An Adaptive and Dependable Middleware for Enhancing the Reliability of Distributed Computations in Dynamic Grid Environments

eingereicht von

André Luckow, Msc.

vorgelegt der

Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

zur Erlangung des Akademischen Grades

Doktor der Naturwissenschaften

– Dr. rer. nat –

angefertigt am

Institut für Informatik der Universität Potsdam
Professur Betriebssysteme und Verteilte Systeme

Gutachter:
Professor Dr. Bettina Schnor
Professor Dr. Shantenu Jha
Professor Dr. Glenn Luecke

August 21, 2008

To my wife, Nicole.

"There are two mistakes one can make along the road to truth – not going all the way, and not starting." Buddha

Preface

Grids are a newly emerging infrastructures, which enable the sharing of resources, such as compute time, storage, and instruments across institutional boundaries. The vision of Grid computing is the creation of a distributed, integrated systems, which provides those resource to applications in a transparent manner. Grid computing has witnessed a phenomenal growth and has started to become mainstream in the last years with commercial offering from Sun, Google or Amazon. This offerings are in general restricted to proprietary APIs and do not offer services across organizations yet. To enable true inter-organization grids open standards and more sophisticated grid middleware platforms are required.

With the increasing deployments and dependency toward grid infrastructures, fault tolerance of grids has already become a major concern.

> "A distributed system is a system on which I cannot get any work done, because some machine I have never heard of has crashed." (Leslie Lamport)

In distributed systems failures are rather the rule than an exception. The more components involved the more error-prone the system gets. Even with more reliable individual components the overall fault rate can be a major problem when thousands of machines are connected together in a grid. Current grid platforms do not address sufficiently is fault tolerance. Aim of the Migol projects, which provided the majore foundation for this work, is the development of a failure tolerant infrastructure for grid applications.

Such a work cannot be conducted without the help of many different persons: In addition to the authors cited in this work, I would like to express my gratitude to all people who made this work possible. First I would like to express my appreciation to Professor Bettina Schnor for her excellent mentoring as well as for her great ideas and suggestions, which significantly shaped this thesis. Her observations and comments helped me to create an overall direction for this work and to move forward with my research.

Also I would like to thank the entire operating and distributed system group for the useful and constructive discussions about this thesis: Lars Schneidenbach, Stefan Liske, Feng Liang, and Thomas Scheffler. I also like to acknowledge Professor Ed Seidel, Gabrielle Allen and Daniel Katz for supporting my visit at the Center for Computation & Technology of the Louisiana State University. The time there has been a great inspiration to me. Further, I would like to thank Shantenu Jha, Hartmut Kaiser, André Merzky, Joohyun Kim, Ole Weider for supporting my work with SAGA. Erik Schnetter and Peter Diener helped me out with my Cactus-related questions. Also, I would like to thank Steve Brandt with whom I had great discussions.

Reviewing a thesis is hard, but also a vitally important work. I am especially grateful to my wife Nicole and Tobias Hutzler for extensively reviewing the manuscript and providing many great improvement ideas. This work is very likely not perfect. Thus, further comments and suggestions are very welcome.

André Luckow
Potsdam, August 21, 2008

Contents

List of Figures

List of Tables

Listings

"Discovery is when something wonderful that you didn't know existed, or didn't know how to ask for, finds you."

Jeffrey O'Brien

1 Introduction

The demand for computing power grows more and more beyond the limits of Moore's law. Despite the exponential increase of the computing power, the solving of challenging scientific problems demands even more resources [247, 124]. Grid computing envisions the sharing of computing resources as well as storage, network and software resources across multi-institutional virtual organizations (VOs) to provide more effective solutions to important scientific, engineering and business problems [130]. Examples for demanding scientific problems are severe weather simulations [18] or astrophysical problems, such as the gamma-ray burst [247].

Distributed applications must be able to orchestrate thousands of heterogeneous resources in a complex, heterogeneous and dynamic environment. Writing such applications is a complex task for various reasons; not least because such environments are inherently prone to failures and thus very unreliable. Nodes can shut down respectively come up again at any time. The same holds for network connections. Thus, fault tolerance is a major concern, in particular for long-running applications [238, 150]. Running such applications in a failure-prone environment requires a sophisticated middleware.

Different studies show that in large computational Grids resources become unavailable at a high rate: In the Grid 5000 it was found that the mean time to failure for nodes is 12 minutes at grid-level, 5 hours at cluster-level and 2 days on node-level [170]. An analysis of high performance systems at the Los Alamos National Laboratory conducted by Schroeder/Gibson also found surprisingly high failure rates [287]. Several other studies, which investigate the failure rates of Grid jobs, showed similar severe results: Depending on the scenario 10-40 % of all submitted jobs fail [206, 187]. Although these analyses have been conducted in different environments with different hardware and software configurations and measurement methods, it is obvious that reliability is not an intrinsic property of Grid environments. Faults are rather the rule than an exception. Efficient support for fault tolerance is therefore a key requirement for Grid infrastructures.

After a brief introduction into the field of Grid Computing, this chapter will give an overview of the motivation, objectives and structure of this work.

1.1 Grid Computing: Introduction and Definition

Inspired by the electrical Power Grid, Grid computing is concerned with the provisioning of an infrastructure, which allows access to a variety of distributed, networked resources such as compute power, memory, storage and network capacity. While the power Grid enables the transparent, pervasive and reliable access to electrical power, Grids aim to transparently share distributed IT resources via high bandwidth networks [78]. High speed, optical networks make the location of resources irrelevant just as alternate current networks allowed the decoupling of energy consumer and producer [68].

The term Grid computing has been coined by Foster/Kesselman/Tuecke and is defined as flexible, secure, and coordinated resource sharing among dynamic collections of individuals, institutions and resources referred to as virtual organizations [130]. A Grid is characterized by the following aspects:

- **Heterogeneity:** Grids comprise of a large number of diverse resources with different capabilities and operating environments. Heterogeneity arises from different hardware platforms, operating systems, protocols, and service semantics [317].
- **Dynamism:** Grid environments are constantly subject to changes: resources can suddenly fail or new resources can become available.
- **No complete knowledge:** Due to the distributed nature no global and complete knowledge of all Grid resources can be assumed. Grid applications must be able to deal with this uncertainty [250].
- **No central control:** A Grid consists of dynamically evolving *virtual organizations* [130]. Resources are managed by their owning organization. In general, local policies determine who is allowed to access which parts of a resource.

Grids provide organizations and scientists access to an immense amount of resources and capabilities. By doing so, Grids more and more change the way how scientists and engineers approach complex computational and data intensive problems. Solving large-scale scientific problems on infrastructures as Grids is also termed *e-Science* [303, 124].

Grid applications must be able to orchestrate heterogeneous resources in a complex and dynamic environment. To achieve this, Grid applications rely on a Grid middleware to discover and allocate resources or to manage data and file transfers. In general, a Grid middleware comprises of several distributed services for provisioning and supporting the remote access to compute resources and data. Several more or less lightweight Grid platforms, such as the Globus Toolkit [123], UNICORE [271] and gLite [65] have been developed.

A major building block for interoperable virtual organizations are open standards. The *Open Grid Service Architecture (OGSA)* [128], which is currently specified within the Open Grid Forum (OGF) [241], aims to provide a blueprint for a service-oriented Grid environment. In this model the capabilities of resources are accessed by users through so called *services*. OGSA heavily relies on Web service standards defined by the W3C [13] and OASIS [11] consortium as framework for the specification of Grid services. Within the overall architectural OGSA framework several sub-standards and profiles dedicated to certain aspects of Grid computing are currently under development.

However, despite the ongoing efforts with respect to middleware and standards development, fault tolerance in a dynamic and unreliable Grid environment still is a great challenge. To utilize the full potential of Grids the dependability of Grid infrastructures must be improved substantially, which leads to the main motivation of this thesis.

1.2 Motivation

Large scale, multi-institutional Grids are inherently unreliable due to many reasons: Grids are geographically distributed and they involve many heterogeneous components as well as multiple organizations, which make the overall structure very complex. With the current trend towards larger Grids and clusters, the mean time to failure (MTTF) will further decrease with every additional component. It is projected that the number of cores per processor will double several times over the next decade [33]. Further, supercomputers will use more processors to increase their performance. Many studies indicate that the failure rate grows proportionally to the number of components, which will lead to a rapid increase of the failure rate in the near future [288].

During long application runs high failure rates are a major concern. Many scientific applications require to run for days or weeks. For example, a meaningful simulation of gamma ray bursts, an astrophysical phenomena, requires over 100 days of runtime on a 1 PFlop/s machine [247]. The larger the systems, the more likely is the failure of single component and thus the lower is the mean time to failure. A study of Gibson/Schroeder [288] found failure rates of up to 1100 failures per year when investigating high performance systems at the Los Alamos National Laboratory (LANL) between 1995 and 2005. This means that an application that runs on all nodes of such an error-prone machine is forced to recover more than twice a day. Efficient support for fault tolerance in long-running applications is therefore essential.

The vision of autonomic computing [186] is to provide a system that is able to manage itself according to the goals of the user and/or administrator. An important property of autonomic computing is self-healing: autonomic systems are to a certain extent able to detect and resolve faults without human involvement. These principles can also be applied to the management of Grid infrastructures and applications. For example, a long-running e-Science application significantly benefits from the ability to automatically detect and resolve failures. Currently, ensuring the successful completion of an application is mainly the responsibility of the user, who has to manually monitor and recover tasks (e. g. using a portal). Without human interaction, recovery times are minimized, while the application and infrastructure utilization is enhanced.

Achieving fault tolerance in Grids consisting of multiple layers of hardware and software is a great challenge. Most user-level application and even middleware services assume that the lower level system is reliable. For example, the current message passing standard MPI 2, which is heavily used in parallel applications, does not consider fault tolerance [115] sufficiently leaving details mainly to implementations. Unfortunately, current MPI implementations such as MPICH2 [233] and OpenMPI [133] do not handle faults very gracefully [154].

While ensuring the reliability of Grid applications is important, the fault tolerance of the Grid infrastructure itself is an essential pre-requisite for achieving this goal. Grid applications largely depend on infrastructure services, e. g. for gathering resource information and for allocating resources. It must be ensured that a failure of a single infrastructure service has the smallest impact as possible. Guaranteeing the availability and reliability of a service infrastructure is a complicated task. For example, different sophisticated IT infrastructures, such as the London Stock Exchange [208] and the Amazon Cloud services [69] failed recently. These events emphasize how vulnerable distributed infrastructures are.

As emphasized, in Grids failures are rather the rule then the exception. Hence, Grid systems must be able to cope with failures on all levels. The capability of a system to continue its operation despite the presence of failures is defined as fault tolerance. This thesis addresses the fault tolerance of Grids from an overall system-wide perspective, covering Grid applications as well as infrastructure services.

1.3 Thesis Objectives and Contributions

As described, with the increasing size of clusters and Grids as well as with the advancing penetration of Grid infrastructures in sciences and in industries, issues like fault-tolerance and self-healing are becoming tremendously important. This thesis aims to address the fault tolerance of long-running scientific applications as found in many sciences, e. g. in astrophysics or life sciences. However, fault tolerance is always associated with overhead. Often, fault tolerance and performance are conflicting goals. Thus, the design of the overall system must carefully trade-off fault tolerance versus usability, costs and performance.

The *Migol* middleware, which is proposed by this thesis, addresses fault tolerance in failure-prone Grid environments. The name Migol abbreviates "Migration in the Grid OGSA lite" symbolizing that the framework provides a lightweight layer on top of existing OGSA services to support fault tolerance. Migol continues the work of Lanfermann [200], who developed a framework to support the nomadic migration of Cactus applications. The mission of Migol is to transfer these concepts to a service-oriented Grid environment supporting a wide range of applications. This requires the extraction and enhancement of the fundamental concepts introduced by Lanfermann to meet the requirements of a service-oriented Grid. Migol aims to provide a fault tolerant, self-adaptive Grid middleware usable for any application that requires recoverability and fault tolerance.

Migol aims to support applications in performing complex tasks such as resource allocation, monitoring, checkpointing and migrating. To build such a Grid infrastructure, a number of important research challenges must be addressed. This thesis makes the following contributions:

- **Requirement Collection:** This work identifies different application scenarios. Based on these scenarios, application requirements with a particular focus on reliability and the acceptable trade-off for achieving this goal have been collected. Further, these scenarios are used to derive reliability requirements related to the Grid infrastructure itself.

- **Fault Tolerance Concepts:** Important concept, such as the system and fault model, have been adapted from classical distributed computing research to the Grid computing use case. This also includes known fault tolerance techniques, e. g. the Totem [25] protocol, which has been successfully implemented on top of a Grid architecture.

- **Self-Healing Infrastructure Supporting the Automatic Recovery of Grid Applications:** The main contribution of this thesis is the Migol framework, which provides a reliable infrastructure for managing arbitrary applications in the Grid. The framework handles the complete application lifecycle, from deployment to monitoring of the application run. If Migol detects a failure, applications are automatically recovered.

- **Reliable Grid Architecture:** The Open Grid Service Architecture (OGSA) [128, 131] defines a blueprint for a Grid architecture. OGSA envisions the Grid as reliable and scalable distributed environment. However, details are mainly left to the implementation. This thesis addresses Grid fault tolerance from an overall architectural perspective. To achieve this goal, all services and applications are carefully evaluated with respect to their criticality. As result of this classification appropriate fault tolerance techniques are selected and deployed.

- **Service Replication:** To ensure the high availability of certain critical services, an active replication protocol, especially suited for Grids, has been developed. This Totem-based protocol is used to ensure the continuous operation of the AIS.

- **Development:** The feasibility of this architecture has been proven with an implementation based on the Globus Toolkit 4 [123] middleware.

- **Evaluation:** The Migol system has been evaluated with different real applications in several production Grid environments. The results show that the approach presented is well usable in real life scenarios.

- **Fault Tolerant Design Pattern:** The core patterns for achieving a reliable infrastructure such as Migol have been collected. These principle can also be applied to other reliable systems.

Theses results are described in the following chapters.

1.4 Structure of this Thesis

This thesis is structured as follows: After an overview about important fault tolerance concepts in chapter 2, the current state of Grid computing is explored in chapter 3.

Chapter 4 surveys common distributed programming paradigms. Based on these application categories, relevant use cases and their requirements with respect to a Grid infrastructure are derived.

The major part of Grid applications are MPI based [231]. Message passing has become the standard for developing parallel cluster applications. Especially for application running on thousands of nodes fault tolerance is an important aspect. In chapter 5 the self-healing and fault tolerance capabilities found in the MPI standard and its implementations are discussed.

The core of this thesis, the Migol middleware, is presented in chapters 6 to 9. Chapter 6 will discuss the functionality of all services as well as different aspects regarding application integration. As described, the fault tolerance of the Migol framework itself is an important requirement, which is extensively described in chapter 7. An important contribution of this thesis is the service replication framework for highly critical Grid services, which is also presented in this part. Chapter 8 discusses selected implementation details. Further, performance results and experiences with our Grid framework in a real Grid environment are presented in chapter 9.

Related work and technologies are reviewed in chapter 10. This thesis concludes with an analysis and assessment of the achieved results and an overview about future work in chapter 11.

"Every thing fails, all the time"

Werner Vogels

2 Fault Tolerance Concepts

Fault tolerance is a desirable property of any large system. This chapter will highlight fundamental issues concerning distributed systems and fault tolerance. The term fault tolerance is commonly defined as follows:

> "Fault tolerance is the ability of a system or component to continue normal operation despite the presence of hardware or software faults [306]."

The goal of a fault tolerant system is to continue its operation even in case of certain operational failures. The idea is to hide the consequences of programming bugs, hardware failures etc. from the user as much as possible. In a Grid, resources are per definition unreliable. It is therefore essential to understand the impact of faults on the system.

Fault tolerance is often seen in the greater context of *dependable computing*, which is defined as the ability of a computing system to deliver a trustable, reliable service [203]. The dependability of a system is described by the following properties:

- *Availability* is defined as the probability that a system is working at a given time (readiness).
- *Reliability* is specified as the probability that a system does not fail until a given time (continuity).
- *Safety* refers to the property of a system that in case of a failure it is ensured that nothing catastrophic happens.
- The *maintainability* of a system determines how easy a failed system can be repaired.

There is always a tradeoff between the different dependability properties. Thus, the design of a fault tolerant system is a complex challenge.

This chapter is denoted to the definition of the used terms and concepts, which aid the specification and design of a fault tolerant system. After an introduction of the terms availability, reliability, fault, error and failure in section 2.1 and 2.2, a classification of fault causes is presented in section 2.3. A system and fault model is the pre-requisite for the selection of appropriate fault tolerance techniques. Section 2.4 presents the system model used in this thesis. In section 2.5 the basic building blocks and design principle for creating reliable distributed system are presented. The following sections are dedicated to different fault tolerance techniques: Section 2.6 describes techniques for the checkpointing of applications, while section 2.7 discusses process replication in detail. This chapter is concluded with a definition of fault tolerance levels, which provide the basis for a good system design.

2.1 Reliability and Availability

The most significant measures for fault tolerance are *reliability* and *availability*. Reliability is often quantified using the *mean time to failure (MTTF)* as parameter. The MTTF describes the average lifetime of a system or component, i. e. the average time from start of operation until its failure. In the following an exponential distribution of the system's lifetime is assumed. The inverse of the MTTF then corresponds to the failure rate $\lambda = \frac{1}{MTTF}$, i. e. the average rate at which a system fails. The reliability of a system at a time t can be expressed as follows:

$$R(t) = e^{-t\lambda}$$

However, in real life systems the failure rate can vary across the lifetime of a system. For example, during its early life and end of life, components are more likely to fail. Nevertheless, for the largest part of a components lifetime, the middle age, a constant failure rate is usually a valid assumption.

The reliability of a system or component is a useful parameter when designing fault tolerant systems. For example, the MTBF can be used to determine the number of redundant subsystems necessary to ensure the reliability of the overall system.

The availability of a system is the fraction of time that a system is able to perform its specified function (readiness for usage). The availability (α) can be expressed using the mean time to failure (MTTF) and *mean time to repair (MTTR)*:

$$\alpha = \frac{MTTF}{MTTF + MTTR}$$

Based on these definitions, a Grid resource, which constantly shows errors, but then automatically reboots, is not very reliable, but available. A service, which is actively replicated on multiple trusted nodes can be considered highly available and highly reliable. Since most Grid resources are not under control of the user they must be considered unreliable and potentially not available. For example, an analysis of the German Astrogrid in May 2008 showed that despite a valid user certificate 10 % of the systems marked as active in the WS-MDS were not accessible. These systems are from the user point of view not available. Reasons for this unavailability are possible hardware faults, misconfigured firewalls, Globus installations or errors during the propagation of the user accounts from the virtual organization management service.

Since a Grid comprises of many resources the modelling of the overall reliability and availability is a complex task. To calculate e. g. the reliability and availability of a Grid service all dependent components and lower level services must be considered. In general, it can be noted that the larger the number of components the potentially more unreliable und unavailable the system becomes.

2.2 Fault, Error and Failure

To aid the debugging of failures, a common vocabulary is required. It is important to differentiate between the cause of a failure and the error symptom. The definitions for fault, error, and failure proposed by Laprie [203] are widely accepted:

- A *failure* occurs when a service does not comply with its specification.
- An *error* is that part of the system state that leads to the failure.
- The cause of the error is defined as *fault*.

The activation of a fault in a certain software module in general does not cause an immediate failure. Rather it causes an erroneous internal condition – an error occurs. An error can lead to further errors in the same or another component of the system before the actual failure of the software system occurs. The process of passing an error through multiple subsystems is also called error propagation. In complex environments different faults can cause the same failure, but some faults may never cause a failure and remain latend.

Large scale distributed systems, such as Grids, consist of many dependent services. The larger and the more complex a system, the more error-prone it gets. Larger systems have a longer error propagation chain, which increases the delay between the occurrence of the fault and the failure. Thus, locating and handling of errors can be very difficult. To detect and handle failures of dependent services successfully a defined failure behaviour (e. g. fail-stop) is required. If a higher level service can provide a service despite a failure of a lower level service using some kind of fault tolerance technique (see section 2.5), it *masks* the failure [90].

2.3 Fault Classifications

Faults can be categorized with respect to different dimensions (see for example [36, 37]). Commonly, faults are classified with respect to their duration: transient faults occur once and then disappear, while permanent faults remain in the system until they are removed. The selection of an appropriate fault tolerance technique depends to a great extend on the characteristic of the fault. For example, transient faults can be handled by a simple re-try.

Further, faults can be classified with respect to their causes and characteristics [144]:

- *Hardware* faults, such as the failure of a CPU, memory or controller, are generally caused by physical reasons. Due to their physical nature, hardware faults in different physical systems can be considered as independent.
- *Software* faults are caused by the software layer. Due to the complex states machines maintained in software, software faults in general occur more often than hardware faults. Although software faults are permanent, most of them show a transient behaviour.
- *Storage*: The most common storage media are harddisk drives. Although harddisks are very reliable, even low failure rates can have a big impact if critical information is stored on disk. Different studies e. g. at Google [258], indicated disk failure rate of 2-10 % per annum.
- *Network*: The communication system is the most unreliable part of a distributed system[147]. However, data center are usually connected by redundant paths, i. e. most faults can then be assumed to be transient.
- *Human* faults are caused by users or system administrators during the use or maintenance of the systems.

While hardware faults are mostly caused by physical degradation and can simply be handle by replacing the component with a new physical copy, software faults usually have more complex causes. According to the investigations of Gray in the Tandem system, software faults represent the most common failure cause [145]. Similar results have been found in other studies [244, 80].

Software faults are per definition permanent. However, many software faults show a transient effect, i. e. such a fault is likely to disappear during a retry. A reason for this behaviour is that such errors are often caused by rare corner cases, e. g. a transient device fault, a counter overflow or a race condition [205].

To describe the different behaviour of software bugs, Gray [144] coined the terms Bohrbugs and Heisenbugs:

- *Bohrbugs* are faults that occur consistently under well-defined conditions.
- *Heisenbugs/Mandelbugs* are non-deterministic with respect to their occurrence and non-occurrence[327, 151].

While Bohrbugs are easy to reproduce, Heisenbugs/Mandelbugs are difficult to detect and resolve. Assuming that Bohrbugs, such as design faults, can usually be resolved during the software development process, the number of these bugs in an operative system should be very low. Grays investigation confirmed this thesis, showing that in the Tandem system about 40 % of the software faults could be successfully masked [145].

However, it must be noted that real world failures can arbitrary manifold (see Kola et. al. [191] for examples). In particular, data corruption can cause very hard to detect failures leading to misbehaving machines.

During the design of a fault tolerant system, possible faults that can lead to failures must be identified. The system and fault model defines which errors a system can detect and handle.

2.4 System and Failure Models

To handle possible failures in a system, a precise specification of the used system and fault model is required. While the system model defines the abstraction, which is used for describing the distributed system, the fault model determines which possible effects of faults on the behaviour of a system are considered. This model will then be used to derive actions that will be taken after occurrence of an error.

A distributed system is assumed to consist of a number of nodes (processors) and a communication network. Nodes can only communicate via messages. Each node has access to a stable storage as well as a hardware clock. The stable storage survives failures [90].

No special configuration of the communication network is required – it is solely assumed that the communication system provides point-to-point connections between all nodes. Multicasts or broadcasts must be emulated using this primitive. Unless otherwise noted, the communication network is assumed to be reliable.

