <timeOut>false</timeOut>
                <userData xsi:type="ns2:DiscoveryUserData">
                  <OSArchitecture />
                  <OSName />
                  <OSVersion />
                  <description />
    <jmxAddress>service:jmx:rmi://10.0.128.2:7707/stub/rO...4</jmxAddress>
                </userData>
              </rmiConnectorInfo>
              <sequenceNumber xsi:type="xsd:long">34</sequenceNumber>
              <source />
              <timeStamp xsi:type="xsd:long">1166096844571</timeStamp>
              <type />
              <userData xsi:type="xsd:anyType" xsi:nil="true" />
            </notification>
            <listenerID>9</listenerID>
          </targetedNotifications>
        </targetedNotifications>
      </fetchNotificationsReturn>
    </ns1:fetchNotificationsResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing A.8.8 SOAP message: JMX SOAP connector notification response – added new worker node[344].

```
String address = "service:jmx:soap://10.0.128.9:7702/jmxconnector";
try {
      JMXServiceURL url = new JMXServiceURL(address);
      JMXConnector conn = JMXConnectorFactory.connect(url);
      ObjectName emitter = new ObjectName("DefaultDomain:name=GWEmitter");
      connection = conn.getMBeanServerConnection();
      connection.addNotificationListener(emitter, this, null, null);
      conn.close();
} catch (Exception e) {
      e.printStackTrace();
}
```

Listing A.8.9 Notifications via SOAP – subscription to JIMS notifications.

```
public void handleNotification(Notification notif, Object handback) {
  if(handback != null) {
    System.out.println("notification(type="+notif+", hbck="+handback+")");
  } else {
    System.out.println("notification(type="+notif+")");
```

[344] See above note.

```
  }
  long notificationTime = System.currentTimeMillis();
    System.out.println("Notification handled at: "+new Date()+
      "["+notificationTime+"]");
}
```

Listing A.8.10 Notifications via SOAP – handling JIMS notifications.

```
String module1 = "Monitoring:class=SystemInformation";
String module2 = "Monitoring:class=SNMPMirror";
String surl = "service:jmx:rmi://10.0.128.1:7707/stub/rO0AB...AAAB4";
JMXServiceURL url = null;
JMXConnector jmxConnector = null;
MBeanServerConnection connection = null;
try {
      url = new JMXServiceURL(surl);
      jmxConnector = JMXConnectorFactory.connect(url, null);
      connection = jmxConnector.getMBeanServerConnection();
      ObjectName systemInfo = new ObjectName(module1);
      ObjectName snmpMirror = new ObjectName(module2);
      Object result = null;
      result = connection.getAttribute(snmpMirror, "sysName");
      result = connection.getAttribute(systemInfo, "AverageUser");
      jmxConnector.close();
} catch (Exception e) {
      e.printStackTrace();
}
```

Listing A.8.11 Direct access to JIMS using RMI[345].

```
ObjectName timerObjName = new ObjectName(MBeanDomain + ":class=Timer");
Timer myTimer = new Timer();
this.server.registerMBean(myTimer, timerObjName);
```

Listing A.8.12 Registration of the JIMS JMX Timer.

[345] Listing simplified.

Appendix B – Selected JIMS Modules and Parameters

Table B.8.1 Selected JIMS SystemInformation module parameters

Name	Type	Ex. Value	Description
AverageNcpuIdle	long[]	n/a	Average Idle time for each CPU[346]
AverageNcpuIowait	long[]	n/a	Average IO wait time for each CPU[347]
AverageNcpuIrq	long[]	n/a	Average IRQ time for each CPU
AverageNcpuNice	long[]	n/a	Average nice time for each CPU
AverageNcpuSoftirq	long[]	n/a	Average soft IRQ time for each CPU
AverageNcpuSystem	long[]	n/a	Average system time for each CPU
AverageNcpuUser	long[]	n/a	Average user time for each CPU
Cline	String	n/a	Command line for kernel execution
Cpuinf	String	n/a	Information about CPUs
FileSystemStatistics	String[]	n/a	Filesystem statistics
L15m	float	0.96	CPU load during the last 15 minutes
L5m	float	1.0	CPU load during the last 5 minutes
L1m	float	1.0	CPU load during the last 1 minute
Maxmem	long	1006	Maximum memory available in [MB]
Maxswp	long	517	Maximum swap available in [MB]
Mem	long	12	Physical free memory [MB]
Membuf	long	78	Memory buffers [MB]
Memch	long	765	Memory cache [MB]
Memsh	long	0	Amount of shared memory [MB]
Model	String	n/a	Model of CPU
Ncpus	long	2	Number of CPUs
Ndisks	long	1	Number of disks
Nproc	long	89	Number of processes
Swp	long	492	Amount of swap used [MB]
TimerPeriod	Int	2	Timer period used for average statistics
Type	String	n/a	Type of CPU
Uptime	Float	629740.06	System uptime
Ver	String	n/a	Version of the operating system

[346] Time spent in the idle mode by each CPU. An average is calculated, based on the time defined in the TimerPeriod attribute (see Linux kernel API documentation in [35]).

[347] See above note. All parameters from the "Ncpu" group share a similar definition.

Table B.8.2 Selected JIMS SNMP module parameters.