The timing assumptions made by the distributed system model are another important aspect. Common classifications differentiate between the synchronous and the asynchronous distributed system model [91]. Synchronous systems assume an upper time bound for events, such as a message transmission or an execution of a step on a processor. In the asynchronous model no timing assumptions are made. However, the asynchronous model has severe limitations – Fischer/Lynch/Paterson showed that it is not possible to solve consensus deterministically in the pure asynchronous model if one process fails. Consensus protocols are an important building block for process replication (see section 2.7). Thus, a more realistic model the partial asynchronous system model is assumed. In this model an upper bound for message transmission and execution times exists, but they are not known in advance [111]. This model is not as restrictive as the asynchronous model and allows the usage of a timeout schema for error detection.

Having defined a system model, the failure behaviour or failure semantic that a system exhibits is particularly important for selecting an appropriate recovery schema. Commonly the following classification is used to describe the failure behaviour of remote systems [90, 302]:

- After a *crash failure*, a server stops working in such a way that the failure can be detected by other servers [280]. That means that the server does not receive or respond to messages.

- An *omission failure* occurs if a server fails to receive or send a request (receive omission respectively send omission).

- A *timing failure* is caused if a process responds too late or too early to a request.

- A *Byzantine failure* [254] causes the system to behave in an arbitrary way. This model permits processes to produce incorrect output, e. g. by intentionally sending corrupted data to other processes.

In this work, a crash fault behaviour is assumed. In general, nodes can recover after a fault. This model covers common faults such as transient hardware faults, a power outage or a transient fault due to a software bug. Crash errors can be accurately detected by a remote peer using a timeout.

Omission faults can easily occur e. g. due to the restart of a system or a re-configuration. Such faults are considered by Migol and can be handled e. g. with a simple retry mechanism in conjunction with a sequence number mechanism to detect the message duplicates.

2.5 Fault Tolerance: Surviving Faults in a Distributed System

Fault prevention [205] (also termed fault intolerance [35]) is the process of minimizing faults during the development of a system prior to its operational phase. Common techniques for fault prevention are design verification and testing. Due to the complex nature of Grids consisting of thousands of concurrently interacting nodes, no amount of verification and testing can cover all faults in a system and give a complete confidence in the reliability and availability of a system. Thus, fault tolerance techniques are indispensable for building a resilient Grid.

To ensure the fault tolerance of a system, all possible faults within the system must be considered in the design. The key factors for achieving fault tolerance are:

- Modularization into independent subsystems that show a well-defined failure behaviour.
- Deployment of efficient error detection mechanisms.
- Usage of fault tolerance techniques to handle errors.

In the following these three aspects are discussed in detail.

2.5.1 Modularization

Modularization, i. e. the hierarchically decompositioning of a system into self-contained, independent modules is a key for achieving fault tolerance [144, 263]. Each module should provide a well-defined service which is able to operate independent of other services. A good separation of concerns avoids faults and eases the process of fault location and handling. Ideally, these modules should expose a fail-stop failure semantic, to ensure that failures can be detected rapidly. Dependent modules must be aware of possible submodule failures. Loosely-couple architectures as envisioned by the service-oriented architecture paradigm (SOA) with minimal knowledge about other services are well suited for fault tolerant computing (see section 3.1.1).

An important aspect of modularization is the management of the state [336]. It is often assumed that modules or services are stateless. Statelessness is especially from the fault tolerance point of view desirable: clients can go to an arbitrary other service with the same capability in case of a failure. However, not all services can be kept stateless. Especially in Grids, services need to maintain short-lived as well as permanent resource states. Common practice is to separate services into stateless and stateful parts. For example, the Web Service Resource Framework (WSRF) [93] standard uses stateless service and encapsulates the state into WS Resources (see section 3.1.1). Ensuring the fault tolerance of the stateful resources requires special precaution, which will be discussed later.

2.5.2 Failure Detection

Failure detection in distributed systems consisting of multiple layers from hardware to software is difficult. Error detectors are a central building block for a fault tolerant system. Failure detectors are generally judged with respect to completeness, accuracy, and efficiency [153]. A rapid failure detection is the prerequisite for a fast repair.

A common way to detect errors are heartbeat protocols, i. e. the sending of a ping message from a system monitor to the monitored node respectively process [158]. However, there are certain limitations with this approach – with timeouts it is not possible to distinguished between:

- the crash of a remote process,
- the slow-down of a remote process,
- the slow transmission of a message, and
- a lost message during a transmission.

That means that a couple of lost messages can be easily mistaken as a failure. Thus, many detectors assume that the network layer is reliable.

Despite the inherently unreliability of timeout-based detectors most systems rely on a variant of the heartbeat protocol [158] for failure detection. In these systems the management of timeouts is a critical issue: While short timeout thresholds are desirable to reduce the detection latency and thus the mean time to repair (MTTR), the probability of false detections increases. But, the longer the timeout, the larger is the window in which the error can propagate to another subsystem. Thus, the timeout threshold must be carefully chosen trading off accurate and timely error detection. Another aspect is the overhead associated with heartbeats: too many monitoring messages can saturate the communication system [75].

Having detected a failure, several options for resolving it exist. The following section surveys common principles for tolerating failures.

2.5.3 Fault Tolerance Overview

In general, fault tolerance is based on providing useful redundancy. There are two forms of redundancy: temporal/time and spatial redundancy. While temporal redundancy usually involves the re-execution of an application, spatial redundancy deploys additional copies of processes or resources [36, 97].

Many techniques for enhancing the reliability and availability of distributed systems have been proposed. The selection of an appropriate mechanism clearly depends on the requirements of the applications as well as the associated costs for deploying redundant resources. A commonly used mechanism is the repeated execution of a program, i. e. the exploration of time redundancy. Often re-execution is combined with checkpointing, i. e. the saving of the application state on a stable storage.

A checkpoint represents a consistent snapshot of an application's state saved. Using this intermediate state the time amount necessary for re-execution of the computation can be significantly reduced [113]. Different techniques for creating checkpoints exists – an overview is given in section 2.6.

To ensure that an updated checkpoint is available, checkpoints must be periodically written during the failure free execution. Applications can then be restarted from a checkpoint on the same or on another Grid resource. The transferring of the latest checkpoint to another host and the resuming of the computation from this checkpoint is also termed to as job migration. Checkpoint/restart is also referred to as backward recovery [173]. Commonly this technique is used in Grids to tolerate faults in long-running applications and Grid services.

The performance of checkpointing depends to a great extent on the frequency of the checkpointing. The larger the checkpoint intervall, the larger the possible recovery time. The more frequently checkpoints are written the larger the overhead. Different models for optimizing the checkpointing interval with respect to the overall availability have been proposed [71].

Another form of fault tolerance is the redundant execution of jobs on different resources. In most cases this approach is not reasonable since at least twice the required resources must be consumed. Especially, for compute-intensive, long-running applications this is no acceptable. For these applications checkpoint/restart recovery is clearly preferable.

However, for certain applications, e. g. some critical Grid services, process redundancy is very useful to ensure the high availability. A critical aspect when maintaining multiple application processes, is the coordination of these replicas (also referred to as processes). In particular it must be ensured that all replicas have the same consistent state. In general, this is achieved by using a reliable broadcast primitive, which ensures the ordering of update messages. Process replication is extensively discussed in section 2.7.

2.6 Checkpointing

A particular problem for distributed applications with interacting processes is the creation of a consistent global state, which captures the dependencies between all involved processes correctly. Uncoordinated checkpointing of all processes may lead to a domino effect [263], i. e. if no consistent state can be reassembled from the independent taken process snapshots, the application must be restarted from the initial state. To avoid this effect some form of coordination between all involved processes must be used, e. g. the distributed snapshot algorithm proposed by Chandy/Lamport [73].

Coordinated checkpointing is associated with some overhead. To reduce this overhead especially for applications, which require frequent state saving, checkpointing is often used in conjunction with message logging [180]. Two kinds of approaches for message logging exist: in the sender-based scenario, the sender writes all messages to a stable storage before transmission, while in the receiver-based scenario, the receiver persists all messages before forwarding them to the application.

Checkpointing can be implemented in the operating system kernel, in a user-level library or in the application itself. Each of these implementation alternatives has its advantages and disadvantages. Due to the great heterogeneity in a Grid, kernel- and user-level checkpointing is less suitable. Checkpoint libraries are usually restricted to a certain plattform, e. g. the BLCR [110] library is only available for Linux. Therefore, Grid systems in general rely on application-level checkpointing to support a recovery among heterogeneous Grid resources [290].

2.7 Process Replication

Replication is a well known method to achieve high availability. Having a group of identical processes allows the masking of a certain number of faults. If a replica fails, clients still can continue to use another replica. In general, replication assumes that the replicas fail independently. Systematic faults, such as design faults, cannot be treated [279].

Replication of stateless modules is very easy. However, if the components are stateful, the management of the consistency of all replicas is a complex problem: Every time a copy is updated it must be decided how to procede with the replicated copy. To what degree inconsistencies are acceptable is mainly application dependent. If strong consistency is required, e. g. *sequential consistency* [198], some kind of synchronization, which ensures that update operations are executed at all replicas in the same order, is required. The group communication model is often used to ensure the consistency of the state of all replicas. The management of process groups and group communication itself are instances of the distributed consensus problem [254], a fundamental problem in distributed computing:

- Processes have to agree on the order of the global state updates.

- Processes have to agree on the group membership.

The consensus problem can be defined as follows: Each process proposes a value. All correct processes must agree on a value despite the presence of failures. The consensus problem is known to be not deterministically solvable for asynchronous distributed systems with faulty processes [121]. Consensus is possible in the synchronous and partial asynchronous system model (see section 2.4). However, consensus protocols are in general associated with high overheads.

In the following we discuss different replication approaches: the primary-backup, the active and the lazy replication schemas.

2.7.1 Primary-Backup Replication

The primary-backup approach [21, 64] uses a special process called the primary. Only this process receives client requests. The primary process then updates its state and distributes the update message to its backups. After the primary received an acknowledgement from all backups, it sends a response to the client process. Since all requests are routed through the primary site, they remain ordered. If the primary site fails, a new primary must be elected. For this purpose the bully algorithm can be used [136]. After the elections all requests are routed through the new primary.

2.7.2 Active Replication and Group Communication

In an active replication schema all replicas have the same role. The coordination of all processes is critical for achieving a reliable and scalable system. Commonly, the group communication abstraction is used to achieve this. In this model processes are structured into groups. A group communication framework ensures that events such as update messages or group membership changes are seen *virtual synchronously* by all group members. Every time a user performs an operation on a replica – the underlying group communication system maps the operations onto the entire replica group. Every replica in the group observes the same communication messages in the same order and can therefore maintain a consistent state. This approach is also referred to as state machine approach for replication [282]. Since all processes appear to update their state at the same time, such environments are termed virtual synchronous environments.

An important criteria for the selection of a group communication system is the correctness and consistency guaranteed by the system. Several correctness and consistency criteria have been defined. A group communication protocol is e. g. *linearizable* [162] if the action on replicated data is equivalent to some legal sequential execution. In contrast, *sequential consistency* [198] demands weaker guarantees by only requiring that all write operations are seen by everyone in the same order. Essentially, this permits that processes read outdated data. A group communication protocol that guarantees sequential consistency must ensure – even in presence of failures – that no update that compromises the global data consistency is conducted.

While strong consistency is always desirable, it is associated with high amount of overhead. There is always a trade-off between high consistency guarantees and performance. Thus, virtually synchronous execution environments in general allow the weakening of ordering properties to meet the performance requirements of applications [53, 50, 51].

Various protocols and broadcast primitives for implementing a virtual synchronous process group environment have been proposed. In the following different protocols and systems are reviewed. At first the group communication must ensure the consistency of all replicas. In general, this can be done using an ordered multicast or a voting protocol, which can guarantee that all processes see the same input messages. Group membership protocols are used to maintain the current member list of the process group.

Ordered Multicasts

The core of a group communication is an ordering protocol, which is used to order group multicasts. Birman [51] proposed amongst others the following multicast primitives each providing a different consistency guarantee:

- The *fbcast* ensures a FIFO order if the multicast is sent by the same sender.
- The *cbcast* orders potentially causally related messages correct. This relation is defined by Lamport's "happened before" relation [197], which allows the partial ordering of events in distributed systems.
- The *abcast* is a totally ordered multicast, i. e. all processes will receive a multicast in the same order.

While a cbcast can be implemented using Lamport's logical clocks, a total ordering abcast usually relies on a central coordinator, the sequencer, which orders the messages accordingly. Implementations of the abcast/cbcast primitives can be found within the ISIS [54] and Horus [316] framework. The *Totem Single Ring Protocol* [25] is another total ordering protocol, which relies on a token circulating on a logical ring.

Voting Protocols

Voting is a different approach for replica control. Initially proposed by Thomas [307] and Gifford [138], voting algorithms are able to mask both node and communication failures. Voting protocols use quorums to make decision – a quorum is a threshold of votes, which a node must acquire to conduct a operation. In general, it is differentiated between a read and a write quorum. Let r be the read quorum, w the write quorum and v the total number of votes, then both quorums must satisfy the following conditions:

1. $r + w > v$
2. $w > \frac{v}{2}$

Lamport's Paxos [199] protocol extends this consensus algorithm by introducing flexible quorums and multiple consensus rounds. The algorithm performs well especially in the presence of failures. Different commercial products and services use Paxos, e. g. Google's distributed lock service Chubby [66] and Microsoft's data center management solution [216].

Group Membership Protocol

The group membership service is a vital part of a replication framework. Task of the membership service is to maintain a list of currently active processes. The list is often referred to as *view*. When the view changes, the group membership service is responsible for notifying all members about the installation of the new view [79, 89].

As described in section 2.5.2, failure detection in distributed systems is difficult and can easily lead to inconsistent results: process A could e. g. detect the failure of process B, while process C still thinks process B is alive. Thus, an agreement protocol is often used to achieve a consensus on the current group membership. This way it is ensured that a process never communicates with a process, which does not agree with the group state. A membership change can be initiated by any process, which detects e. g. a process failure, or a new joining process.

During a network partition communication links fail in such a way that nodes are separated into groups. Nodes in each group can communicate with each other, but not with nodes of the other group. If in this case the system continues to operate, the consistency of all replicas cannot be guaranteed when the network reconnects. Different strategies to deal with network partitions exist (for a survey see Davidson et. al. [98]). Pessimistic approaches only allow access to replicas in the primary partition, i. e. all other replica processes are de-activated. To determine the primary partition a majority quorum can be used [307]. Optimistic approaches continuously permit access to all replicas. To detect conflicts, version schemas are used. In case of conflicting updates, manual reconciliation is required.

2.7.3 Lazy Replication

In contrast to active replication, lazy replication [196] algorithms asynchronously propage updates to replicas. While the performance of these algorithms is better, concurrency anomalies are very likely to occur in case data is concurrently read and written. In general, lazy replication uses some kind of versioning schema to detect inconsistency and resolve them. In contrast to active replication using the state machine approach, this often requires a precise consideration of the application's semantics.

As described, various ways for achieving the fault tolerance of applications and services exist. In the next section a classification of fault tolerance techniques with respect to the availability and reliability requirements of applications is presented.

2.8 Fault Tolerance Levels

Different applications have different dependability requirements – the selection of the fault tolerance techniques used depends on these requirements. As mentioned, there is a trade-off between the different dependability attributes. Therefore, it is useful to establish a classification of applications. Fault tolerance techniques can then be used to meet the applications dependability requirements.

While this classification can be applied to a wide variety of application, we focus on a service-oriented Grid environment. The criticality is in general determined based on the effect that a possible failure of a system has. The following fault tolerance levels are proposed by Huang/Kintala used [166]:

- *Level 0 – no fault tolerance:* These application require not fault tolerance mechanisms. In case an error occurs, the application has to be manually restarted from its initial state.
- *Level 1 – Automatic detection and restart:* If the application aborts, the error is detected and the application is automatically restarted (time redundancy).
- *Level 2 – Level 1 plus periodic checkpointing resp. logging of the internal state:* The application can be recovered from the latest saved state. Re-computations are minimized and thus the runtime of the application only increases moderately (time and space redundancy).
- *Level 3 – Level 2 plus replication of checkpoint data:* Even in case the entire node including the associated storage fails, the application can be recovered from a checkpoint stored on an external system (time and space redundancy).
- *Level 4 – Continuous operation without any interruption:* By using replicated processes it is ensured that the system has a high degree of availability (space redundancy). However, replication is not only associated with a high amount of overhead, it also adds great complexity to a system. Thus, it must be carefully considered which modules shall be replicated.

While most computational Grid applications fall into category 1 to 3, infrastructure services must be carefully ranked according to this classification. Especially category 4 services are associated with a lot of complexity and runtime overhead. In chapter 7 an extended analysis of the Migol framework with respect to fault tolerance is presented.

2.9 Summary and Discussion

Fault prevention aims to reduce the number of faults during the design and development phase of a software development endeavor. A key to achieving fault tolerance is the structuring of a software product into modules. These modules should provide a consistent, if possible fail-stop, failure behaviour enabling the detection of failures in upper layer modules. Each upper layer must carefully consider the possible failure of dependent modules. The earlier a fault is handled, the more likely a failure of the entire system can be avoided. All modules must be aware of the failure behaviour of lower-level modules. For example, clients must be able to iterate a couple of times and retry their requests to cope with transient failures.

A fault tolerant system is only able to handle anticipated faults, i. e. the fault model must be carefully chosen. Only based on a complete fault model, suitable fault tolerance techniques that cover the entire system can be deployed. A single overlooked fault can still cause a failure of the entire system. For example, even a perfectly designed software system can be subject to a power outage.

Fault tolerance is achieved by relying on redundancy in space (e. g. additional hardware, software) and/or time. Mostly relevant for computational Grid applications is backward recovery based on checkpoints. Process redundancy is usually only appropriate for critical infrastructure services. Fault tolerance is always a tradeoff between being high level of reliability and costs. Especially service replication is often associated with certain scalability limitations [148]. Thus, it is important to carefully select suitable fault tolerance techniques with respect to the requirements of the application. For example, in complex distributed systems, is reasonable to partition the system in reliable and unreliable parts. The core of a system can be designed with respect to high availability, while less critical components solely rely on checkpoint/restart.

In addition to its costs, process replication has another drawback: Replication can protect well against Heisenbugs/Mandelbugs, but not against deterministic Bohrbugs. It is assumed that the programs that are replicated are correct and do not have design faults. This is a valid assumption in case that the program was subject to rigorous debugging and testing. Nevertheless, not all kind of faults can be handled with this technique. Replication of a faulty program will lead to faulty results. Further, there is always a chance that all replicas fail at the same time – even if it is not very likely.

The reliability must also be addressed during the operational phase of a system. Several studies have shown that long-running systems show an increasing failure rate and a decrease in performance. This phenomena is also referred to as software ageing. A proactive measurement against software ageing is rejuvenation [192].

In chapter 6 and 7 the techniques covered in this chapter are applied to construct a reliable Grid environment. Chapter 10 will give an overview about fault tolerance aspects in different distributed plattforms.

"The whole is more than the sum of its parts."

Aristoteles

3 Grid Computing: State of the Art

Grid computing is concerned with efficient data and resource sharing across dynamic multi-institutional virtual organizations [130]. Different categorizations for Grids exists. Commonly, Grids are classified with respect to their intended usage into computational and data Grids. Computational Grids originated from high performance computing focusing on providing access to heterogeneous compute resources. In contrast, data Grids [77] emphasize the management of large amounts of distributed data. In practice, both types of Grids are converging, i. e. both compute and data resources must be addressed simultaneously [32]. Many compute-intensive applications e. g. also generate huge amounts of data, which later require further processing.

Although Migol considers certain data management aspects, e. g. the staging and replication of checkpoints, the main emphasis is put on computational resources. Figure 3.1 illustrates the simple Grid model used in this thesis. A Grid is assumed to consist of a federation of multiple clusters owned by different organizations. In general, a cluster consists of compute node with associated storage. Each cluster is managed by a local resource management system (LRMS). Grid users can access these machines via a OGSA services, such as an execution management services and a file transfer service.

Open standards are the pre-requisite for interoperability in heterogeneous Grid environments and are therefore an important building block. The *Open Grid Service Architecture (OGSA)* [128], standardized within the *Open Grid Forum (OGF)* [241], envisions a Grid as loosely coupled service-oriented environment. Section 3.1 provides an extensive overview about the current state of the OGSA standardization process and the implementation of those standards in current Grid middleware platforms. Fault tolerance aspects within the standards and middleware implementations are especially considered.

Having introduced important Grid standards, the penetration of these standards within common Grid middleware platforms is analyzed. Section 3.2 will describe the Globus Toolkit, while section 3.3 briefly introduces UNICORE and gLite. Further, a comparison with Globus

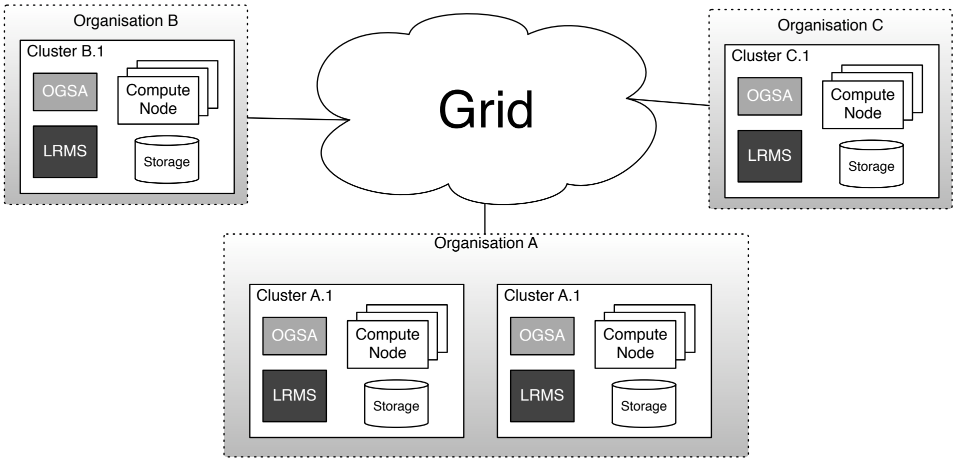


Figure 3.1: Grid Model: A Grid comprises of a set of dedicated compute clusters that are connected via a high speed network. Each cluster is managed by a local resource manager. Access to the Grid is provided via an OGSA-conform Grid middleware.

is presented. This chapter is concluded with an overview of different Grid research efforts in section 3.4 and a presentation of the Cloud computing paradigm in section 3.5.

3.1 Grid Standardization: The Open Grid Services Architecture (OGSA)

The *Open Grid Services Architecture (OGSA)* [128, 131] aims to define a blueprint for standard-based Grid computing. OGSA adopts Web services as foundation for a service-oriented Grid architecture: In this model resource capabilities are provided via well-defined Web services (see section 3.1.1). Based on different e-Science and e-Business use cases [125], OGSA identifies different basic Grid services from the following domains:

- *Information Services:* The discovery of suitable resources for applications and services in a Grid is a great challenge. This requires a common data model for resource information as well as a mechanism for publishing and querying information.

- *Execution management* services are concerned with the execution of applications on Grid resources [299].

- *Data management* addresses the aspects access, storage, movement and manipulation of data [48].

- The *security infrastructure* must facilitate the integration of security mechanisms and policies of different administrative domains to support virtual organizations.

OGSA serves as framework for the technical specification of protocols, APIs and information schemas. OGSA profiles detail other specifications for a specific use cases to clarify standards and to achieve a greater level of interoperability.

OGSA heavily relies on Web services technologies as a framework for defining protocols and information schemas. After a short overview of service-oriented computing and Web services, this section will describe the current state of the OGSA standards relevant for this work.

3.1.1 Service-Oriented Architecture and Web Services

A *service-oriented architecture (SOA)* [59] is an architectural paradigm for constructing loosely-coupled distributed systems. In a SOA capabilities are delivered through network-accessible *services*. A service is a well-defined entity which provides a certain capability to clients via the exchange of messages. The capabilities of a service are specified using a description language. Services can be discovered and invoked dynamically at runtime. Service-oriented architectures enable the construction of systems, which are independent from the implementation and location of services. Such an architecture enables the composition of distributed applications out of loosely coupled components.

Web services are a technology to build service-oriented environments [59]. The Web service model is very simplistic: It addresses heterogeneity by relying on simple XML document transfers using standard Internet protocols, such as HTTP, as input and output messages for services. The message format is specified within the SOAP standard [87]. Service capabilities are described using the Web Service Description Language (WSDL) [211] in conjunction with XML schemas [116] for data type definitions. As described, Web services are the foundation of the service-oriented OGSA framework. Thus, the development of Web service standards plays a major role for Grids.

Web Service Resource Framework (WSRF)

Web services are in general assumed to be stateless. Various extensions around the core Web service model have evolved to support more complex use cases, which require e. g. stateful services, reliable messaging, security etc. For example, the management of stateful services is a key requirement for dynamic Grids. The Web Service Resource Framework (WSRF) [93, 42], which evolved from OGSI [311] and is standardized within the OASIS consortium, is primarily concerned with the management of service states using standardized Web service interfaces. This includes e. g. interfaces for creating, accessing, and querying service states. The state of a Web service is referred to as resource.

Figure 3.2 shows the state management model used in WSRF. The factory as well as the instance service are stateless Web services, i. e. the failure of such a service will not cause the loss of any state information. Each resource is associated with a unique identifier. A client can address a service and resource using an endpoint reference (EPR); the EPR format is specified within the *Web Services Addressing (WS-Addressing)* [321] standard.

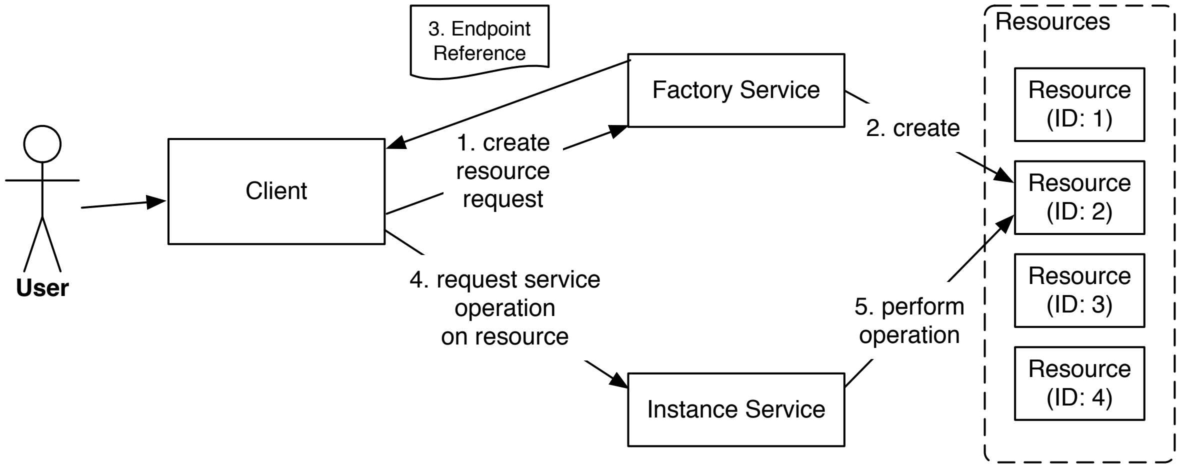


Figure 3.2: WSRF State Management: The state of a Web service is maintained within so called resources. Resources are created via a factory Web service. The endpoint reference returned from the factory is used to manipulate the state of the service.

In addition to the service interface, which is described using the *Web Service Description Language (WSDL)* [81], WSRF introduces the *Web Service Resource Properties (WS-ResourceProperties)* [334] standard, which standardizes a Web service interface for querying resource states. Resource states are maintained within a resource property XML document and can be queried and modified e. g. by using XPath [84]. The lifetime of resources can be manipulated using operations specified by the *Web Services Resource Lifetime (WS-RL)* [333] standard. WS-RL utilizes the soft state abstraction [83], i. e. unless the lifetime is periodically extended by the client, services only maintain the resource state for a limited time. *WS-Notification (WS-N)* [322] provides interfaces for implementing publish/subscribe patterns, i. e. it allows the registration of clients for receiving a notification of certain events.

WSRF is currently implemented by the two major Grid middleware platform Globus 4 and UNICORE 6. However, the successor of WSRF, *Web Service Resource Transfer (WS-RT)* [266] has already been published as draft in 2006. Although no implementation of WS-RT currently exists, it indicates that WSRF will not gain considerable more adoption. Also, some other Grid platforms, such as gLite [112], GridSAM [3] or KnowARC [190] do not implement WSRF and expose the the resource state via a proprietary interface.

Web Service Security (WS-Security)

In addition to the WSRF framework, the *Web Service Security (WS-Security)* [240] suite is heavily used by Grid platforms. The WS-Security family consists of a set of different standards, which address security issues, such as authentication, authorization, message protection, expression of policies and trust negotiation. For example, the following standards are commonly applied within Grid middleware platforms: XML-Encryption [319], XML-

Signature [320], WS-SecureConversation [152], WS-Trust [27], and SAML [239].

Web Service Naming (WS-Naming)

Multi-level naming schemas are commonly used to provide location, failure, migration and replication transparency [302] in distributed systems. The *Web Service Naming (WS-Naming)* [149] aims to provide a standard for supporting multi-level naming schemas for Web services. For this purpose WS-Naming extends WS-Addressing with a profile, which introduces a globally unique endpoint identifier (EPI) and a resolver element as addition to the endpoint reference (EPR). A resolver service can then be used to resolve an EPI to an EPR. This naming mechanism can be applied to various use cases, e. g. the migration of a Grid application. After a successful migration, a client must use the resolver service to obtain the new EPR of its application. The resolver service is a highly critical service, i. e. the reliability of this service must be ensured. An implementation of WS-Naming is found in the experimental Grid middleware Genesis II [229]. The Globus Toolkit currently does not support WS-Naming. WS-Naming is a desirable building block for a fault tolerant Grid. But, it must be ensured that implementations of the resolver service are highly fault tolerant. Otherwise, the resolution service would clearly represent a single point of failure.

Web Service Reliable Messaging (WS-ReliableMessaging)

Web Service Reliable Messaging (WS-ReliableMessaging) [99] addresses the need for message delivery guarantees for the transmission of SOAP messages. WS-ReliableMessing provides the ability to recover SOAP messages after a failure. This is in particular useful for long-running Web services.

WS-ReliableMessaging extends WS-Addressing by introducing a sequence number. The WS-ReliableMessaging runtime is then responsible to deliver the specified delivery guarantee, i. e. in case the receiver is not available the sender will retry a specified number of times to re-transmit the message. *Apache Sandesha* [31] is an implementation of the WS-ReliableMessaging based on the Apache Axis [38] Web service stack. Sandesha has currently severe limitations: Web service messages are currently only kept in an in-memory queue, i. e. rather than improving the fault tolerance, the systems increases the complexity and represents another possible point of failure. Current Grid middleware platforms do not support WS-ReliableMessaging. But, a support for WS-ReliableMessaging is planned for future Globus versions [26].

Conclusion

Web services, in particular technologies such as XML schemas, WSDL and SOAP simplify the development of protocols since these standards provide a powerful abstraction to define message exchange patterns, to specify interfaces and information schemas. Implementors can rely on existing Web service frameworks and minimize the implementation effort.

In addition to the described Web service standards, several further partially competing standards exist or are currently specified, e. g. WS-Policy, WS-ResourceTransfer [266] (planned as successor of WSRF). While much of the functionality specified by the WS-* standards is indeed very useful, a main concern is the increasing complexity (see section 11.2).

3.1.2 Information Model

An abstract, implementation-independent information model for describing different aspects of the Grid, such as resource capabilities, is another pre-requisite for supporting important Grid functions such as resource discovery. Within the OGF currently two different information models are discussed:

- The *Grid Laboratory Uniform Environment schema (GLUE schema)* [28] was especially designed with respect to Grid computing drawing from experiences of production Grids, such as the EGEE or NorduGrid. The schema is used within Globus, UNICORE, gLite and other Grid middleware platforms.

- The *Common Information Model (CIM)* [108] is a standard developed by the Distributed Management Task Force (DMTF). It provides a common information model for different aspects such as systems, networks, applications, and services. The schema is primarily used in business-oriented environments.

Due to its penetration in current production Grids (TeraGrid, NorduGrid) and middleware platforms, the emerging GLUE 2.0 standard is favored by the OGF Grid Interoperability Now (GIN) community group [268].

3.1.3 Execution Management

The most important use case of compute Grids is the execution of jobs on arbitrary resources. Different standards, such as the *Job Service Description Language (JSDL)* and the *OGSA Basic Execution Service (OGSA-BES)* as well as the *High Performance Computing Basic Profile (HPCBP)* have emerged. This section will give a brief summary of these specifications.

Job Submission Description Language (JSDL)

The *Job Submission Description Language (JSDL)* [30] is an XML-based language for describing the requirements of computational jobs. This includes the specification of the following capabilities:

- Resources required to execute a job, i. e. the number of nodes, operating system etc.

- Runtime environment for a job, i. e. the executable, the parameters, environment variable etc.

- Data files that must be staged before a job runs.

Most commonly, JSDL is used to submit a job to an execution management service, e. g. an OGSA-BES service. The standard JSDL focuses on POSIX applications, i. e. applications which are executed on a single node. The JSDL language can be extended via a JSDL profile or a non-standardized extension. The following extensions exist:

- **High Performance Computing (HPC) Profile**: The HPC profile extension [168] adapts the standard POSIX application profile defined by JSDL 1.0 to meet specific HPC requirements and remove interoperability constraints due to a solely focus on Unix applications. The application description is based on the classical JSDL POSIX application, but removes certain features to ensure the interoperability with other operating systems, such as Windows.

- **Single Program Multiple Data (SPMD) Extension**: The SPMD extension [5] standardizes the specification of execution requirements for parallel applications, which are based on the Message Passing Interface (MPI)[231] or OpenMP [243]. For example, the profile supports the specification of multiple hosts and defines a set of URIs for referencing different parallel environments.

Most middleware platforms support JSDL in addition to their proprietary description language. For example, Globus GridWay provides a JSDL to GridWay job description converter.

JSDL is associated with different limitations: the standard is designed very generically, which limits the interoperability especially when using proprietary extensions. Further, JSDL does neither address the lifecycle of jobs nor simple workflows.

In general, a JSDL document is used as input parameter for an execution management service, such as an OGSA-BES compliant service.

OGSA Basic Execution Service (OGSA-BES)

The *OGSA Basic Execution Service (OGSA-BES)* [126] aims to standardize the Web service interface for creation, monitoring and cancellation of activities. Activities can be considered as generalization of classical jobs. In general, activities are described with JSDL. After submission, the OGSA-BES service executes the defined activity on its managed resource, e. g. a compute cluster. For this purpose, OGSA-BES defines a set of Web service porttypes, which provide the following operations:

- Querying of information about the managed resource.

- Creation of activities.

- Lifecycle management of activities, i. e. querying of the activity state and termination of activities.

Different Grid middleware platforms and commercial vendors have adopted the OGSA-BES specification, e. g. Unicore 6, gLite/CREAM-BES, Microsoft Windows Compute Cluster, Altair PBSPro and Platform Computing BES++ [4].

High Performance Computing Basic Profile (HPCBP)

The *High Performance Computing Basic Profile (HPCBP)* [107] refines JSDL, the JSDL HPC Profile, and OGSA-BES to support an interoperable job submission of HPC jobs. To achieve interoperability the HPC Basic Profile specifies the precise usage of certain JSDL elements, refines the OGSA-BES resource information query interface and defines the required security features. Despite the achieved level of interoperability, there are still limitations:

- OGSA-BES/HPCBP allows the modelling of BES services with and without WSRF, i. e. clients must support both types of interfaces. This restricts the interoperability.
- MPI applications are not within the scope of the HPC Basic Profile.
- Data staging is not part of the profile.
- Security issues such as credential delegation are not part of BES and are not addressed by HPCBP.

With this severe limitations the HPCBP compliant BES services are still not suitable for production environments [268].

Another important resource management aspect, is the negotiation of service level agreements, such as advance reservations. This capability is in particular interesting for deadline driven scenarios, e. g. for modeling severe weather events such as hurricanes.

WS-Agreement

WS Agreement [29] defines a Web service based protocol for the dynamic negotiation of agreements between service providers and consumers. An agreement ensures a set of resources and/or certain quality of services to the service consumer. WS-Agreement aims to address a wide range of scenarios, e. g. advance reservations. An advance reservation commits a particular resource over a defined time interval from the service provider to the consumer [270].

The WS-Agreement specification defines the format of an agreement template and an agreement based on a XML schema and a WSRF-based protocol for establishing and monitoring agreements. However, the specification only defines the higher level structure of an agreement. The agreement types must be supplemented by domain specific information, e. g. to describe the type of service.

WS-Agreement provides a very high-level standard for implementing SLA negotiations. Due to the many potential different information models, interoperability is difficult to measure. Further, the protocol does not support more complex negotiation and re-negotiation processes.

While different Grid and cluster services support the notion of quality of services, sophisticated agreement processes as proposed by WS Agreement are very rare. An example for a WS-Agreement implementation is the Viola/UNICORE [300] project, which supports the reservation, co-reservation and co-allocation of compute- and network resources. Another open source implementation of WS-Agreement has been provided by the AssessGrid project [45].

Distributed Resource Management API (DRMAA)

For the development of Grid applications and higher-level resource management services, a standardized interface to local and Grid resource management systems is crucial. The *Distributed Resource Management API (DRMAA)* [310] is a high-level API for interfacing applications with resource management systems. DRMAA covers the complete lifecycle of LRMS and/or Grid jobs, i. e. the submission, monitoring, and controlling of a job. DRMAA bindings are available for many commercial and open source LRMS and Grid scheduler, e. g. Sun Grid Engine, Platform LSF, Altair PBS Pro, Condor, Torque, and GridWay.

In addition to the resource management standards, the SAGA and GridCPR standard are discussed in the following.

3.1.4 Simple API for Grid Applications (SAGA)

The *Simple API for Grid Applications (SAGA)* [143] provides a high-level, unified, application-oriented programming interface to Grid infrastructures. The API aims to abstract the complexity of current Grid middleware platforms, which is mainly caused by the Web service environment. SAGA is structured into two sets of packages: the functional packages and the look-and-feel packages. The functional packages provide a simple access to Grid middleware services, such as file transfer, job submission, replica management etc. Look-and-feel packages address non-functional aspects, such as authentication, authorization, and monitoring. At the same time, the API provides an abstraction layer to different Grid middleware platforms. The C++ as well as the Java reference implementation provide an adaptor mechanism, which is used to encapsulate middleware specific code. With this functionality applications can use the same API independent from the underlying middleware [176]. The SAGA API enables developers to easily integrate applications and Grid infrastructures: applications can easily spawn tasks or move files using SAGA. This makes the framework very attractive for Grid applications and scenarios addressed in this work (see section 4.3).

In the following, the GridCPR draft, which describes an architecture for checkpointing and recovering of Grid applications, is described. GridCPR and the Migol architecture served as template for the newly developed SAGA Checkpoint Restart (CPR) API, which is discussed in section 6.7.

3.1.5 GridCPR

GridCPR [290, 40] is an early attempt made of the OGF to describe a service architecture for the automatic recovery of computational jobs. Figure 3.3 shows the proposed GridCPR architecture. GridCPR identifies the following Grid services:

- *State Management Service (SM):* Stores metadata about jobs and checkpoints.
- *Event Handling Service (EH):* This service is responsible for error detection.

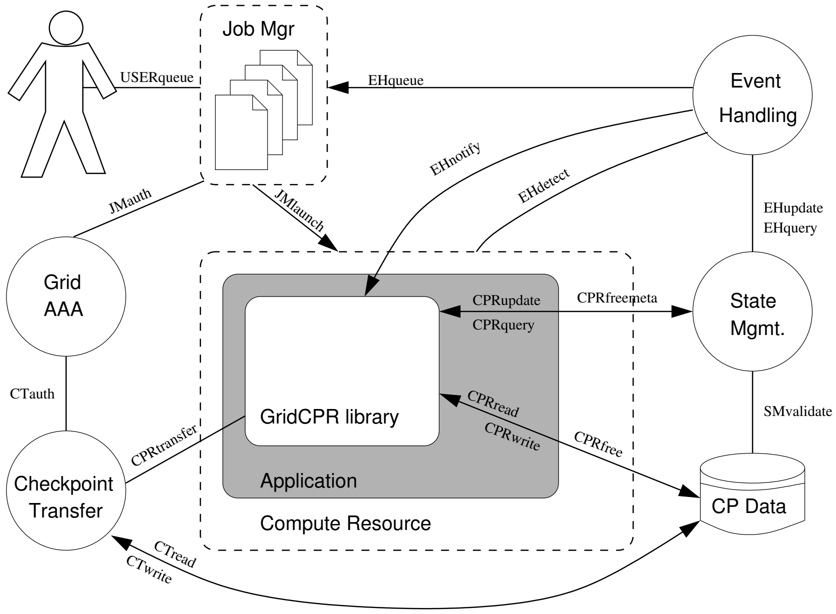


Figure 3.3: GridCPR Architecture: GridCPR describes a set of services for supporting the checkpointing and recovery of Grid applications.

- *Checkpoint Transfer (CT):* Reliable transfer service for checkpoints (e. g. GridFTP, RFT).
- *Job Management (JM):* The job manager is responsible for launching respectively restarting applications.

The GridCPR library provides an unified access to these services. Currently, the application interface is standardized within the SAGA working group. The SAGA CPR package [223], which was significantly influenced by the Migol framework, is the result of these efforts. SAGA CPR provides an API for applications to manage checkpointing and recovery.

GridCPR emphasizes the need for an automatic checkpoint recovery system for Grid applications. However, further refinements are necessary to transfer GridCPR into a working system. This includes the specification of the service interfaces, of the state model, as well as of a common information model. Further, the security impacts of such an infrastructure must be discussed. Also, there is an overlap with other OGSA specifications. For example, job management is covered by the OGSA-BES [126] specification. Further, GridCPR does not address the reliability and availability requirements of the proposed services. In particular, the availability of the GridCPR state manager, the central metadata store of GridCPR, requires further consideration.

3.1.6 Conclusion

Although Web and Grid standards are evolving fast, todays Grids are far from being interoperable. The Web service foundation has reached some level of maturity mainly due to

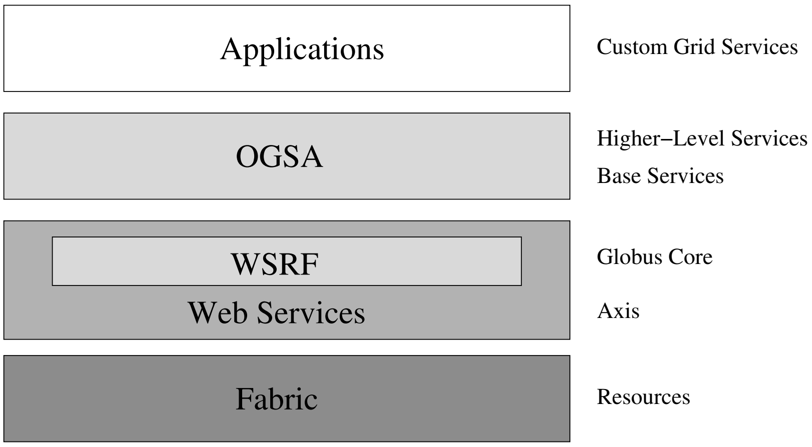


Figure 3.4: Globus Toolkit 4, OGSA and WSRF (cmp. [292])

interoperability standards such as the WS-I Basic Profile[332], which refines the usage of SOAP, WSDL and UDDI to achieve interoperability. However, the OGSA framework is still under construction: the highest level of standardization has been achieved for the job execution use case. Activities like *Grid Interoperability Now (GIN)* [8] indicate that standards as OGSA-BES, GLUE and JSDL require more standardization work to support the HPC use case sufficiently. Further, advance features such as advance reservation, which is demanded by many OGSA use cases [125], is not addressed at all.

Another promising standards is the Simple API for Grid Applications. SAGA aims to standardize the Grid from the application point of view, i. e. rather then specifying the WSDL interfaces of a Grid services, SAGA provides an easy-to-use, middleware-independent, application-level API.

Although Fault tolerance is a fundamental requirement of almost all use cases [125], it is not considered sufficiently by most standards. For example, OGSA-BES does not support the automatic restart of jobs. Further, the fault model specified by the standard is insufficient. While OGSA-BES describes some faults, such as a `UnsupportedFeatureFault` or `InvalidRequestMessageFault`, which clients must expect, a specific behaviour in case of a node failure is not demanded.

In the following the Globus Toolkit as well as the gLite and UNICORE Grid middleware platforms are discussed.

3.2 The Globus Toolkit

The *Globus Toolkit* [123] is an open source middleware for building OGSA-based Grids. The current version is the Globus Toolkit 4 (GT4). GT4 extensively uses Web services and the Web Service Resource Framework (WSRF) for the implementation of these services. However, many production Grids still rely on the GT2 pre-WS services, such as the pre-WS GRAM [94], which are also part of GT4.

Figure 3.4 shows the relationship between GT4, WSRF and OGSA. At the lowest layer GT4 provides a Web service runtime environment for hosting loosely coupled services. This runtime environment is based on the Apache Web service implementation Axis [38]. GT4 extends Axis with several handlers and providers to support federated security and WSRF. Further, GT4 provides various OGSA-inspired services, which publish different resource capabilities from the fabric layer. This includes services for execution management, data management, information management and security. Most of these services are hosted within the GT4 WSRF container. These services can again be aggregated by other higher-level services and Grid applications.

3.2.1 Grid Security Infrastructure (GSI)

The *Grid Security Infrastructure (GSI)* [324] is the security framework of GT4. GSI provides the fundamental security services needed to support virtual organizations. This includes services for authentication, authorization, single sign-on, and delegation. The security framework is based on a public key [106] infrastructure.

Public Key Infrastructure and Proxy Certificates

A public key infrastructure (PKI) is a system for the management of public and private keys as well as the associated certificates. Certificates are digital documents that confirm that a certain public key is owned by a particular user or service. They are issued by a trusted certificate authority (CA). GSI is based on X.509 certificates [164]. Each certificate is identified by a unique name called distinguished name (DN). Every user, service, and host in a Grid is associated with an X.509 certificate. In the following, an X.509 certificate with the associated private key, which is used by a Grid entity to authenticate itself, is also referred to as credential.

Proxy certificates [312] were introduced by GSI as an extension to X.509 certificates and enable the implementation of single sign-on and delegation. A proxy certificate is a short-living certificate, which is derived from and signed by an X.509 end entity certificate or another proxy certificate. Proxy certificates usually assume the identity of the issuer (impersonation) and can therefore be used to delegate privileges from an issuer to another party (see section 3.2.1).

GSI Services

GSI provides different services and handlers for proxy certificate-based authentication, message protection, and delegation. The framework supports the Secure Socket Layer (SSL)/ Transport Layer Protocol (TLS) [105], as well as Web service based protocols, such as WS-Security, WS-SecureConversation, WS-Trust, and SAML, to provide security (see also [139, 272]). Most of these security features are implemented transparently for the application within the GT4 hosting environment [325].