Attribute	Type	Ex. Value	Description
hostIp	String	127.0.0.1	IP host address
icmpInEchoReps	Long	7	Number of incoming ICMP echo replies
icmpInEchos	Long	10535	Number of incoming ICMP echo requests
icmpInErrors	Long	0	Number of incoming ICMP errors
icmpInMsgs	Long	12475	Number of incoming ICMP messages
icmpOutEchoReps	Long	10535	Number of outgoing ICMP echo replies
icmpOutEchos	Long	0	Number of outgoing ICMP echo requests
icmpOutErrors	Long	0	Number of outgoing ICMP errors
icmpOutMsgs	Long	11923	Number of outgoing ICMP messages
ifAdminStatus	Integer[]	n/a	Interfaces' administrative status
ifDescr	String[]	n/a	Interface description
ifInDiscards	Long[]	n/a	Incoming frames discarded by the interface
ifInErrors	Long[]	n/a	Errors in incoming frames
ifInOctets	Long[]	n/a	Number of octets incoming
ifIndex	Integer[]	n/a	Interfaces' indexes
ifMtu	Integer[]	n/a	Interfaces' MTUs
ifNumber	Integer	2	Number of interfaces
ifPhysAddress	String[]	n/a	Physical addresses of interfaces
ifSpeed	Long[]	n/a	Speed of interfaces
ifType	Integer[]	n/a	Types of interfaces
ipDefaultTTL	Integer	64	Default TTL value for IP packets
ipRouteTable	String [][]	n/a	IP route table
portNumber	Integer	7705	SNMP agent port number
snmpInPkts	Long	75	Incoming SNMP packets
snmpOutPkts	Long	61	Outgoing SNMP packets
sysContact	String	n/a	SNMP system contact person
sysDescr	String	n/a	SNMP system description
sysLocation	String	AGH	SNMP system location
sysName	String	access	SNMP system name
sysUpTime	Long	63027852	SNMP system uptime, in 10^{-2}[s]
tcpConnTable	String [][]	n/a	TCP connection table
tcpMaxConn	Integer	10	Maximum number of TCP connections
tcpOutSegs	Long	23035415	Outgoing TCP segments
udpInDatagrams	Long	11543839	Incoming UDP datagrams
udpOutDatagrams	Long	11770616	Outgoing UDP datagrams
udpTable	String [][]	n/a	UDP table

Table B.8.3 Selected JIMS NetworkMetrics module parameters and methods

Name	**Type**	**Ex. Value**	**Description**
Attempts	int	1000	Default number of retries
UDPTimeout	int	2000	Measurement timeout [ms]
UdpReceiveDataFieldSize	int	1472	Def-lt rcvd datagram size
UdpSendDataFieldSize	int	1472	Def-lt sent datagram size
double measureThroughput			
	(String)p1	149.156.9.15	Destination IP address
	(int)p2	10000	Number of retries
	(int)p3	1472	Sent UDP datagram size
	(int)p4	1472	Rcvd UDP datagram size
double measureThroughput			
	(String)p1	149.156.9.15	Destination IP address
double measureUDPLatency			
	(String)p1	149.156.9.15	Destination IP address
	(int)p2	10000	Number of retries
	(int)p3	1472	Sent UDP datagram size
	(int)p4	1472	Rcvd UDP datagram size
double measureUDPLatency			
	(String)p1	149.156.9.15	Destination IP address
double measureICMPLatency			
	(String)p1	149.156.9.15	Destination IP address

Appendix C – JIMS Availability and Documentation

1. Bałos K., *JIMS - the JMX-based Infrastructure Monitoring System*, User Guide, available at: http://www.ics.agh.edu.pl/jims, AGH-UST, Department of Computer Science, Kraków, 2006, visited: Dec. 2006.
2. CrossGrid - CG WP4.4 Verification and Quality Control team, *Middleware test results, status and results of the integration and validation requests*, available at: http://www.lip.pt/computing/projects/crossgrid/task4/test-results.htm, 2006, visited: Dec. 2006.
3. CrossGrid, *Repository - directory - sources: crossgrid/crossgrid/wp3/wp3_3-moninfr/wp3_3_3-jims*, JIMS CrossGrid CVS repository at Karlsruhe, available at: http://savannah.fzk.de/cgi-bin/viewcvs.cgi/crossgrid/crossgrid/wp3/wp3_3-moninfr/wp3_3_3-jims, 2006, visited: Dec. 2006.
4. CrossGrid, *Index of /distribution/crossgrid/autobuilt/i386-rh7.3-gcc3.2.2/wp3/SOURCES*, JIMS source packages in autobuild repository at Karlsruhe, available at: http://savannah.fzk.de/distribution/crossgrid/autobuilt/i386-rh7.3-gcc3.2.2/wp3/SOURCES/, 2006, visited: Dec. 2006.
5. CrossGrid, *Index of /distribution/crossgrid/autobuilt/i386-rh7.3-gcc3.2.2/wp3/RPMS*, JIMS Linux RedHat 7.3/IA32 RPM binary packages repository, available at: http://savannah.fzk.de/distribution/crossgrid/autobuilt/i386-rh7.3-gcc3.2.2/wp3/RPMS/, 2006, visited: Dec. 2006.
6. CrossGrid, *Index of /distribution/crossgrid/autobuilt/i386-rh7.3-gcc3.2.2/wp3/SRPMS*, JIMS Linux RedHat 7.3/IA32 RPM source packages repository, available at: http://savannah.fzk.de/distribution/crossgrid/autobuilt/i386-rh7.3-gcc3.2.2/wp3/SRPMS/, 2006, visited: Dec. 2006.