Authentication and Authorization

GSI supports authentication of clients and services based on proxy certificates. Authentication is performed by verifying the signature of the peer's proxy certificate and conducting a simple challenge response protocol.

Further, GSI provides a framework for client-side and server-side authorization. The following authorization types are supported:

- **Gridmap:** All users that are specified in a special file are granted access to a service.
- **Self:** Only the user with the same identity as the service identity is allowed to use the service.
- **Identity:** This authorization mode is only used on client-side. Authorization is accomplished by comparing the identity of the service to a specified identity.

In addition, Globus provides various possibilities to authorize users based on attributes, e. g. based on a SAML assertion or an attributed certificate.

Another important issue is the identity management within virtual organizations. For this purpose, several virtual organizations management services for maintaining users and roles have emerged. For example, the *Virtual Organization Membership Service (VOMS)* [15] provides a service for managing VO members and roles. Based on the defined user roles and attributes, a VOMS service issues on request authorization assertions in form of attributed certificates or SAML assertions. These assertions can then be used by services to authorize users. In addition to Globus, VOMS is also supported by other platforms such as UNICORE and gLite (see section 3.3.3). The VOMS service is widely used in production Grids, e. g. the D-Grid and EGEE Grid.

Delegation

An important requirement in a service-oriented environment is the delegation of privileges. Delegation allows a service to act on behalf of another security principle to get access to another service. The delegation of credentials is the foundation for the establishment of dynamic trust relationships between services.

After a successful authentication and authorization a service can request the delegation of the caller credentials by using a method of the `SecurityManager` class, which is provided by the hosting environment.

Figure 3.5 illustrates the delegation of a credential from service A to service B.

1. The initiator service A (delegator) connects to the target service B (delegatee). Service A and B perform a mutual authentication and authorization using GSI (TLS/WS-Security).
2. After service A indicated its desire to delegate, service B generates a new public and private key pair. Further, a signed certificate request is created.
3. The certificate request is sent to service A.

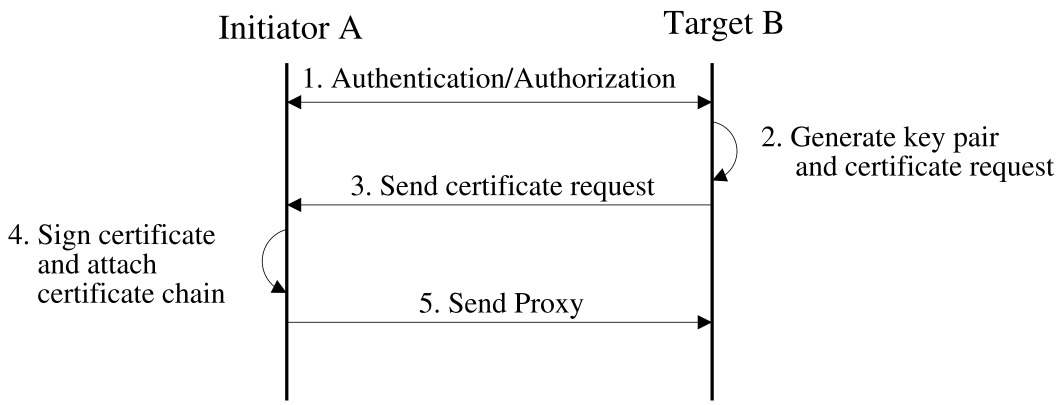


Figure 3.5: Delegation of Proxy Certificates

4. Service A uses its private key associated with its own proxy certificate to sign the certificate request. To enable a validation of the proxy, the certificate chain is updated and attached to the proxy.

5. The newly created proxy certificate is sent back to service B.

This delegation protocol enables the transfer of privileges over a network connection without the exchange of private keys.

Having described the underlying security infrastructure of GT4, the basis Grid services offered by Globus are introduced in the next sections.

3.2.2 Web Service Monitoring and Discovery Service (WS MDS)

The discovery of resources is a vital operation in a service-oriented environment. The *Monitoring and Discovery Service (WS MDS)* [286] is the central information service of Globus. The WS MDS provides an extensible framework for aggregation of resource information, querying and monitoring of information. The service makes heavily use of the interfaces defined within the WSRF framework. Information is collected via the WS MDS aggregator framework, which e. g. allows the simple registration of information providers. An information provider can be a WSRF service, but also a simple script, which queries local information. By default local cluster information is collected und published in the local MDS using the GLUE schema [28]. This information can be complemented by additional information providers, e. g. the Ganglia [135] or NWS [329] information provider.

3.2.3 Data Management

Globus provides different services for executing data transfers as well as for managing file replicas.

Data Transfer

The movement of data is a fundamental task in Grids. A standard protocol for the transfer of large files is *GridFTP* [16, 218]. GridFTP extends the standard FTP [260] protocol by Grid specific features, such as GSI authentication, support for third-party transfers, several transfer optimization options, such as pipelining and parallel transfers (see [62]) and reliability features such as the manual restart of transfers.

The *Reliable File Transfer Service (RFT)* [17] provides a WSRF-based Web service for the management, i. e. the initiation, controlling and monitoring of file transfers. For the actual transfer, RFT relies on GridFTP. In contrast to GridFTP, the transfer is managed by the service, i. e. it is not required that the user remains connected during the entire transfer. In addition, RFT transparently handles transfer errors, such as transient network failures, by automatically restarting transfers.

Replica Management

The *Globus Replica Location Service (RLS)* [76] provides a simple information service, which maintains information about physical locations of file copies (replicas) using a unique logical filename. RLS instances can be hierarchically aggregated: Local Replica Catalogs (LRCs) store replica information for a local site, while Replica Location Indices (RLIs) consolidate data of a set of LRCs. While this setup provides some fault tolerance, it lacks strong consistency guarantees since different RLIs are not required to have the same consistent view of the current Grid state. Further, the data model of the RLS currently does not support all required aspects: for example, it is not possible to store the state of a replica in the RLS. Especially in a Grid it is possible that certain physical files become unavailable - thus, storing the state of a replica is a critical feature.

The *Globus Data Replication Service (DRS)* [141] offers a higher level Grid service for automatic replica creation. The DRS automatically conducts file transfers to a specified resource and creates the respective RLS entries. But, the service lacks autonomic capabilities, such as the automatic selection of a destination Grid resource or the detection of a new checkpoint file to replicate.

3.2.4 Execution Management

The Globus Toolkit 4 provides different services for the management of computational Grid jobs. In the following, the GRAM4 job submission service and the meta-scheduler GridWay are described.

Grid Resource Allocation and Management (GRAM)

The *GT4 Grid Resource Allocation and Management (GRAM4)* [118] addresses the aspect execution management in OGSA Grids.[1] The service provides a Web service interface for the

[1] The GRAM4 was prior Globus 4.2 known as WS GRAM.

initiation of an execution of an arbitrary program on a remote resource. GT4 also contains the deprecated GT2 GRAM (GRAM2/pre-WS GRAM), which provides similar functionality as the GRAM4, but uses a proprietary non Web service based protocol (for further information, please refer to Foster et. al. [127]). In the following, the term GRAM refers to, if not otherwise noted, to the GRAM4.

Core task of the GRAM service is to provide an abstraction layer on top of local resource management systems (LRMs). Currently a wide range of LRMS are supported, e. g. the Portable Batch System (PBS) [161], the Load Sharing Facility (LSF) [340], Torque [309], Sun Grid Engine (SGE) [289] and Condor [209].

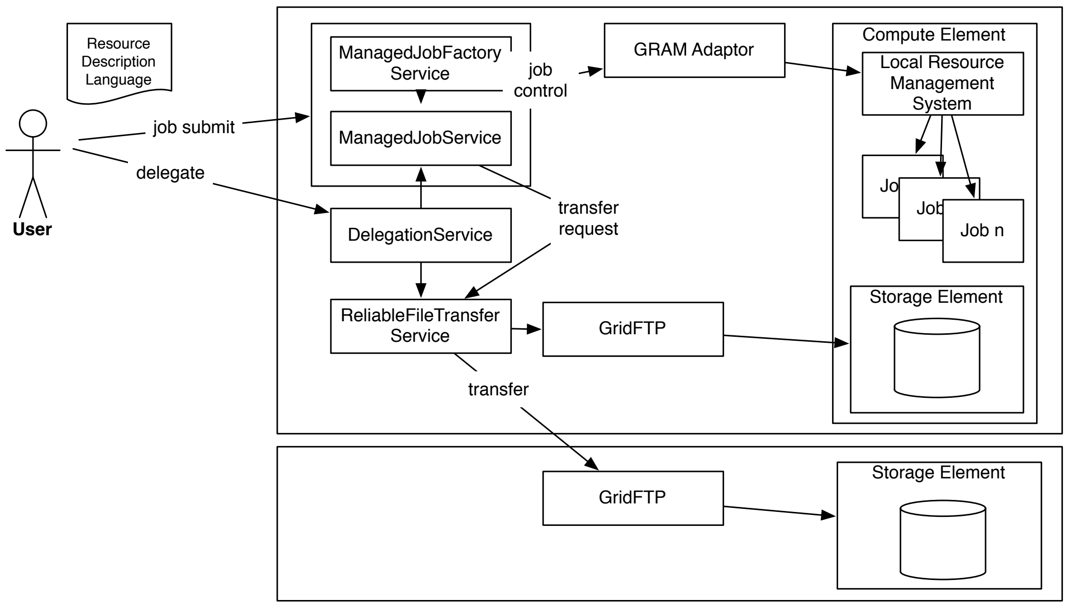


Figure 3.6: GRAM4 Architecture

The GRAM service handles all job submission related aspects, i. e. the staging of the necessary files, the interaction with the local resource management system as well as the monitoring of the application run. Figure 3.6 shows an overview of the GRAM4 architecture. For the specification of applications requirements and characteristics the *Resource Description Language (RSL)* is used. The initiation of a job is done via the Managed Job Factory Service. The service creates a WSRF resource, which is used to maintain the job state. In the following the management of the job itself is done via the Managed Job Service. For file staging the GRAM4 relies on the Reliable File Transfer (RFT) service and GridFTP. After the staging is complete, the Managed Job service will use the GRAM adaptor to start the application. The adaptor will then map the job description to the syntax of the local resource management system (LRM) and start the job. During the runtime the job can be monitored via the Managed Job service. Clients can also subscribe to job state notifications. After the job terminates or is cancelled the job resource will be destroyed.

Security aspects such as the authentication, authorization and message protection is handled by the Globus container using standard GSI mechanisms. For delegation of the credential the GT4 Delegation Service is used. The GRAM services ensures the least privilege principle by mapping the Grid user to a local user id, which is then used to execute the job.

The GRAM4 currently has different limitations: The specification of the job must be done using a proprietary description language. Important standards such as JSDL or OGSA-BES are currently not implemented. Advanced features, such as the automatic restart of jobs or advance reservation are also not supported.

GridWay

Gridway [167] is a distributed resource manager, also referred to as meta-scheduler. Client applications can communicate with the scheduler via a command-line program or using the DRMAA API. For the specification of jobs, Gridway requires a custom job description. A JSDL description can be converted using a parser. Similar to the GRAM, GridWay clients can control the lifecycle of jobs, i. e. the submission, monitoring, cancellation and re-scheduling of jobs. The dispatch manager is responsible for periodically scheduling or re-scheduling unsubmitted, pending or failed jobs. GridWay supports automatic resource selection e. g. based on global WS MDS data. Resources are prioritized using the resource selector module. The module supports different standard policies, such as round robin, but can also be extended. The starting of a job on a resource is done via the submission manager, which submits jobs to the Grid middleware. Although GridWay was primarily developed with focus on the Globus GRAM, it can support other platforms via a middleware access driver.

GridWay also provides capabilities for supporting the fault tolerance of jobs. Error detection is done via of the job status or by using the notification mechanism of GRAM. When an unrecoverable failure is detected, GridWay attempts to resubmit the job. The reliability of the service itself is ensured using a checkpoint, i. e. after a failure GridWay can recover its state. In comparison with Migol, GridWay lacks certain features, such as application-level error-detection, which is able to detect more complex und also user-defined errors, and a reliable monitoring service. Further, the overall reliability of the infrastructure is not addressed sufficiently.

3.3 Other Grid Platforms

In addition to the Globus Toolkit several other Grid platforms have emerged. In this section, the UNICORE 6 and gLite middleware as well as a short comparison with Globus is presented.

3.3.1 UNICORE

With *UNICORE 6* [222] the core Grid functionalities, such as remote job submission, data management or information services, are exposed via a WSRF compliant interface. For modeling of stateful resources UNICORE also uses WSRF. UNICORE provides a custom WSRF service container called WSRFLite. The container is based on the XFire SOAP stack and the Jetty servlet container.

The security model of UNICORE is based on end-entitiy X.509 certificates, i. e. in contrast to Globus it does not use proxy certificates. User certificates are maintained within the UNICORE GUI client. For authorization UNICORE relies on a XACML-based ruleset.

Figure 3.7 shows the UNICORE 6 stack. At the bottom layer the Target System Interface

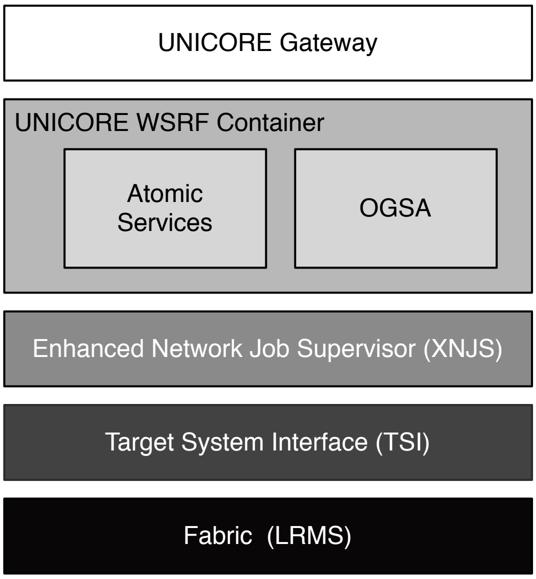


Figure 3.7: UNICORE 6 Architecture (cmp. [221])

provides an interface to the fabric layer, i. e. the site specific LRMS (e. g. PBS or LSF). The main execution management functionality is encapsulated within the enhanced Network Job Supervisor (XNJS). The XNJS expects jobs described in JSDL. After submission the XNJS is responsible for mapping the JSDL job description onto site-specific application commands. With UNICORE 6, different Web service interfaces can be used to access UNICORE. The UNICORE Atomic Services (UAS) provide a WSRF-based interface for the management of jobs and data. The Job Management Service, Target System Service and Target System Factory Service abstract the XNJS capabilities. The Storage Management Service (SMS) and File Transfer Service (FTS) are used to manage the file staging, which is done using HTTP/HTTPS. The Gateway provides a single point of entry to UNICORE sites and is responsible for the authentication of the user. If the user is successfully authenticated, the request is forwarded to the requested service. UNICORE 6 can be used with a variety of clients, e. g. the Java-based Grid Programming Environment (GPE) [265], an Eclipse based client and the command-line client.

UNICORE 6 also supports the emerging OGF standards OGSA-BES and ByteIO [221]. Both interfaces are similar to the proprietary interface provided by the atomic services. Thus, most of the functionality can simply be mapped to the standard interfaces.

As part of the VIOLA project the Meta Scheduling Service (MSS) [326] was developed. The MSS supports the co-allocation of computational and network resources based on advance reservation and WS Agreement. Unfortunately, the MSS is based on another WSRF implementation (Apache MUSE) and thus is not part of the standard UNICORE distribution.

3.3.2 gLite

The *gLite* middleware [65, 112] offers a variety of services for data and compute Grids and is currently developed within the EGEE and the Worldwide LHC Computing Grid project (WLCG). The gLite middleware integrates different components from the Globus and Condor project. The architecture of gLite follows the service-oriented paradigm, but not all services are currently Web service based.

As gLite is based on GT2, the Globus Grid Security Infrastructure (GSI) is used for all security related aspects. The core of gLite is the Workload Management System (WMS) a centralized meta-scheduler. The WMS is responsible for distributing compute jobs across different Grid resources. Jobs are described using the gLite Job Description Language (JDL). The WMS accepts job requests of users and forwards them to a selected Compute Element. The selection of a resource is done using a matchmaking procedure. For this purpose, the matchmaker can obtain resource information from the gLite information service. After selection of a resource, the job is submitted. The access to the resource is managed by the Computing Element (CE) service, a service with similar functionality as the Globus GRAM. The CE service interfaces with the local resource management system. For management of the job execution Condor-G [132] is mostly re-used. With CREAM-BES [88] an OGSA-BES compliant layer on top the gLite CE exists.

The Berkley Database Information Index (BDII), a more scalable redevelopment of the pre-WS MDS, is used as central information service for gLite-based Grids. Similar to the MDS, it is distinguished between a site-level and a global BDII service. The global BDII aggregates all local BDIIs. Information is stored in the GLUE schema. Queries can be done using the LDAP query syntax.

The Storage Element (SE) Service is utilized as abstraction layer for different data transfer protocols. At minimum all storage elements must support GridFTP. In addition, support for further protocols e. g. dCache Access Protocol [100] is provided.

3.3.3 Middleware Comparison

A variety of Grid platforms have evolved: Globus, UNICORE and derivates such as gLite. Many of these frameworks are used in production Grids around the world. Although they offer the same set of core services, e. g. for execution management, these services are implemented in different ways and are usually not interoperable.

In Table 3.1 the Globus Toolkit 4, UNICORE 6 and gLite are compared with respect to their functionalities and standard adoption. The table indicates that Grid platforms are neither on syntactic level (not all platforms provide WSRF support) nor on semantic level (different job management and information service interfaces) interoperable. OGSA-BES is currently being adopted by gLite and UNICORE and will be implemented in a future Globus version. However, OGSA-BES is not sufficient for most scenarios due to the missing support for a common security model, file staging and MPI applications. From the information model point

Service	**Globus Toolkit 4.2**	**UNICORE 6**	**gLite**
Web Service Support			
State Modeling	WSRF 1.2	WSRF 1.2	plain Web services
SOAP Framework	Axis 1	XFire	gSOAP
Execution Management			
Job Execution	GRAM4	JMS/XNJS	Computing Element
Meta Scheduling	GridWay	Viola/MSS	WMS
Data Movement	GridFTP	ByteIO	GridFTP
EMS Standards	JSDL (GridWay)	OGSA-BES, JSDL	OGSA-BES, JSDL
Information Model and Services			
Information Model	GLUE	GLUE	GLUE
Information Services	WS MDS	Registry Service	BDII
Security			
Authentication	X.509/X.509 proxy	X.509	X.509/X.509 proxy
Authorization	XACML SAML VOMS	XACML SAML VOMS	VOMS

Table 3.1: Grid Middleware Comparison

of view, the GLUE schema has gained considerable adoption within all three platforms. The WS MDS and the UNICORE Registry Service both expose the information meta data via a WSRF interface, while the gLite BDII relies on the LDAP syntax. The common denominator of the security architecture are X.509 certificates (end-entity or proxy certificates). Further, the VOMS service, which supports both SAML and attributed certificate based authorization, receives good support from all Grid platforms.

3.4 Grid Research Initiatives

Many Grids and e-Science infrastructure are currently in development. Often these initiatives have been funded by the government to enable advance scientific discovery and economic development within a country or region. These Grid infrastructures include:

- D-Grid [95] is the German Grid initiative, which started with several scientific community Grids, e. g. the astrophysical Grid (AstroGrid-D) and the high energy physics Grid (HEP) communities, in 2005. The second phase of the D-Grid, which started 2007, additionally emphasizes industrial and business application use cases. Further, operational aspects and sustainability are addressed.

- EGEE [304] is a European initiative, which aims to integrate current regional and national Grid efforts. A well-known pilot application for EGEE is the Large Hadron Collider Computing Grid. This Grid supports the high-energy physics experiments conducted at CERN in Switzerland.

- The TeraGrid [305] is the US Grid infrastructure initially established in 2001. With currently more than 750 TFlops of compute resources, over 30 Petabyte of storage as well as high network connections the TeraGrid is one of the world's largest Grid infrastructures. In addition to the core infrastructure, the TeraGrid science gateways have become a driver of the TeraGrid. These gateways provide a community specific Grid entry point designed by domain experts of the field.

All these activities share the same vision of providing a seamless infrastructure to support scientific research. In addition to these three examples many other similar infrastructures are currently developed around the world, e. g. NAREGI in Japan [227] or the Chinese CROWN Grid [92].

3.5 Cloud Computing

Cloud computing is often referred in the context of Grids. Staten defines Cloud computing as follows:

> "A Cloud is a pool of abstracted, highly scalable, and managed compute infrastructure capable of hosting end- customer applications and billed by consumption [294]."

Clouds facilitate the deployment and dynamic scaling of applications and services without the need for an own compute and storage infrastructure. With this concepts Clouds are the latest incarnation of the classical utility [220] and Grid computing concepts. But, Clouds are different with respect to the used abstraction. Clouds heavily rely on server virtualization technologies such as Xen [43] to decouple the functionality of applications and physical resources.

The best known example for Cloud services are the Amazon S3 and EC^2 services. While the Amazon Simple Storage System (S3) [23] service provides network-accessible storage via the Internet, the Elastic Cloud (EC^2) [22] offers compute cycles via dynamically instantiated virtual servers. Access to these services is provided via open protocols, such as HTTP, REST and SOAP, which allow an easy integration of these services in existing applications.

Figure 3.8 shows a high-level overview of a Cloud infrastructure, which provides access to virtual server instances. The instances are automatically managed by a hierarchical infrastructure consisting of host, cluster and central Cloud controllers. The user interface is provided via a Web service.

Broader definitions describe Cloud computing as any subscription-based or pay-per-use service that is used over the Internet. This includes in addition to utility infrastructure providers also software-as-a-service (SaaS) providers. SaaS is a delivery model for software, which solely relies on the user's Web browsers. SaaS providers can of course again aggregate services provided by utility providers such as EC^2.

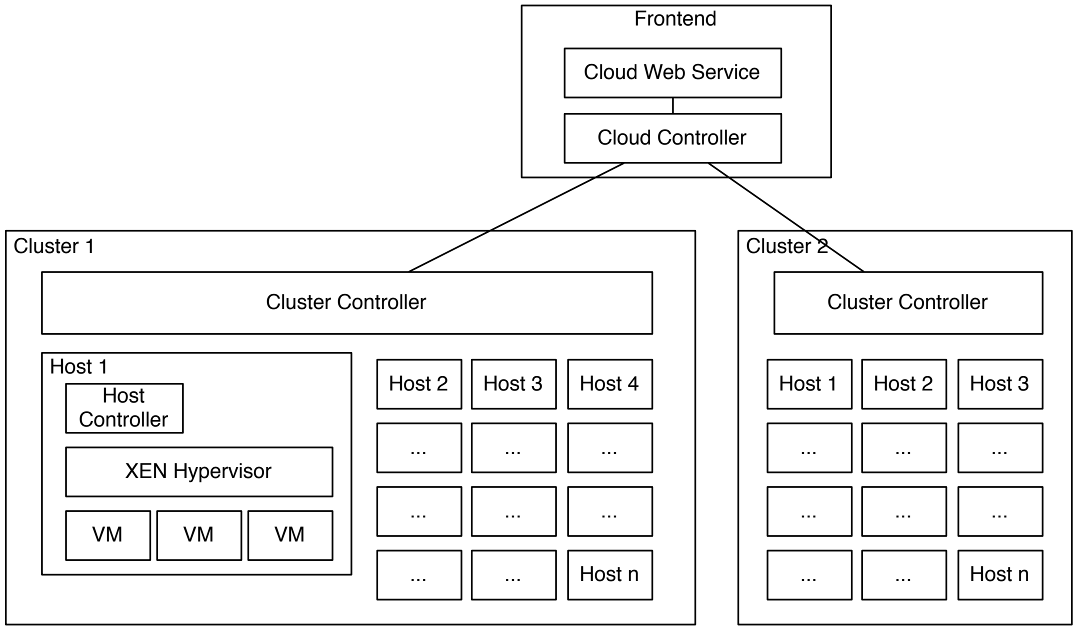


Figure 3.8: Eucalyptus Cloud Service Architecture [330]

Currently, no clear alignment of Cloud and Grid computing exists. Jha et. al. [177] define Cloud computing as a narrow Grid, which exposes a limited set of well-defined features to serve domain-specific use cases. Grids in contrast tend to offer a maximal set of semantics to end-users. In the example of Amazon EC^2 the capability provided is simply the possibility to create on-demand virtual machines or clusters. These machines can serve arbitrary applications e. g. MapReduce. General purpose Grids are seen as technology basis for the provisioning of Cloud services. In contrast, Vogels describes a Cloud as environment which can be used to deploy Grid services as well as Grid applications [318].

Clouds services like EC^2 enable users to deploy whole application stacks to a virtual data center. Grid applications, especially those requiring complex deployment procedures, can benefit from the virtual machine abstraction [184]. Various Grid middleware services, which focus on the management of virtual machines on remote clusters have emerges, such as EUCALYPTUS [330], OpenNEbula [228] and Globus Virtual Workspace Service [184]. However, Palankar et. al. [248] and Bégin [46] highlight that Clouds are not as feature rich as Grids and that not all applications are suitable for running in a Cloud. While these reports emphasize the limitations of Clouds, they also state that the abstractions provided by Clouds are clearly something Grids can benefit from in the future.

Running applications in Clouds is quite similar to Grids: resources must be discovered, allocated and efficiently used. This requires a suitable programming abstraction, which can exploit the parallelism in a virtual cluster. Despite the use of open protocols such as HTTP, Cloud solutions are mostly restricted to a single Cloud provider. Interoperability, i. e. the federation of resources across different organizations is, in contrast to Grids, not a major concern. Projects such as RESERVOIR [267] try to address this issue. Another important aspect for Cloud computing is trust, which is currently solely based on the reputation of the cloud provider.

3.6 Summary and Discussion

The Open Grid Service Architecture (OGSA) envisions an open, service-oriented Grid architecture as the fundament for building Grids. To achieve this, OGSA aims to standardize the services interface for important Grid services. Unfortunately, these standards are still evolving and the penetration in commercial and open source Grid platforms is very limited.

It can be concluded that after seven years of standardization within the OGF (the first GGF document was published in 2001), the execution management use case, i. e. the submission of a job to a computational resource, is on its way to becoming an adopted OGF standard. Despite the existence of these standards, activities like Grid Interoperability Now (GIN) [8] demonstrate that it is difficult to interoperate between existing Grid middleware platforms. The current production Grids only implement a small subset of the OGSA vision. Many of the OGSA ideas [131, 125], such as the vision of the Grid as a self-managing, fault tolerant and autonomic infrastructure, are far away from being standardized or adopted in Grid platforms.

Various Grid middleware platforms with different levels of standard support exist. Migol heavily relies on existing Globus services to provide access to the Grid. However, an adoption of OGSA standard services should be straightforward in the future. However, these platforms mainly focus on the provisioning of lower-level primitives like job-launching, information services, security and file transfer. This particularly also applies to Globus. An autonomic Grid middleware must provide more enhance autonomic features to support the reliable execution of applications in the Grid. This includes e. g. application monitoring, checkpointing, job recovery and migration.

Cloud computing gained a considerable attention mainly due to the simplicity and high-level of abstraction provided by Cloud services. While Grids usually expose a wide set of available semantics, Clouds usually provide a very narrow interface to resources. With this simplicity Cloud interfaces to Grids could foster the penetration of Grids. However, Clouds are not a silver bullet - reliability is also a desirable property for these systems. Several outages of the Amazon Cloud offering, e. g. in February 2008 [69], have shown that there is still room for improvement. Further, current Cloud services do not address interoperability or security issues sufficient.

"Compute resources are not electrons."

Rich Wolski

4 Grid Application Scenarios

This chapter discusses different programming models for Grid applications. Not all applications are equally well suited for distributed infrastructures. Section 4.1 presents an attempt to classify different Grid programming paradigms. Based on this classification, different programming frameworks for Grids are presented in section 4.2.

Having given a general overview of Grid programming abstractions, section 4.3 describes several example applications, which are used in section 4.4 to derive relevant usage scenarios for an autonomic Grid middleware.

4.1 Application Classification

Different classifications for Grid applications have been proposed in literature [20, 204]. Figure 4.1 gives an overview of several programing paradigms. The way applications can utilize Grids depends to a great extend on the communication pattern, which is required to solve the application problem. Two kinds of applications can be observed:

- *Loosely coupled* applications mostly consist of a set of independent tasks that are executed across thousands of nodes in a cluster or Grid. A subcategory of loosely coupled applications are embarrassingly parallel applications also referred to as task farming. Such applications do not require any synchronization or coordination. More coupled are master/worker applications: these applications require a master to distribute tasks, to coordinate interactions, and to collect results. Despite the simple nature, many problems can be modelled using task farming, e. g. parameter studies found in many sciences or Monte Carlo simulations.

- *Tightly coupled* simulation are communication intensive and require a regular synchronization, which usually involves the exchange of data. These applications are also

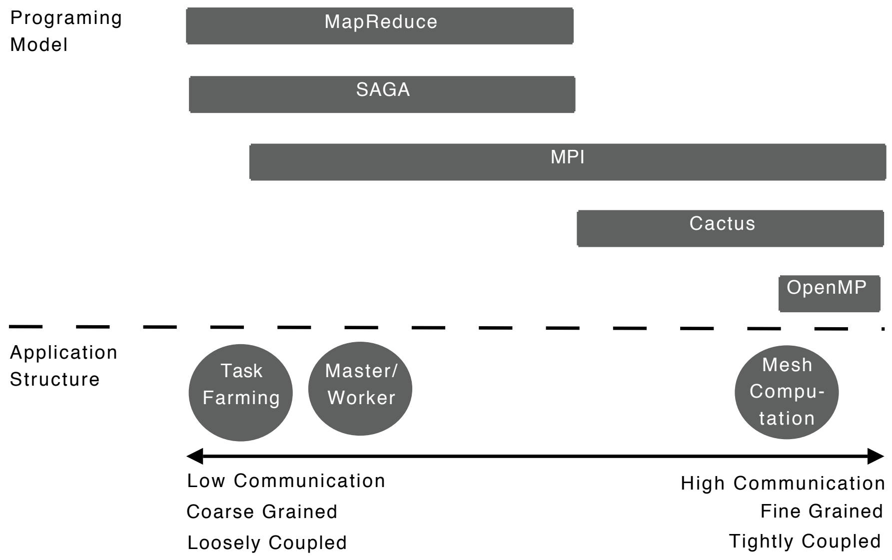


Figure 4.1: Distributed Programming Models

considered latency-bound. Examples for tightly coupled simulations are linear algebra solvers, mesh computations also referred to as grid computation, or N-body codes [33]. Common programming models used are OpenMP [243] and MPI [231].

In addition, hybrid models are often used, e. g. when conducting a parameter study with a MPI application.

Although both task farming as well as coupled applications have been successfully deployed in Grids [19], tightly coupled applications are mostly run on single clusters or supercomputers [219]. Still coupled applications can strongly benefit from Grids:

- The waiting time for resources can be minimized by running the application on a suitable remote site of the Grid.
- In case of the failure of a resource, the application can be migrated to another resource.

In the following sections different programming models and frameworks, which are commonly used to build Grid applications, are explored. After an analysis of the application landscape, we will highlight advanced Grid use cases, which should be supported by a Grid infrastructure.

4.2 Programming Models and Frameworks

Several frameworks and programming models for the development of distributed Grid and cluster applications have been developed. This sections aims to highlight some important frameworks.

4.2.1 Message Passing Interface

A widespread programming model for parallel applications is message passing. In this model processes exchange information using messages. The *Message Passing Interface (MPI)* [231, 232] provides a standardized programming interface, which can be used across different parallel computing systems. Various implementations of the MPI standard exist, the most widely used are MPICH2 [233] and OpenMPI [133].

MPI is commonly used for coupled, but also for loosely coupled task farming applications. For example, many mesh computations, such as the n-body code Gadget [293], utilizes MPI for the coordination of the processes, the exchange of ghostzones and parallel IO.

MPICH-G2 [183] supports the the development of Grid-enabled MPI applications that utilize multiple Grid resources. The individual Grid resources are accessed via the Globus Toolkit. By introduction of special communicators the framework allows the tuning of the application with respect to intra- and inter-cluster communication. It has been demonstrated that applications can be developed efficiently using this mechanism [19]. But, MPICH-G2 has different limitations:

- MPICH-G2 only supports the Globus Toolkit 2.
- The development of code that efficiently scales across multiple clusters is very complex.
- MPICH-G2 cannot deal with private IP addresses.

The successor of MPICH-G2, MPIg is currently in development[308, 255]. MPIg is based on MPICH2 [233] and relies on the GT4 WS GRAM for co-allocation.

4.2.2 Task Farming and Master-Worker Frameworks

Several classes of applications, which are well suited for loosely coupled Grids, exists; Arguably the best known and most commonly used ones are task farming applications. As described, task farming can be applied to many problems, such as parameter studies or Monte Carlo simulations. These applications depend on repeated random trials to generate statistical results. Different frameworks that exploit the task farming pattern have been developed for clusters and Grids. For Grid computing a variety of task farming frameworks exist, e. g. Nimrod-G [67] and Condor-G [132]. Condor-G can utilize local Condor resources as well as Globus resources for tasks. Also, the SAGA API provides a good abstractions for implementing task farming applications [323].

4.2.3 MapReduce

MapReduce [102] is a framework to parallelize the processing of large data, e. g. for building a search machine index of the entire web, for machine learning [82] or for finding areas in DNA/protein sequences as done by BLAST [33]. The model of MapReduce is similar to the master/worker model, but introduces an additional synchronization phase. A MapReduce program consists of two phases:

1. Iterate over a set of input data to compute intermediate key/values pairs (Map).

2. Aggregate all intermediate key/values (Reduce).

The MapReduce infrastructure is responsible for the execution of the respective mapping and reducing tasks. Fault tolerance is an important issue for a MapReduce simulation: Since the reduction phase cannot start unless all map tasks have been successfully finished, the infrastructure must handle the transparent re-execution of mapper tasks to ensure the progress of the program. The design of MapReduce was initially proposed by Google - with Hadoop [156] an open source MapReduce implementation exists. Hadoop provides a Google File System (GFS) inspired infrastructure and the MapReduce framework itself.

4.2.4 Cactus

The *Cactus Computational Toolkit* [142] is a framework for high performance scientific computing. Cactus consists of a central core, the flesh and various modules, the so called thorns. The flesh serves as module manager, which orchestrates thorns by scheduling thorn routines and passing data between thorns.

Grid applications can benefit from a wide range of existing thorns. For example, Cactus can automatically handle the parallelization of mesh computations – the framework takes care of all required activities, i. e. the decomposition as well as the management of data exchanges. For communication between nodes, Cactus relies on MPI. Further, Cactus provides several thorns for steering and monitoring of applications using HTTP and checkpointing. A wide range of Cactus applications, e. g. in the area of numerical relativity as well as in other scientific areas such as oil drilling simulations, exist.

4.3 Grid Applications

After this brief overview of programming paradigms and frameworks, in the following a set of scientific example applications is presented.

4.3.1 Cellular Automaton: Game of Life

The cellular automaton [2, 237] is a simple MPI-based mesh computation simulation of Conway's Game of Life [137]. Cellular automata use a discrete mesh of cells (also referred to as grid (with small caps)). Each cell has a state - the set of all states is finite. Depending on the problem the mesh can expand to multiple dimension. A simulation starts with an initial configuration of cell states. At every step of the simulation, the new state of the mesh is determined based on a set of rules. These rules usually take the state of the neighboring cells as input. The state of all cells changes synchronously.

Conway's game of life uses the cellular automaton abstraction to study the evolutional behaviour of living organisms. The game of life uses a two dimensional mesh. Each cell represents an organisms. Using carefully chosen rules, organisms are born, survive or die. But,

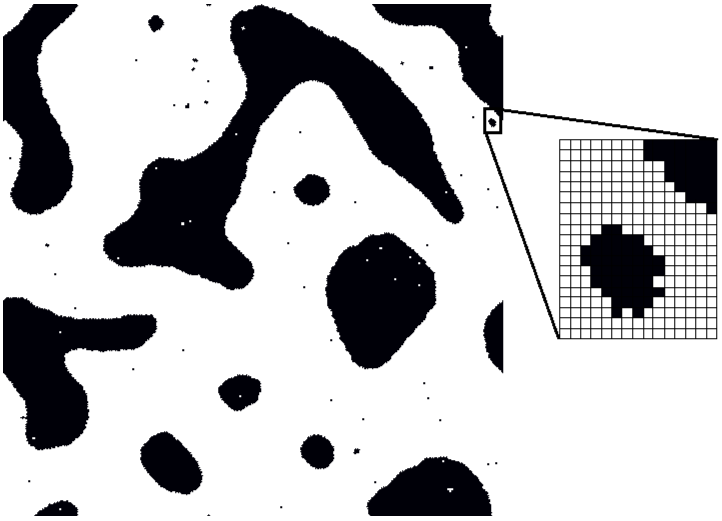


Figure 4.2: Cellular Automation Simulation Results [276]

cellular automata are also used to solve various other scientific problems, e. g. to model self-organizing biological systems or to simulate diffusion problems [328, 193].

Figure 4.2 show the state of the cellular automaton after a simulation run. Each cell of the cellular automaton is represented as pixel (either being black or white). In each iteration step the new state of a cell is determined based on states of the neighboring cells. The Cellular Automaton supports checkpointing on application-level as well as the restart of simulations.

The game of life application has been parallelized using MPI. The runtime of the application depends on

1. the size of the mesh.
2. the number of iterations, and
3. the speed of the machine.

On a 8 node Infiniband cluster with 2.33 GHz Intel CPU, an iteration of a 4096 x 1000 cell grid took about 1.5 msec [281]. Even for this small problem size the study of one million iterations requires a runtime of more than a day. Thus, the support for checkpoint-restart fault tolerance is essential. The checkpoint size solely depends on the grid size: for 16 cells a stable storage of 4 bytes is required. The checkpoint size of the above mentioned problem is about 1 Mbyte.

4.3.2 AMIGA

AMIGA [188] is a grid-based n-body code, which simulates the universe from the big bang until now. The simulation calculates the inter-particular forces due to particle interactions using Poisson's equation. Table 4.1 gives an overview of possible problem sizes with respect to resource requirements, checkpoint sizes and runtime. The runtime directly correlates to the problem size. Larger AMIGA problems can have runtimes in the magnitude of weeks. For a simulation the checkpoint size depends on the size of the problem. For each particle N, the position (x, y, z) as well as the velocities (vx, vy, vz) and mass (m) must be stored:

$$checkpoint\_size = 7 \cdot N \cdot 4\ bytes$$

Problem Size	Resources	Runtime	Checkpoint Size
128^3 Particles	2 GB/1 CPU	1 week	56 MByte
256^3 Particles	4 GB/1 CPU	>2 weeks	448 MByte

Table 4.1: AMIGA Runtime and Checkpoint Sizes

A checkpoint is written periodically according to a specified interval. In addition, multiple snapshots of the simulation can be kept for a later analysis. AMIGA is currently a sequential code, i. e. at maximum one node can be used. Due to this restriction, the code is currently limited to problems with 256^3 particles.

Amiga is often used to conduct task farming experiments. A common scenario is the re-simulation of smaller areas with a higher resolution after a large run. Further, multiple AMIGA runs with different input parameters are used to perform statistical analyses [189].

4.3.3 Replica Exchange Molecular Dynamics (REMD)

Replica Exchange Molecular Dynamics (REMD) [159, 301] simulations are used to understand important physical phenomena – ranging from protein folding dynamics to conformational configuration changes. REMD are essentially loosely-coupled, but with some coupling between the tasks. Each task is represented by a NAMD [257] simulation run. Occasionally, an exchange between pairs of replica tasks is required. The pairing of replicas is also not a constant and is dynamically determined. However, once the pair of replicas has been established, a delay or loss of one replica will stall the other replica. In the worst case, a single failing task can render the entire computation worthless. Thus, fault tolerance of the application must be considered.

4.3.4 Cactus Wave Simulations

The wave simulation is an example application provided by the Cactus framework. It solves the wave equation by finite differencing. Cactus handles the parallelization as well as the checkpointing of the application transparently. Similar to other mesh computation as the cellular automaton the length of the simulation depends on the size of the mesh, the number of iterations and the speed of the machine. The application is used to evaluate the requirements of Cactus applications in general with respect to an autonomic Grid infrastructure.

4.4 Grid Usage Scenarios and Requirements

Running the described applications in a Grid requires a sophisticated software infrastructure. The infrastructure should allow domain scientists to focus on their primary research problems

– low-level management of jobs and data should be handled transparently by the middleware. In this section we describe advanced Grid usage scenarios and their requirements with respect to a Grid middleware.

4.4.1 Job Brokering

Grid applications in general orchestrate hundreds of different heterogeneous compute, storage and network resources. The execution of computational jobs and job brokering on these resources is an essential use case required for all applications in section 4.3. The significance of this scenario is emphasized by the various use case documents of the OGF [125, 224] and use cases of Grid projects such as the AstroGrid-D [34].

Contrary to cluster systems, a Grid consists of many independent, heterogeneous resources, which are part of different organizations. The integration of those resources is a core task of the Grid middleware. While the initial challenge of executing jobs in the Grid is well understood and partially standardized with OGSA-BES, efficient meta-scheduling, i. e. the scheduling across multiple shared resources, in a Grid is still subject to research [234].

The job broker is the component in the Grid, which is responsible for resource discovery, selection and allocation. Applications are started through the Job Broker Service, which finds the most suitable resource for the application, deploys, and starts it. The broker service should support the direct execution of jobs on a user selected resource as well as an automatic resource selection. In this case, the broker must locate and select a resource taking into account the available nodes, CPUs, disk spaces, but also factors such as the transfer time of the required files, the queue time, the estimated runtime of the job and fairshare. Before the start of the application, the Job Broker stages the necessary libraries and data files automatically.

Another feature, which is demanded by the scenarios presented in section 4.3 as well as by several OGSA use cases [125] are advance reservations. Advance reservations increase the predictability of application runs and support the usage of these applications in deadline driven scenarios. Further, advance reservations allow brokers and users to obtain execution guarantees from local resources without requiring detailed knowledge of current and future workloads or of the resource owner's policies. It enhances the predictability of the system and enables co-allocation, i. e. the allocation of multiple resources at the same time. For example, many scientific workflows require the management of multiple dependent compute and data transfer jobs and benefit from the predictability guarantees of advance reservation. Further, deadline driven applications such as severe weather and hurricane simulations demand the co-allocation of different resources as well as guaranteed completion times [58].

To ensure the successful execution of long-running jobs, the middleware should support the reliability of Grid applications. This aspect is further discussed in the next section.

4.4.2 Checkpoint Replication

A critical aspect for an application recovery is the availability of the current application state, which is commonly stored in checkpoints: the application can only be restarted from the last known state, if the checkpoint is available. Otherwise,the application must be started over from the beginning. Especially for long-running applications, this could mean the loss of weeks or months of computations. In a Grid scenario it is very likely that locally stored checkpoints are not available for a recovery, e. g. due to a file server crash or a network partition. A reliable Grid middleware must therefore support the automatic replication of checkpoints. In addition to fault tolerance, replicas are often used to optimize access to files, e. g. by load balancing read requests.

4.4.3 Application Recovery and Migration

The ability to recover applications is critical in particular for long-running applications, such as AMIGA or other astrophysical codes. Failure detection is an essential pre-requisite for application recovery. The Grid infrastructure must be able to detect the failure of an application. After a failure is detected, the Grid infrastructure must be able to react accordingly, e. g. by restarting or *migrating* the application. A migration is referred as act of transferring an application between two machines. Various events can trigger a restart or migration.

Automatic Failure Recovery

A Grid being highly dynamic cannot be assumed to be error-free: planned and unplanned maintenance of networks, machines, and software, or sudden failures pose severe problems for long-running applications. The ability to restart failed applications on another resource, providing failure transparency, is the key for supporting fault tolerance of MPI applications. The middleware must monitor all running applications. In case an application or a necessary resource becomes unavailable due to a software, hardware, or network failure, a recovery or migration is initiated.

Performance-initiated Migration

In a Grid it is often required to relocate an already running application process from one resource to another resource. An application or user can trigger a migration, e. g. if the requirements of an application exceed the offered computing time on a site. In case of opportunistic migration a relocation is initiated if a faster or cheaper compute capacity becomes available during the execution or if a higher throughput can be achieved if the job is migrated.

Another motivation are the runtime limits often imposed on applications. For example, if the queue walltime at the LRMS is limited to four days, an application with a runtime of 7 days must be restarted at least once.

Administrator-initiated Migration

Grid machines usually require some maintenance and down-time. Local administrators can access all processes and applications and are therefore able to terminate them, if necessary. But, it would be more desirable to allow the administrator to manually relocate all running applications to another Grid site to prevent data losses.

4.4.4 Summary and Discussion

A variety of Grid applications and programming models that utilize different levels of abstractions have emerged. This chapter presented an overview about different Grid programing models. Based on these models, we derived different representative usage scenarios. This work aims to address the fault tolerance of long-running high performance applications from tightly coupled N-body codes to loosely coupled master/worker applications. Loosely coupled applications can easily be partitioned into independent tasks and are therefore the natural programing model for Grids. In contrast, for applications that require extensive interprocess communication, such as mesh computations, the communication overhead is likely to negate the performance gain. Thus, parallel applications, such as MPI or Cactus applications, are usually deployed on single clusters or HPC machines. However, long-running MPI codes running on thousands of nodes are highly likely to fail and thus require a resilient execution environment.

Based on the presented applications, a set of middleware usage scenarios was presented. While the requirements are based on the presented MPI and Cactus applications, the Migol framework presented in chapter 6 is not limited to these applications. In general, many applications can benefit from an autonomic middleware, which supports the automatic monitoring and recovery of Grid applications. These features can significantly reduce the application downtimes while improving the utilization of resources at the same time.

5 Fault Tolerant MPI

To ensure the fault tolerant execution of distributed applications, the application must be designed with respect to fault tolerance. Since the network is a highly unreliable component, the fault tolerance of the communication framework is important. A standard framework, which is used by many computational applications for message passing between cluster and grid nodes is MPI (Message Passing Interface) [232]. To enable application developers to write fault tolerant programs, the message passing framework should expose a defined fault model, which provides support for detecting and handling communication and node failures.

Section 5.1 briefly analyses the fault propagation chain in MPI applications. Since most MPI implementations use TCP as transport protocol, the failure detection algorithm of TCP is described in section 5.2. The fault tolerance aspects covered by the MPI standard are discussed in section 5.3. Section 5.4 evaluates the failure behavior of MPICH2 [233] and OpenMPI [133].

5.1 Fault Handling in a Layered Architecture

The execution of an MPI application in a Grid requires an interaction between various subsystems. To handle failures of lower level services those services must expose a defined failure semantic. If a higher level server can provide a service despite a failure of a lower level service, it masks the failure. In general, it is desirable to mask a failure on the lowest possible level. In a MPI application failures can occur and can be masked on several levels (cmp. [90]):

- **Operating system level:** Many failures can be masked by the operating system, e. g. transient network faults are handled by the TCP/IP stack.
- **MPI level**: Although not demanded by the MPI standard (see section 5.3), the MPI implementation should provide additional fault detection and handling mechanisms to handle e. g. the permanent failure of a node. An example for a resilient MPI implementation is MPICH-V [60].

- **Application level**: If failure masking is not possible within MPI, the failure must be propagated via an exception to the application. The application can then handle the error, e. g. by restarting itself.
- **Application external**: If all masking attempts within the MPI application fail, a crash failure occurs. An external monitoring service could then detect the failure and restart the application either on the same cluster or on another Grid resource. This requires a possibility to monitor the application externally.

Depending on the level on which a fault occurs, different survival techniques can be applied. For example, transient network faults can be handled by using a retransmission mechanism or a reliable transport protocol such as TCP. Permanent failures, such as a node failure require more sophisticated recovery strategies, e. g. checkpoint/restart.

5.2 TCP/IP Failure Detection

To deal with failures on MPI- or on application-level, the communication infrastructure must be able to detect errors and to notify the application layer. In general, Grid-enabled MPI deployments use TCP/IP [261] as transport layer protocol. Thus, the reliability of MPI depends to a great extend on TCP. TCP uses the partial asynchronous distributed fault model (cmp. [51]), i. e. it is assumed that lower and upper bounds for events, such as roundtrip latencies, are known. The protocol was designed to handle transient network faults, such as lost packets or the reception of packets in the wrong order.

TCP uses delayed acknowledgements and dynamic timeouts to detect failures after sending data to a remote host. After a fault is detected, a retransmission is initiated. The retransmission timeout is calculated using the smoothed roundtrip time and the standard deviation of the roundtrip time. After each timer expiry, the timeout times are doubled (exponential backoff). The exact retransmission and backoff behavior is implementation dependent. This leads to a high variance in the failure detection time of the different implementations: the time varies from 100 s in FreeBSD (the minimum value demanded by the RFC 2988) to about 12 minutes in Linux (see Figure 5.1).

Since TCP does not require any transmission over an established connection, the detection of a crashed system during an idle connection is difficult. Although most TCP implementations support a keep-alive mechanism, this feature is not part of the TCP RFC and therefore not activated by default. If a keep-alive is used, RFC 1122 [61] requires that the keep-alive default value must be at least two hours, which makes it unsuitable for rapid failure detections. With an activated keep-alive, a failure detection time of about 2 hours and 10 minutes was observed (see Figure 5.1). Most TCP implementations allow a tuning of the keep-alive behavior. Unfortunately, the tuning parameters are not standardized.

Finally, it can be concluded that TCP is less suitable for rapid failure detections. To survive short network interruptions, TCPs keeps connections alive as long as possible. A fault tolerant MPI implementation must be aware of this shortcoming and should implement an additional error detection mechanism on top of TCP.

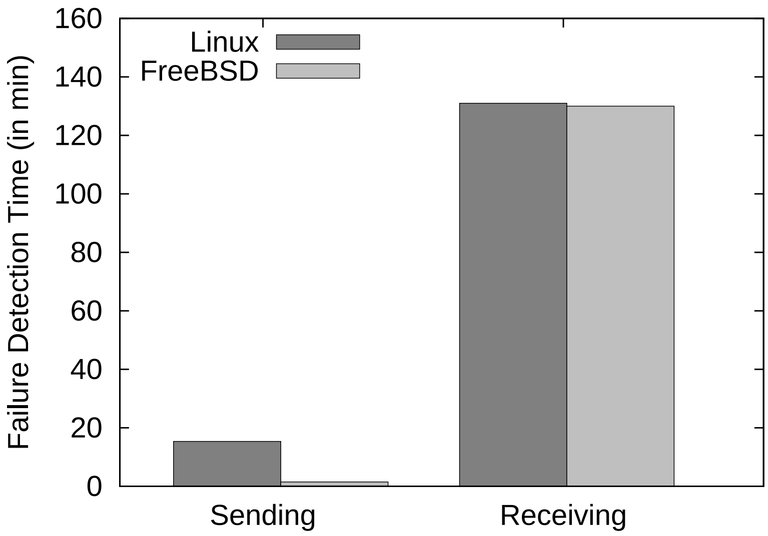


Figure 5.1: TCP Fault Detection Times

5.3 Fault Tolerance within the MPI-Standard

In terms of application transparency the handling of faults within the MPI implementation is desirable. However, the MPI standard demands implementations only to provide basic support for failure detection and handling. The only guarantee a MPI implementation must provide is reliable messaging, i. e. omission faults during network transmissions must be handled by MPI. This can be achieved either by using a custom retransmission mechanism or a reliable transport protocol such as TCP.

If a fault cannot be handled by an MPI implementation, the fault should be propagated to the application. MPI forwards errors to an application using either a return value or an error handler. Unfortunately, the standard specifies only a small set of errors that must be returned to the application, e. g. program errors if a message is sent to a wrong node. Implementations are allowed to extend this set. Since no error codes for crash faults are defined, those faults will mostly be categorized as `MPI_ERR_UNKNOWN` or `MPI_ERR_INTERN`, depending on the MPI implementation. In practice, most MPI implementations will, at best, forward failures detected by the transport layer, or at worst, crash or hang without returning to the caller (see section 5.4). If an MPI function returns despite of an error, the MPI standard does not demand further MPI operations to work. As a consequence, reconfiguration possibilities are very limited: in most cases the only option is the complete restart of the application.

In following, the fault behavior of MPICH2 and OpenMPI is analysed.

	MPICH2	OpenMPI
Fault detected and message printed	yes	yes
Return of error code or call to error handler:	no	no
Point to point communication to non-failed nodes possible	no	no
MPI_Finalize working	no	no
mpiexec terminates	no	no

Table 5.1: Failure Behavior when Sending to a Failed Node

5.4 Experiences with a Fault Tolerant Master-Worker Application

In general, fault tolerant Grid applications are required to provide an application-level recovery routine. Different recovery scenarios must be considered for different application categories: In a master-worker application, the worker nodes in general maintain only a small state consisting of the latest work package received. Assuming the worker does not modify any global state, the crash of a worker can be regarded as an omission failure for the master. A master can deal with this failure by re-executing the work package on another node. In contrast, the failure of a master or a process in a mesh computation is more severe. Mesh computations are comprised of a complex state, which is distributed across all nodes. For mesh computations, a checkpoint-based recovery is often the only option.

To evaluate the failure behavior of OpenMPI [133] and MPICH2 [233], a simple master-worker application, which does not require any collective operations and solely uses point-to-point communication between master and workers, was developed. In the following the trivial case of handling a fail-stop worker failure is examined. The experiments were conducted on a Linux cluster (Kernel 2.6.21) using MPICH2 1.0.5 with the default `ch3:sock` channel device and OpenMPI 1.2.3 with the TCP BTL.

The following criteria were considered in the evaluation (see Table 5.1). The master must be able to detect the failure of a worker node. That means that the MPI implementation must detect and propagate the fault – `MPI_Send` should return and the application should not abort. If `MPI_Send` successfully notifies the master process, the master handles the failure by re-submission of the respective work package. This implies that further point-to-point communication to non-failed nodes is possible. As the MPI standard does not demand this feature, it depends on the MPI implementation whether further communication is possible. At last, `MPI_Finalize` and `mpiexec` must return to ensure a clean program termination.

Table 5.1 summarizes the results of the experiments with MPICH2 and OpenMPI. The fault detection time of the master correlates with the time required by TCP to detect a node failure. Both implementations do not support a special fault detection mechanism and solely rely on TCP for error detection. Also, they do not use the TCP keep-alives mechanism likely due to interoperability issues. However, that means that a process is not able to detect an error while waiting for a message.

The crash fault is detected by MPICH2 and OpenMPI. However, neither did `MPI_Send` return (using `MPI_ERRORS_RETURN`) nor was the custom error handler called. As a consequence of the fault, the test application did not terminate correctly and remained suspended. Even worse, `mpiexec` left a zombie process on each cluster node. A reason for this behavior is the collective property of `MPI_Finalize`, which requires that all processes within `MPI_COMM_WORLD` must be still alive.

In addition, the experiments indicate that the failure behavior varies slightly with different external factors, such as the network stack, the operating system, and the state of the TCP connection. For example, `MPI_Send` returned under certain rare conditions, e. g. if the failure occurs before the establishment of a TCP connection in the case of OpenMPI.

Finally, it can be concluded that no application-level recovery from within the application is possible. Especially for a master-worker application, which can easily recover from a worker failure by just re-executing the work package, it is not very efficient to restart the entire application.

5.5 Summary and Discussion

The MPI standard leaves fault tolerance mainly to its implementations. However, OpenMPI and MPICH2 currently lack the ability to efficiently detect and propagate errors showing a very inconsistent failure behavior. Only if errors are propagated consistently to the application, an application-level recovery routine can be applied. For example, the master can re-submit the work package to another worker. Of course, it must be guaranteed that point-to-point communication to non-failed nodes is still working.

In case of a fatal error, it must be ensured that the application gracefully terminates on all nodes without leaving any zombie processes. A clean program termination allows monitors, such as provided by PBSPro or LSF, to detect the failure of the application based on the return value. The detector can then initiate a restart.

With the current MPI semantic, communicators become invalid as soon as one process fails. FT-MPI [115] addresses this issue and permits applications to survive a failure without requiring the re-spawning of all processes (see section 10.1). To achieve this, FT-MPI communicators can be reconfigured in case a member fails. However, these recovery features require the extension of the MPI standard. But, such capabilities are a key requirement for the development of fault tolerant MPI applications. Due to the lack of support of these features in the MPI standard, an availability of such features, although provided by implementations such as FT-MPI, on Grid resources cannot be assumed.

Despite the desirability of an efficient application internal error detection and recovery routine, external monitoring and checkpoint-based recovery is in most cases the only option for providing fault tolerance to MPI applications in the Grid. The Migol framework presented in the next chapter offers these features and supports an automatic recovery of MPI applications.

"The unavoidable price of reliability is simplicity."

C. Hoare

6 Migol: A Fault Tolerant Grid Architecture

Migol aims to provide an autonomic infrastructure for supporting the automatic recovery of computational Grid applications. With this functionality, Migol addresses the short comings of application frameworks such as MPI and the Globus Grid middleware. Both do not sufficiently support fault tolerance – especially for long-running and critical applications this lack of resilience renders Grid infrastructures unusable. This chapter presents the architecture and design of the Migol infrastructure, a services-oriented, autonomic middleware for supporting the fault tolerance of Grid applications. These services are able to guarantee the correct and reliable execution of Grid applications even in the presence of failures.

Building a reliable system from many potential unreliable components is a difficult task. Based on the usage scenarios presented in chapter 4, we define the functional components and interactions necessary to provide an autonomic Grid infrastructure that can adapt silently and automatically to changing environments. The architecture of Migol is inspired by the OGSA service model and defines services for

- the collection and publication of information,
- the management of computational and data resources,
- the replication of data, and
- monitoring and recovering applications and services.

All Migol services are composed of modular components to allow an optimal re-use of modules and to achieve a maximum of robustness. Existing services, components and libraries, especially those of the underlying Globus framework are re-used if possible. Further, all services provide fault tolerance support, e. g. by supporting an automatic recovery in case of a container crash.

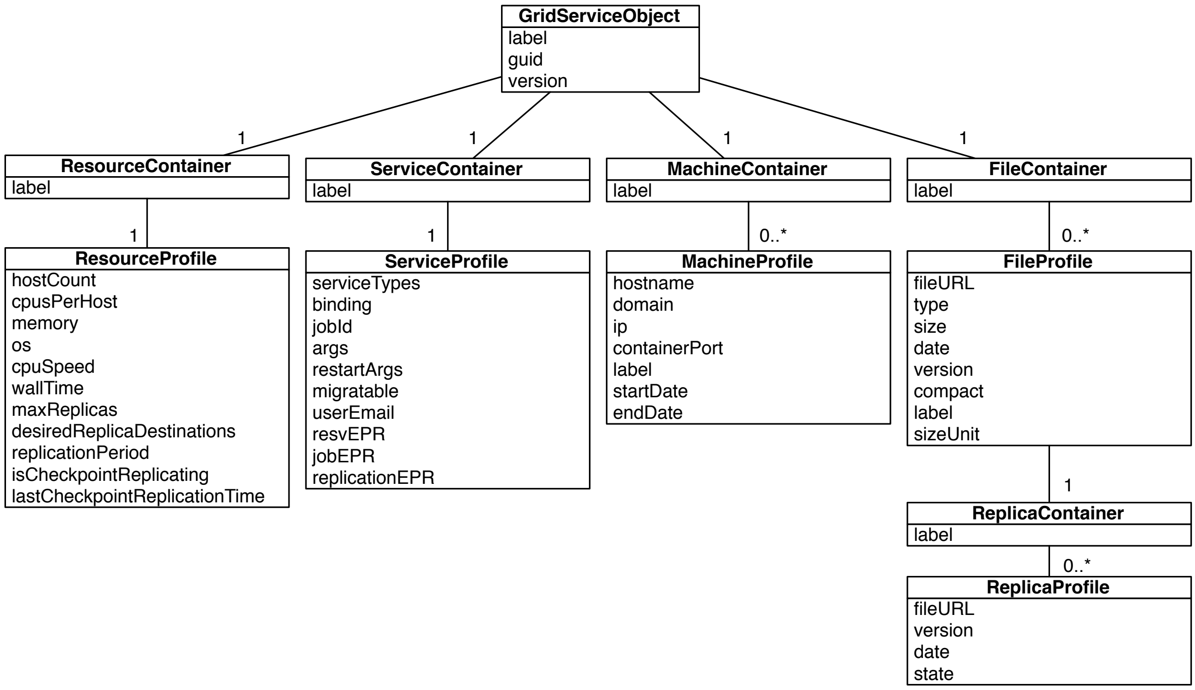


Figure 6.1: Grid Service Object Model

This chapter starts with the description of the Grid Service Object data model, the fundament of Migol, in section 6.1. All services follow the resource-oriented architectural style defined by WSRF. In sections 6.2 to 6.4 these services are described in detail. Section 6.7 will extensively describe how Grid applications can be integrated into the Migol framework. Security and concurrency issues are briefly discussed in sections 6.8 and 6.9.

6.1 Grid Objects: Describing Grids and Grid Application

Fundamental part of Migol is the *Grid Object Description Language (GODsL)*, which defines a generic and extensible information model for describing different aspects of the Grid, e. g. hardware, applications, services or data [202]. With GODsL resource requirements as well as current resource states can be expressed. The language enables an automatic mapping of applications onto available hardware and software resources. Migol uses a subset of GODsL to define Grid Service Objects based on language independent XML schema definitions.

Figure 6.1 shows a Grid Service Object (GSO). A GSO consists of four sub-objects, so called containers: a resource, service, machine and file container, which again hold one or more profiles:

- The resource profile describes the resource requirements of an application.
- The service profile is used to describe the functions of an application. It contains all

information necessary to start respectively restart an application.

- Machine profiles store a history log of all used machines.
- File profiles are used to describe files and directories, e. g. application binaries and checkpoints. The replica container and profiles are used to store the location of file replicas.

Further, each GSO is associated with a global unique identifier, a version and the current state of an application. The state of an application can be, e. g. active, pending, inactive, migrating or done.

Existing resource description languages such as the Globus Resource Specification Language (RSL) or the Job Submission Description Language (JSDL) [30] are not sufficient for this purpose. The JSDL standard is more normative than the proposed Grid Service Objects model. For example, JSDL provides a standardized way to differentiate between different CPU architectures, operating systems, or MPI versions. GSOs can also express this constraints, but in a non-interoperable way.

The main restriction of JSDL is its limitation to the description of computational jobs requirements. To enable a migration or recovery of an application additional information is necessary, e. g. the Migration Service must know where the most up-to-date checkpoint is stored and how it can be accessed. If checkpoints are stored replicated, the nearest file should be selected. Further, Grid Service Objects are used to save current application states and a history log of used machines.

The Grid Service Object model is used as common information model by all Migol services. Figure 6.2 gives an overview of all Migol services. Beginning with the information services, all Migol services are discusses in the next sections.

6.2 Information Services

A basic requirement of an autonomic infrastructure is the dynamic discovery and monitoring of resources and applications. The availability of various cluster and node information, such as the number of processors, processor types, operating system versions, available storage, memory, CPU load, local resource management policies, network information etc. is the pre-requisite for selecting a location of a job. Since the quality of the scheduling procedure essentially depends on the available resource information, the gathering of resource data as well as the provisioning of this information is very important. The following section describes how Migol manages application and system information.

6.2.1 Application Information

Accurate and detailed information about applications is essential to select an appropriate resource for an application or to automatically manage an application recovery. A standard framework for the discovery of Web services is the *Universal Description, Discovery and*

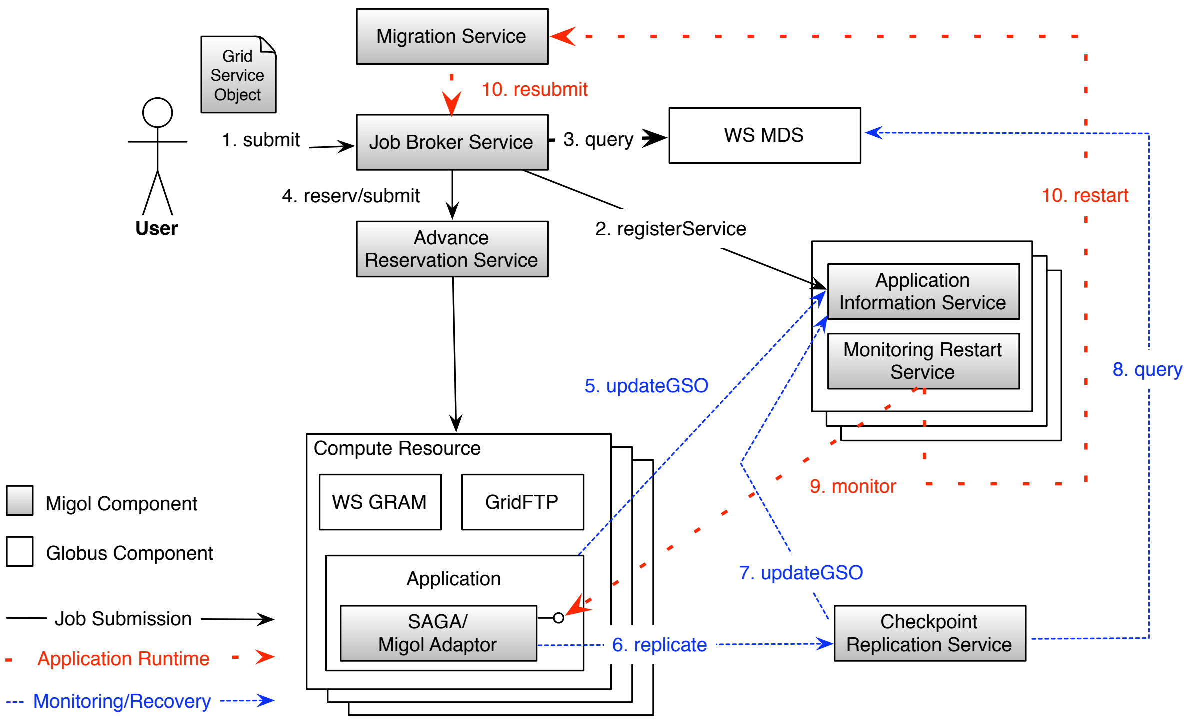


Figure 6.2: Migol Service Architecture: Migol provides services for supporting the fault tolerance of Grid applications. Applications that are managed by Migol are transparently monitored and recovered in case of failures.

Integration framework (UDDI) [314]. But UDDI suffers from different limitations: Due to its lack of explicit typed data, UDDI can only offer a limited query model based on strings. Further, UDDI is not able to handle the expiration of data, which is a key requirement in a dynamic Grid [47].

UDDI provides basic fault tolerance by supporting the replication of registries. Unfortunately, the replication model has a major disadvantage: Changes can only be performed at the node, which the service originally registered to. If this host becomes unreachable, no changes to the service are possible. This behavior is not acceptable for a Grid with rapidly changing information. Thus, Migol relies on the *Application Information Service (AIS)* for managing application data. As metadata scheme Migol uses the Grid Service Object description for specifying application requirements and states.

The Application Information Service (AIS) is the core registry service of Migol. It stores information about all running Grid applications. Before the start of an application the user must register the Grid Service Object at the AIS. If a job is submitted through the Job Broker Service (JBS) (see section 6.3) it is registered automatically (see message 2 in Figure 6.2). The Grid Service Object contains all relevant application information, for example the resource requirements, the location, and the state of the application. The AIS enables the user to locate his application after a job start or a migration using a global unique identifier. Since the AIS is a core component of the Grid, its reliability must be ensured. To avoid a single point of failure the AIS is replicated using a group communication protocol, which ensures virtual synchronous updates of stored Grid Service Objects on multiple Grid sites (see section 7.3).

While application metadata is in particular critical to support the automatic recovery of applications, resource information are required to discover and select suitable Grid resources for an application.

6.2.2 Resource Information

Resource information can be characterized according to the speed it changes into static and dynamic information. Static information such as the CPU clock rate or the physical memory varies more slowly, while dynamic information changes rapidly. Examples for dynamic information are CPU load and network throughput.

Migol uses the Globus WS MDS (Web Service Monitoring and Discovery Service) [286] for gathering, storing, and publishing of dynamic and static resource information. This includes information about the availability and utilization of Grid resources as well as data about the current state of the batch queuing system, which is provided by the standard WS MDS collector. This information can be enhanced using third party collectors, such as Ganglia [135] and NWS [331]. The underlying data model is based on the GLUE schema [28]. Information can be queried via a WS-ResourceProperties interface e. g. using XPATH.

Especially important for making job placement decisions is the availability of network information. This information is essential to estimate the time needed to move data from a source location to a certain resource. The Network Weather Service (NWS) [331] provides an extensive framework for the collection of bandwidth samples. Further, the NWS provides a prediction engine, which can be used to forecast the performance of networks or compute resources. Migol utilizes a collector script for publishing this information to the WS MDS. If available, Migol heavily relies on these data for selection of computational resources within the Job Broker Service (see Figure 6.2, message 3) as well as for the discovery and prioritization of checkpoint replica locations within the Checkpoint Replication Service (see Figure 6.2, message 8).

6.3 Job Management Services

Contrary to cluster systems, a Grid consists of many independent, heterogeneous resources, which are part of different organizations and thus subject to different policies. Figure 6.3 shows a typical resource management architecture for Grids. In general, computational resources are managed by a local resource management system (LRMS) like PBS [161] or LSF [340]. A Grid execution management service, such as the WS GRAM [118], provides a Grid-level interface to the LRMS. While a LRMS enforces the access policies of the resource owner at a single site, a Grid broker manages resources across different organizations.

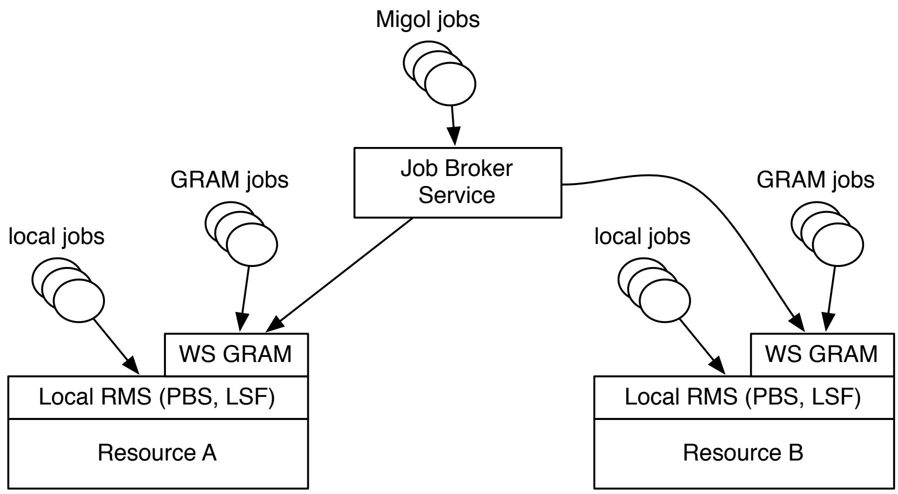


Figure 6.3: Grid Scheduling Architecture

Often, Grid brokers and LRMSs have conflicting objectives: Grid brokers desire an optimal performance for an application while LRMSs aim for an optimal throughput.

Due to site autonomy a Grid broker has no control of Grid resources, i. e. a broker has no influence on the actual start time of a job. Without local control and global knowledge about the Grid a Grid broker can only provide best effort services. But, best effort guarantees are not sufficient for supporting all Migol use cases. For example, an application migration requires a committed start time to improve placement decisions. Advance reservation is a common mean to establish stronger assurances between resource provider and user. No detailed knowledge of future workloads or of the resource owner's policies is required.

Migol significantly benefits from the availability of advance reservation. With advance reservation fail allocations in a Grid can be minimized and unnecessary file transfers in case of a migration can be avoided. In addition, advance reservation is necessary to better support co-allocation, i. e. the simultaneous start of a job on multiple resources.

In the following sections the Advance Reservation Service, which supports the negotiation of advance reservation, and the Job Broker Service, which selects a suitable resource for a user's job are presented.

6.3.1 Advance Reservation Service (ARS)

Different LRMSs, such as PBSPro [161], LSF [340], or a scheduler like Maui [171], have advance reservation capabilities. Unfortunately, these feature are neither available on Grid-level, e. g. through the GRAM service, nor sufficient with respect to their functionality.

The *Advance Reservation Service (ARS)* [175] offers APIs on top of different LRMSs, such as PBSPro, LSF, and Maui, to create, monitor, and destroy reservations and to bind jobs to existing reservations. Using a plugin architecture and a configuration API the service can be easily extended to support other LRMSs.

For each user request a LRMS specific shell script, which conducts the respective operations at the LRMS, e. g. calling `pbs_rsub` to place a PBSPro reservation, is generated by the respective plugin. The script is then executed using the WS GRAM (Fork) service of the resource. This approach can be generically applied to any WS GRAM managed compute resource. The WS GRAM ensures that all LRMS requests are securely executed using the privileges of the respective Grid user. The output of a reservation request is an Endpoint Reference (EPR) representing the reservation resource, which encapsulates the reservation ID of the LRMS. The reservation can be managed using this EPR.

Since the WS GRAM is not aware of reservations it is not possible to bind GRAM jobs to an ARS reservation. Thus, the ARS must also be able to bind and to manage jobs. For this purpose, the same approach as for reservations is used: the ARS generates a shell script which binds a job to a reservation of the LRM. The same holds for job status queries or cancellations.

Another issue is the specification of the advance reservation. Most LRMS expect the specification of an advance reservation in a very rigid form, usually a fixed time and duration. However, most users prefer to specify a time window consisting of the earliest start time, the deadline and the duration of the application (see Figure 6.4).

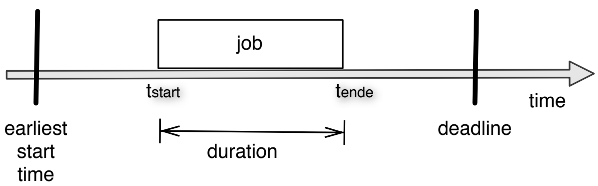


Figure 6.4: Job Scheduling Parameters

Further, it would be beneficial to query LRMSs for the earliest possible start time of a job in a specified reservation windows. Using a simple negotiation protocol as shown in Figure 6.5, a Grid broker can easily rank multiple resources by comparing the proposed start times or estimating the expected delay based on these start times. However, neither current LRMSs nor Grid brokers do not support this feature. Therefore, the ARS provides an earliest start time estimation (ESE) module, which aims to predict the earliest start time for the job taking into account:

- All current running jobs.
- All queued or suspended jobs.
- All committed reservations.

The necessary information is collected from the respective LRMS.

Some LRMSs prioritize normal batch jobs and advance reservations different. For example, PBSPro grants reservations a higher priority than regular batch jobs. This approach significantly penalizes regular jobs. In contrast, ESE enforces fairshare by ensuring that reservations and normal batch jobs are equally prioritized. The negotiation process used by the ARS is compatible with WS-Agreement and can easily be migrated to this protocol.

In the following section, the Job Broker Service, which relies on the ARS for conducting reservations and for the management of Grid jobs, is further discussed.

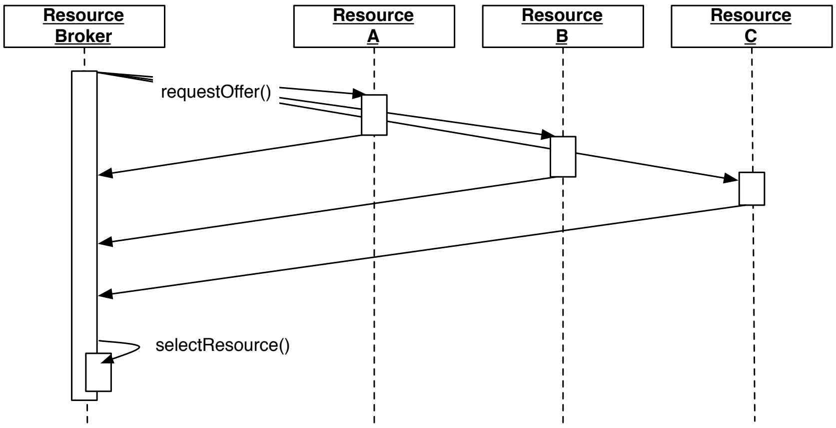


Figure 6.5: Start Time Negotiation Protocol

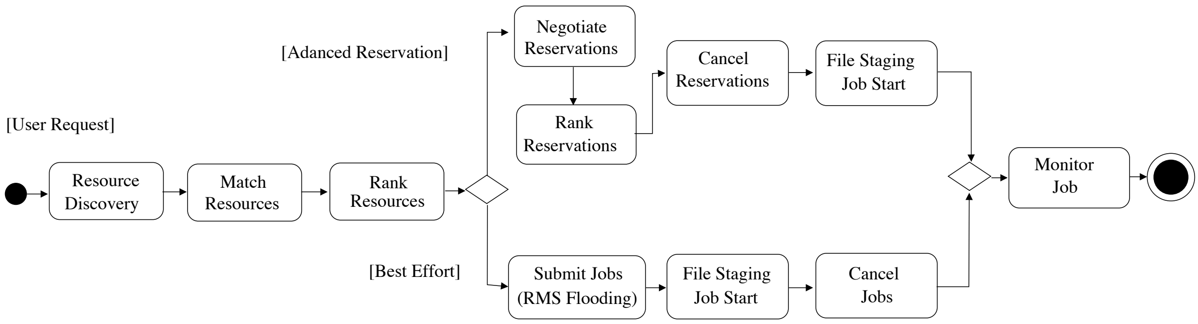


Figure 6.6: Job Broker Activity Diagram

6.3.2 Job Broker Service (JBS)

Due to the great diversity of resources in a Grid the discovery and allocation of suitable resources as well as the execution of applications is a complex task. Rather than handling the necessary steps manually by the user, these tasks can be conducted automatically by a job brokering service. With the *Job Broker Service (JBS)* Migol offers a meta-scheduling service for virtual organizations.

Figure 6.6 shows the submission process for a job. The Job Broker Service can cope with any application that is properly described by a Grid Service Object. After submission the JBS performs a resource discovery by querying the WS MDS, which returns different static and dynamic information about available Grid resources. But, it cannot be assumed that these information is complete and up-to-date. Despite this uncertainty, the Job Broker Service must select an acceptable resource for the application.

In the first step, the obtained resource information is matched against the application requirements. In the next step all resource candidates are ranked. After ranking of the resources, either the resource manager selective flooding module or advance reservation is used to determine the best site. All steps are explained in detail in the next section.

Resource Ranking

Many brokers assume that 100 percent of the extremely fine grained information needed to find an optimal resource is available, always correct, and up-to-date. This is not very likely in a dynamic Grid. Since the JBS has no control over local resources, common performance and allocation metrics are likely to fail in a Grid.

Therefore, the JBS scheduler ranks all suitable resources based upon a simple delay factor metric, which includes the application runtime, transfer time and the queuing time. Various models and techniques for estimation of application runtimes and queuing times on a respective resource have been proposed [273, 291, 109]. Unfortunately, these models are highly dependent on the application's characteristic and cannot be applied generically. For example, it is usually required to specify the amount of parallelism and scalability of an application. In the following, a more realistic Grid model where such information is not available is assumed.

Another important factor is the transfer time. Since the costs for moving checkpoint data are significant, the proximity between source and selected hosts has to be considered to avoid expensive data transfers or the selection of a difficult-to-reach or disruption-prone site. Especially, for data-intensive applications, which require large sets of input data, the only viable option is to put the computation close to the data [146].

The Job broker ranks all resources of the candidate list based on a heuristic comprising of the following factors:

- **Run time:** The CPU clock rate of a resource is used as speed factor and indicator for the runtime.
- **Waiting time:** The queue length at the LRMS and the number of requested nodes per application is used as indicator for the waiting time respectively for the reservation acceptance probability.
- **Transfer time:** If network data is available at the WS MDS, the transfer time for the necessary input files is estimated based on the current performance data.

Because of the mentioned Grid limitations, this metric is only a rough approximation. Grid brokers must be able to deal with uncertainty – no broker will be able to obtain all global information necessary to make a precise forecast. For example, the waiting time of a job is, due to insufficient information and in-transparent allocation principles of the LRMSs, hardly predictable. Further, it is nearly impossible to forecast the run time of an application solely based on the CPU clock rate.

In the following, the two resource selection algorithms that are part of the JBS are discussed, the resource manager selective flooding procedure and the advance reservation based selection schema.

Resource Selection

If the Grid environment supports advance reservation, the candidate list obtained by the ranking metric is now used to send reservation requests to the top three sites using the Advance

Reservation Service (ARS). In contrast to the generic negotiation procedure depicted in Figure 6.5), all reservation requests and the corresponding offers must be tunneled through the ARS, since start time estimation is not natively supported by the used LRMS.

All return offers are then compared based on the *shortest expected delay (SED)* [284] metric, which is calculated based on the transfer time, the start time and the specified runtime of an application. For the top resource the broker commits the reservation. All other reservations are cancelled. The JBS then plans and executes the transfer of all necessary files. Finally, the job is submitted using the assigned reservation ID through the ARS.

In the case of best effort jobs Migol uses a very simple but practical approach to minimize the waiting time. The job is dispatched to the top three sites from the candidate list – we call this approach *Resource Manager Selective Flooding (RMSF)*. As soon as one of these jobs gets active, all other jobs are cancelled. This approach avoids orphan jobs and ensures that the overhead caused by an allocation of more than necessary resources is minimized. However, there are some disadvantages: For example, since the LRMS decides when a job is started, no commitments with respect to a certain start time or deadline can be given. Thus, it is not possible to plan e. g. the transfer of large files. Therefore, Migol relies on advance reservation whenever possible.

In the following the file transfer and job start procedure is explained in detail.

Application Start and Fault Tolerance

Typically a LRMS controls access to resources by enforcing different accounting, priority, and security policies specified by the resource owner. The JBS/ARS relies on the unified interface to local resources provided by the WS GRAM [94] (see message 4 in Figure 6.2). For reservation-based jobs, the ARS is used to bind GRAM jobs to an existing reservation.

File staging is handled via the WS GRAM interface to the Reliable File Transfer Service (RFT). In general, no in-advance deployment of the application is necessary – all required files will be transferred automatically before the start of the application.

During the start of a job, failures can occur due to several reasons. Thus, it is essential to handle common failure scenarios, such as e. g. the crash of the front node running the WS GRAM service. The Job Broker Service continuously monitors the application respectively the job state at the GRAM service until it has been completely started. In case of a failure, the JBS is re-schedules the job to another resource.

With this simple mechanism, the JBS can handle failures such as:

- The failure of a compute resource, e. g. the crash of the front node, during the job start.
- The failure of an input file transfer, e. g. due to the shortage of disk space or a network failure.
- The failure of the job start due to various other reasons, such as a missing library.

After a successful start the application changes its state in the AIS to active using the Migol library (see section 6.7). The JBS can then stop monitoring the application start. During its runtime, the application is monitored by the Monitoring and Restart Service (MRS).

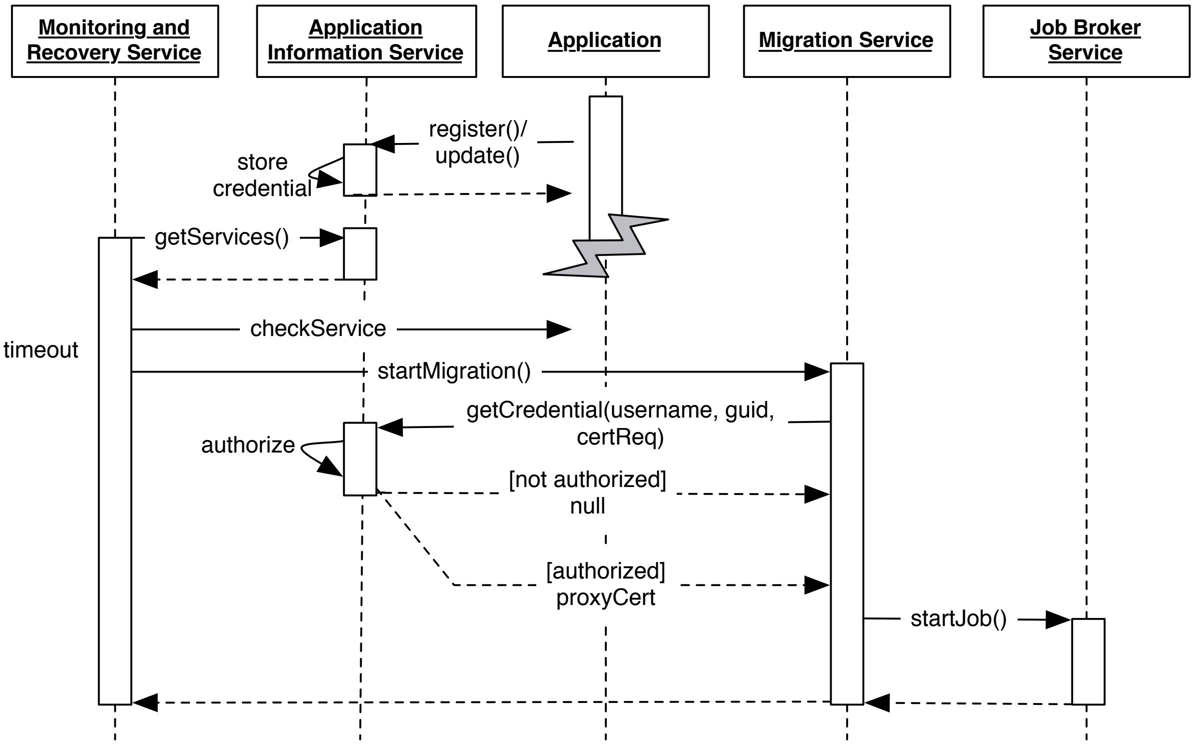


Figure 6.7: Recovery after Failure Detection

The JBS itself is designed with respect to fault tolerance. The state of all JBS resources is persisted. After a failure and the restart of the JBS, the JBS deploys a simple recovery procedure – all job resources are restored from the stable storage. In case the job has not been submitted yet, i. e. the job is in the state `unsubmitted`, the scheduler and submission procedure is restarted. Otherwise, the job state is polled from the LRMS. In case the job failed, a resubmission is triggered.

6.4 Monitoring and Restart (MRS) and Migration Service (MS)

Fault tolerance requires at least two basic mechanisms: failure detection and recovery. Figure 6.7 illustrates the Migol failure detection and recovery process. For failure detection Migol relies on the *Monitoring and Restart Service (MRS)*, which periodically checks all services registered at the AIS using the monitorable interface (see section 6.7). To distinguish between failures and slowdowns the MRS uses a configurable timeout.

A Grid usually provides sufficient structural, dynamic redundancy, i. e. usually free resources exist that can be dynamically allocated in case of a failure. If an unreachable application or network is detected, a backward recovery is initiated by restarting the application on another resource using the *Migration Service (MS)*. In addition to an automatic recovery, the MS can also be triggered by the user, the application or a system administrator.

After initiation of a migration, the MS attempts to checkpoint the target application. If the application is not available, the most up-to-date checkpoint is used. After checkpointing, the MS seeks to gracefully stop the application. In case of an automatic migration, the MS uses the delegation-on-demand protocol [157] to obtain a current credential of the user from the AIS (see also section 6.8). The application is then restarted using the JBS. On the new resource, the application process is recreated and the state is restored from the checkpoint file.

6.5 Application State Model

An Migol application can have various states during its lifetime. Figure 6.8 summarizes the state model used by the Job Broker and Migration Service. With the creation of the JBS

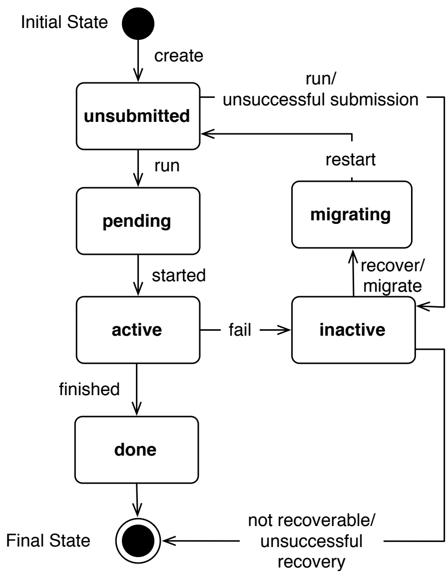


Figure 6.8: Migol's Application State Model

resource the job is assigned to the state `unsubmitted`. After completion of the scheduling algorithm and the successful submission of a job to a LRMS queue the job state changes to `pending`. In case an application cannot be submitted successfully, the JBS will terminate the procedure after a threshold of tries – the state is then set to `inactive`. If the application starts running, the state will switch to `active`. If the application terminates abnormally, the MRS will detect this failure and adjust the application state to `inactive`. The MRS will then attempt to recover the application. The application state changes from `inactive` to `migrating`. The application will then be processed according to the described JBS state model, i. e. it run through the state `unsubmitted`, `pending` and `active`.

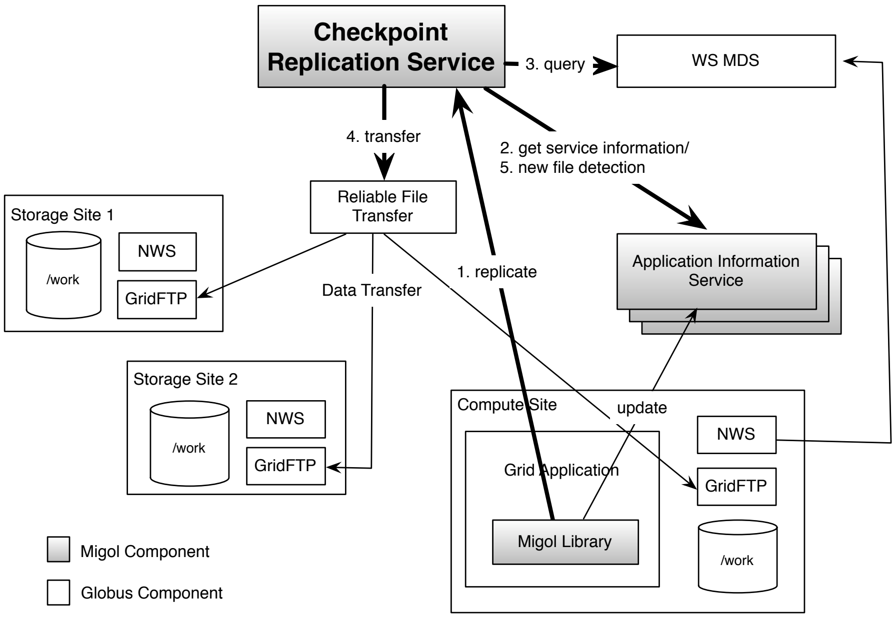


Figure 6.9: Checkpoint Replication Service Architecture and Interactions

6.6 The Checkpoint Replication Service

The *Checkpoint Replication Service (CRS)* provides an adaptive service for the automatic and reliable replication of files in the Grid. In the following section the functionality and architecture of the CRS is described.

6.6.1 CRS Architecture

The CRS is responsible for the automatic discovery and selection of storage resources, the management of the data transfer and the detection of new file versions. Figure 6.9 illustrates the Checkpoint Replication Service in the context of the Migol Grid infrastructure.

The replication of a checkpoint can be triggered by a user or by an application, e. g. when reaching an important milestone (message 1). An easy-to-use interface to the CRS for C/C++ applications is provided by the Migol library (see section 6.7). With the initiation of a checkpoint replication, a WS-Resource, which stores the soft state information of the process, is created. To obtain information about all checkpoint files, the CRS queries the AIS for the Grid Service Object of the application (message 2). Information about available storage sites and bandwidths are obtained from the MDS (message 3). For this purpose, the NWS is used to determine the bandwidths to possible target sites. The bandwidth data is periodically propagated by the NWS to the MDS. Depending on transfer sizes, the available storage resources and current bandwidths, the CRS selects a suitable resource. For management of the data transfer the RFT is used (message 4).

During the entire application runtime (i. e. the application state in the AIS is `active`) the CRS monitors the Grid Service Object of the application to detect updated checkpoint versions or new checkpoint files (message 5). All files listed in a GSO are associated with a version number, which is updated with each modification. Whenever a new file event occurs, a new replication is automatically triggered by the CRS.

6.6.2 Adaptive Location Selection

The selection of an appropriate replication schema is critical to guarantee the overall performance of the system. Different selection strategies for determining the replication target site have been proposed [264]. To support various policies and to allow future extensions, the CRS provides a plugin mechanism.

The default plugin uses a simple adaptive selection algorithm:

1. The storage capacity of all available resources is matched with the size of the checkpoint file.

2. The remaining resources are ordered by the available bandwidth obtained from the NWS data in the MDS.

3. The resource with the highest available bandwidth is selected.

In addition, users are allowed to specify preferred sites. For example, the D-Grid has dedicated storage sites, which are especially well suited as backup site for checkpoints.

Because of the uncertainty found in a Grid environment, this approach can only be considered a rough approximation. A main limitation is the restricted availability of global information about bandwidth and storage resources. Thus, this resource selection schema can only be considered as a simple heuristic.

6.6.3 Replication Procedure

Having selected a suitable resource for a checkpoint replica, a data transfer is initiated. For the management of transfers the Reliable File Transfer (RFT) [17] service is used. After successful termination of a transfer, a replica entry in the Grid Service Object (GSO) of the application is created. Typically each checkpoint written by an application has a unique filename. This way naming collisions between different checkpoint versions are avoided. In addition, each replica is associated with a version number and a state (i. e. available, unavailable, deleted).

The version number is used to detect update conflicts at the AIS. Conflicts can occur for example if the application and the CRS try to update a certain GSO at the same time. Thus, the AIS only permits updates if the right version number is presented. If a conflict occurs, the CRS reconciles the GSO and then re-executes the update operation (see section 6.9).

After the replication process is finished, the checkpoint files are periodically monitored by the new file detector task of the CRS. In case the service detects a new file in the application's GSO, a replication process for this file is started. Further, all checkpoint replicas are periodically monitored - in case a file replica is not reachable it is marked unavailable and a new replica is created.

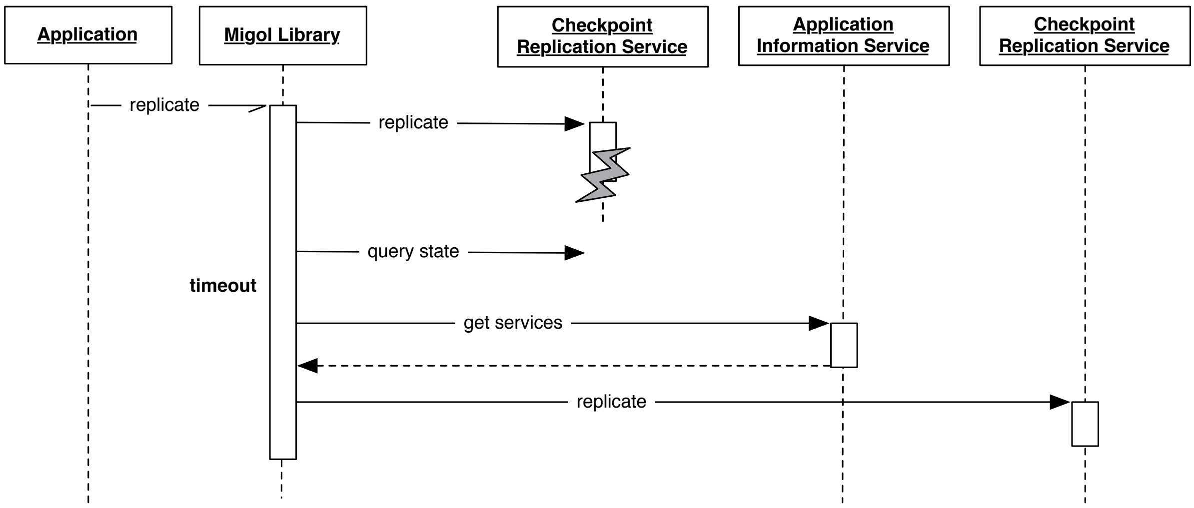


Figure 6.10: Recovery after a CRS Failure

6.6.4 Replica Selection

To conduct a restart of an application, the Job Broker Service (JBS) must select a suitable checkpoint replica, which is then used for recovering the application. Different trade-offs must be considered: the JBS may either select a remote resource, which is available right away, but requires a data transfer, or it may choose to wait for a local resource to become available. To rank available compute resources the JBS uses the shortest expected delay heuristic. For all checkpoint replicas and compute resources, an endtime for the application is estimated. The compute resource and checkpoint replica with the shortest delay is chosen (see also section 6.3.2).

6.6.5 Fault Tolerant Checkpoint Replication

Different faults, such as network failures, storage node failures or the failure of the Checkpoint Replication Service itself can occur during a data replication. Transfer errors, such as transient network failures are handled by the RFT service: all transfer states are stored in a database. In case of a permanent transfer failure the CRS will select a different replication target.

Special precautions must be taken for failures of the CRS itself. For this purpose multiple CRS instances are deployed in the Grid. Since the critical application state, consisting of all checkpoints, replicas and file transfers, is stored in the highly available AIS, a different CRS can take-over the replication and the new file detection process. As shown in Figure 6.10, the Migol library (see section 6.7) can handle transient and permanent failures of the CRS without involvement of the user. After initiation of a replication the library continuously monitors the state of its CRS resource. In case of a failure, the client respectively the Migol library initiates the discovery of a new CRS using the AIS. The replication request is then re-executed at the new CRS. Since the necessary meta-data is stored in the GSO at the AIS, the new CRS can transparently take-over the replication task.

6.7 Application Integration: Migol Library and SAGA

A major challenge is the provision of generic fault tolerance services without requiring large modification of applications. To support an automatic recovery, Grid applications must be able to interact with the Migol infrastructure in the following way:

- The application must register checkpoint and job metadata with the infrastructure.
- The infrastructure must be able to externally monitor the application.
- The infrastructure must be able to externally notify the application and trigger the saving of a checkpoint.

In the following two different approaches for integration of the Migol infrastructure into Grid applications are discussed: the Migol library and SAGA. The Migol library is a lean, portable API for accessing the basic functionality of Migol. The Simple API for Grid Applications provides a feature rich programing abstractions. In addition to the Migol functionality provided via the SAGA Checkpoint Recovery API, various other APIs for supporting file transfer, RPC communication etc. are offered.

6.7.1 Migol Library

The Migol library is based on C/C++ and also supports MPI applications. Applications must solely include the Migol header, make an initialization and finalization call, and link against the library. During initialization, the embedded gSOAP Web service is started. The application can then be monitored using the `Monitorable` Web Service interface (see Figure 6.11).

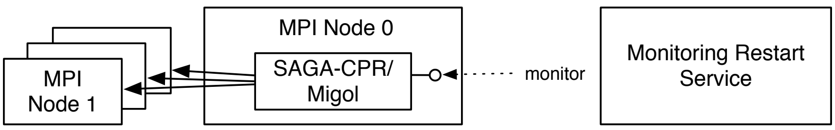


Figure 6.11: Migol Application-Level Monitoring

The Migol library supports the detection of node crashes as well as custom, user-defined error conditions. Special support is provided for MPI applications: a call to the monitoring interface of a parallel application will initiate a socket ping to all nodes. If all nodes are reachable a successful response is returned, otherwise the application is marked as failed. In contrast to other fault detection schemas, which mainly operate on operating system level and must solely rely on monitoring of the external process state, this approach allows the detection of much more different and complex faults. For example, Migol is able to detect hanging MPI applications exposing the failure behavior described in section 5.4. However, applications

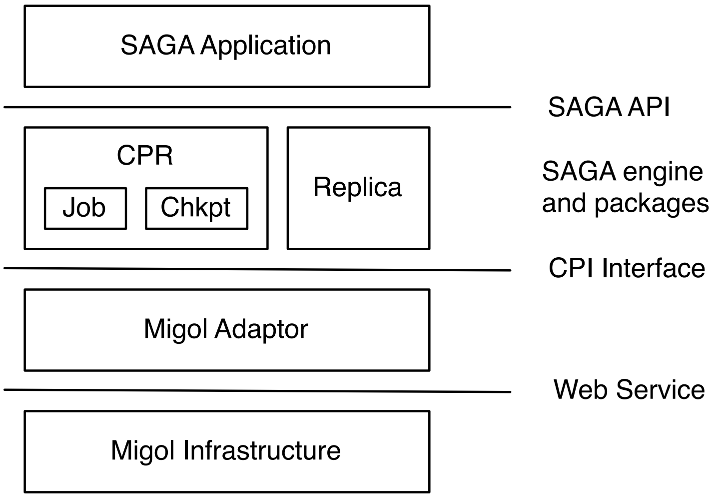


Figure 6.12: SAGA Migol Architecture

can chose to override this implementation, e. g. deploying a more efficient but less accurate gossip-based failure detector.

In addition, support for automatic checkpointing and restarting of applications is provided. For this purpose, the Migol library provides an interface for application-level checkpointing. If an application wants to utilize the automatic recovery support of Migol, this application-level checkpointing API must be used to register checkpoint metadata with the Migol infrastructure. In principal, this interface can also be used in conjunction with lower-level checkpointing interfaces. However, in particular system-level and/or user-level checkpointing mechanisms are not portable across heterogeneous Grid resources. Checkpoints can be written either periodically by the application or in response to a request of the Migration Service.

6.7.2 SAGA - Migol Adaptor

As described in section 3.1.4 the Simple API for Grid Applications (SAGA) [143] is an application-level API for the development of distributed applications. While Migol provides a service-oriented middleware for the supporting fault tolerant applications, SAGA focuses on how Grid applications can access middleware functionality on API level. Thus, Migol and SAGA are complementary: the functionality of the Migol can ideally exposed via the new *SAGA Checkpoint Recovery (CPR)* API. SAGA-CPR provides a well-defined, standardized abstraction for the management of checkpoint recoverable Grid applications as well as application-level checkpointing.

Currently, a C++ and Java implementation for SAGA exists. Although the Migol backend mainly consists of Java-based Grid services, the frameworks especially addresses the fault tolerance of long-running applications. These compute intensive applications have high performance requirements and, in general, require native libraries, e. g. for numerical computations, or MPI for cluster communication. Thus, we focus on the SAGA C++ reference implementation [182].

Figure 6.12 illustrates the integration of the Migol services into SAGA. The Migol SAGA adaptor implements the *CheckpointRecovery* and *Replica* capability provider interface. While the CPR Job APIs can be used to start jobs via the JBS, to trigger checkpoints and migrations, the checkpoint API is used to register checkpoint metadata. The Checkpoint Replication Service is supported via the Replica API. Similar to the Migol library, the monitoring server is automatically started with initialization of the SAGA library. In addition to the CPR APIs, application can utilize a variety of other API provided by SAGA, e. g. for RPC communication or management of file transfers.

6.7.3 Application-Level Fault Tolerance

The Migol library extensively provides support for fault tolerance. The library is aware of possible service failures within the Migol infrastructure and provides extensive re-connect capabilities. In case a failed service is discovery, the library automatically re-executes requests or issues a request to another service instance.

But, especially highly distributed applications should provide intrinsic support for fault tolerance. Grid application must be able to detect errors or to receive notification about errors from the underlying middleware. The Migol library and SAGA adaptor successfully address most of these issues. However, following the end-to-end argument of Saltzer et. al. [275] it is essential to address fault tolerance on application level. For example, a disadvantage of Migol is that it only supports a very coarse granular recovery. It is rather desirable that application are able to recovery from partial failures without requiring a complete restart: In a large distributed MPI application, the failure of a single node should not lead to a outage of the complete system. Following the fault tolerance principles discussed before, any application should be comprised of independent modules, that expose a well-defined failure behaviour. If necessary, applications should recover from partial failures without requiring assistance from Migol. Fault tolerance on application-level can evite much overhead.

6.8 Security Infrastructure

Clearly, security is another important property of a dependable infrastructure. Grid resources are potentially shared among many different virtual organizations. Since a single corrupt machine can effect the entire Grid, security issues must be addressed in the system architecture.

The ability to migrate applications using checkpoint files has obvious security implications. For example, to enforce the least privilege principle the credentials of the user are required to conduct a recovery. At the same time, the user might be offline during the failure of the application. In the following the security requirements are summarized:

- **Mutual authentication:** In general, resource owners require authentication to protect themselves against malicious users. Resource users may also demand the authentication of resources, to protect themselves against spoofing by malicious resource providers. Thus, all interactions with Migol services, i. e. the MS, JBS, GRAM, and RFT service, demand a mutual authentication.

- **Authorization:** To protect a resource from unauthorized access a fine-granular authorization framework is required. Migol demands an authorization to control access to the AIS, the MS, and the JBS. Individual service resources and applications should only be accessible by their owner.

- **Message protection:** To prevent eavesdropping and man-in-the-middle attacks, the integrity and confidentiality of data communicated between resources must be ensured, particularly when communication occurs over public networks. In general, both integrity and privacy is required for all Migol operations. Due to performance reasons users can choose a lower level of protection, e. g. for transferring large checkpoint files.

- **Single-sign-on:** The user should be able to interact with different services without repeatedly entering his password.

- **Least privilege principle:** Each entity is assigned only the minimal privileges needed to accomplish a task. The MS and the JBS must perform all operations, e. g. a request for a resource or a job submission, on behalf of the user. This approach ensures that only resources, which the user is entitled to use, are accessible. The user's identity needs to be passed securely and transparently between the Migol services by credential delegation.

The Migol security implementation is based on the Globus Toolkit and its Grid Security Infrastructure (GSI) [129]. GSI provides security features as authentication, authorization, credential delegation, and message protection.

But some Migol use cases, e. g. application recovery, require additional security features, which are not part of GSI. If an application fails, it is not capable anymore to delegate its credentials to the Migration Service. Hence, the user credential must be stored in a repository. Migol integrates the credential repository into its Application Information Service. During service registration user credentials are transparently delegated and stored at the AIS. Access to the AIS is enforced by a role-based access control, i. e. a retrieval can only be initiated by an authorized service, such as a recovery service like the MS or the CRS, via delegation-on-demand [157] (see Figure 6.7). Hence, a migration or checkpoint replication is possible even if the user is not available.

The main reason for incorporating these features into the AIS is that existing solutions are not reliable and flexible enough. For example, the MyProxy credential repository [44], which is part of the Globus Toolkit, cannot be replicated and therefore represents a single point of failure.

6.9 Concurrency Considerations

Distributed service infrastructures such as Migol are concurrent by nature. In their lifecycle Grid applications interoperate with many different Migol services. As described, the core of the application state is always stored within a Grid Service Object in the AIS. Since multiple

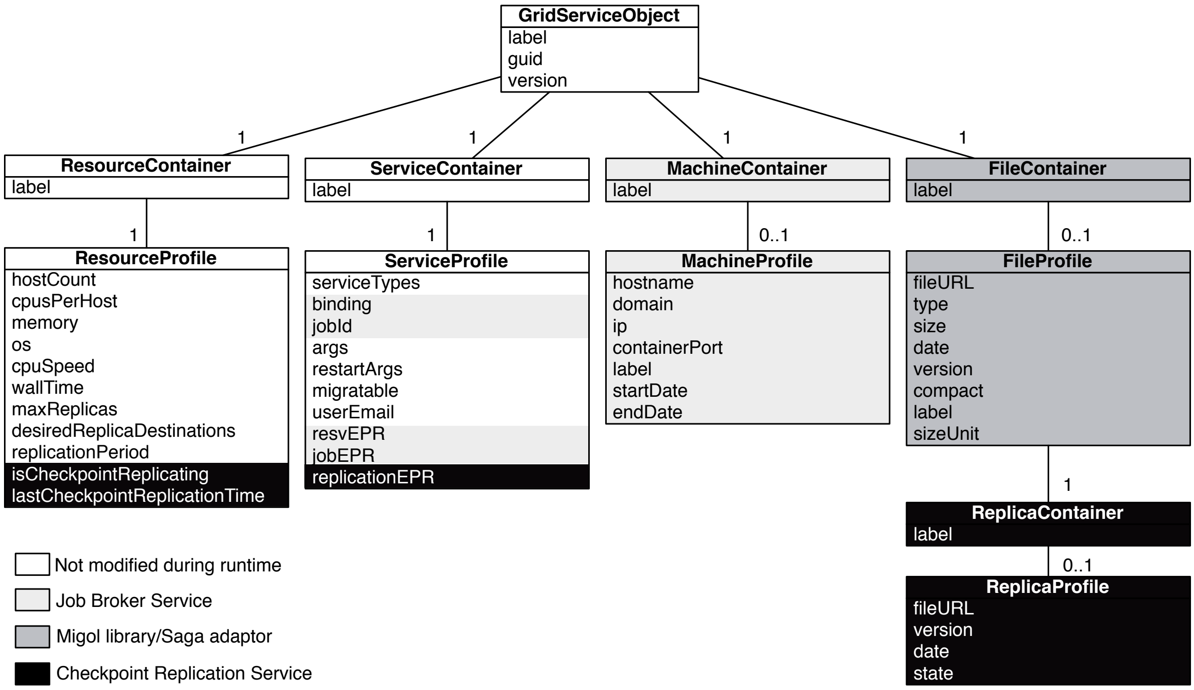


Figure 6.13: Grid Service Objects: Modification Patterns

services potentially modify the same Grid Service Object, it is necessary to consider the update change of all services to detect possible conflicts. A conflict always occurs if two operations concurrently update the same data [302]. Figure 6.13 illustrates which data is modified by which service after a job start. As shown, most of the resource and service metadata is static and only initially set. However, several fields are modified during the runtime of an application. Most of the modification are non-conflicting, i. e. different data fields are modified. The only field, which is updated by multiple services is the binding field of the application. This field contains the contact URL of the application, which is written by the JBS and the application itself. However, a concurrent modification can be eliminated since the JBS only modifies this field during a job start or a migration, i. e. the application is not active at that time.

Despite the exclusion of overlapping updates, concurrency conflicts can occur since the AIS always updates a complete GSO at the time. Thus, an optimistic concurrency control schema [195] based on a version number, which is part of GSO, is used. The version number is incremented with each change at the AIS. Updates are only permitted if the new GSO contains the right version number. This way, conflicts can be detected and resolved by the respective client. Since each service manipulates a different area in the GSO, the reconciliation can be conducted transparently by the AIS client without user interaction. This client is used by all services and the Migol library.

6.10 Summary and Discussion

Migol addresses the fault tolerance of long-running Grid applications by handling common failures in Grid applications transparently without user interaction. In case of failures, e. g. a node-crash, applications are automatically restarted from the last saved checkpoint. Checkpointing ensures that the amount of lost computation is minimized. The framework automatically selects an appropriate resource by predicting the end time of a job on all resources. The local site is preferred – if the prediction favors another resource the job is migrated. With this capability Migol makes the resetting of the system state completely automatic - the programmer must solely implement an application-level state saving mechanism. Failure detection and recovery is handled transparently within the Migol framework.

Migol relies on the classical rollback recovery. Another approach often discussed with respect to short-running tasks is the usage of process replication. However, the overhead of running all task at least twice, is immense. Especially for compute intensive tasks the wasting of half the cycles is not an option.

As discussed in chapter 2.5.2, heartbeat-based failure detection has some limitations. While the central monitoring service is not a single point of failure (see chapter 7), it can become a performance bottleneck. Depending on the monitoring intervall the MRS can contribute some significant network load. A more distributed failure detection schema could be address this problem in the future.

"A computer lets you make more mistakes faster than any other invention in human history."

anonymous

7 Fault Tolerance in Migol

The components of an autonomic infrastructure such as Migol must be able to tolerate and recover from failures. This chapter extensively details the fault tolerance techniques used to build the Migol infrastructure. Section 7.1 presents the common principles and patterns that Migol is based on. Of course, these patterns can also be applied to other systems. Not all systems have identical reliability and availability requirements. Thus, section 7.2 proposes a service classification, which is used to select appropriate fault tolerance mechanisms for the different services. The core of Migol is the Application Information Service (AIS) – due to its central role the service is actively replicated using a Web Service-based protocol. Section 7.3 discusses different issues related to active replication of Grid services in particular with focus on the AIS.

7.1 Fault Tolerance Patterns and Principles

Design patterns summarize reoccurring design problems and their solutions to allow the re-use of proven techniques to solve the respective problem. Design patterns for object oriented systems have been introduced by Gamma et. al. [134]. Since patterns are an effective way to understand design problems, this section aims to highlight patterns, which are in particular relevant for designing reliable distributed systems, such as Migol. In the following an overview of patterns and practises for fault tolerant system design extracted from various literature sources is given [36, 51, 144, 158, 252]:

- **Fault prevention** aims to ensure reliability by removing possible faults of a system prior to its regular use. Common techniques include defensive programing, coding standards, quality assurance by extensive testing and code reviews. Although it is not possible to eliminate all possible faults, fault prevention should be able to find most of the deterministic faults, such as Bohrbugs.

- **Modularization:** A service-oriented architecture naturally comprises of modular services. Services can again be decomposed into modules. The more fine-grained the modularization, the more fine-grained error detection and recovery mechanisms can be used. In such a hierarchical system structure faults can be earlier detected and masked.
- **Well-Defined Failure Semantic:** All services should expose a well-defined failure semantic. In case of a failure, service should stop working rapidly to allow a rapid error detection.
- **Recoverability:** All services must be able to recover after a failure. A fast recovery increases the availability of a system. Many errors are transient and can be handled by a simple re-execution. A checkpoint-based recovery avoids unnecessary re-computations and minimizes the amount of lost data. Especially for stateful Grid services it is essential that the resource state is recoverable after a failure.
- **High Availability:** Critical services that require continuous operation must be replicated. A common pattern for service replication is the state machine approach [282].
- **Timeout-Based Failure Detection:** Failure detection is commonly done using a variant of the heartbeat protocol. In general, the management of the timeouts is the most critical aspect.
- **End-to-End Reliability:** Following the end-to-end argument [275], the client is ultimately responsible for reliability. That means e. g. that all service clients must validate every result. Further, clients must be aware of possible service interruptions. Thus, clients cannot solely rely on intrinsic fault tolerance and must extensively use fault tolerance techniques, such as timeout based failure detection and simple re-execution mechanisms. This feature is commonly referred to as reconnection capability.
- **Automatic Recovery**, i. e. the recovery of a service or module without human intervention, minimize the downtime and increases the overall availability. To avoid lost data or costly re-computations, this pattern should be used in conjunction with the checkpoint pattern.
- **Component Reuse:** Proven and tested frameworks and modules reduce the probability of faults and should therefore be reused.
- **System Maintenance:** The operation and maintenance of a system is an important aspect. The system should provide a separate interface for maintenance and monitoring. System administrators require precise information in order to be able to resolve complex errors.
- **Keep it simple:** The more complex the system, the more manifold the failure behaviour. Especially complex fault tolerance schemes such as active replication can lead to undesired failures. Thus, the benefits and risks of each deployed fault tolerance technique must be carefully weighted.

- **Security:** To avoid malicious activities, basic security mechanisms such as authentication, authorization and message protection must be used.

This principles have been extensively used within Migol. However, they are not solely restricted to Migol and can be applied to a variety of different systems and applications. Based on the criticality classification of all Migol services, the usage of these principles within Migol is explained.

7.2 Service Classification

The individual Migol services have different dependability requirements and have therefore been classified into critical and medium-critical services. In the following, the different classes and services are discussed. Based on this classification, the fault tolerance levels introduced in section 2.8 are applied.

7.2.1 Critical Services

The failure of a critical service has an impact on the entire Grid. In particular, this definition applies to the Application Information Service (AIS) and the Monitoring Restart Service (MRS). As described, the AIS is the core of Migol, which maintains essential state information. A failure of the AIS has severe consequences:

- The AIS serves as central naming service. Applications are assigned a GUID. During the lifetime of the application, users and services can obtain the current location of the application by querying the AIS. If the AIS is down, applications cannot update their application metadata – this data is urgently required to support an automatic recovery.

- The MRS must be able to obtain the monitoring endpoints of all Grid applications from the AIS. In case of a complete AIS failure no monitoring is possible.

To avoid a single point of failure, the AIS and MRS are replicated across several Grid nodes. Since the AIS stores stateful data, which is both read and updated, ensuring data consistency across multiple nodes is difficult. To avoid that queries to different service instances return ambiguous results, a strongly consistent replication semantic is required. That means that it must be ensured that all update requests are executed at all AIS replicas in the same order (see section 7.3).

7.2.2 Medium Critical Services

A failure of a medium critical service has only small implications on the overall system, e. g. the loss of the transient, short-lived state of a constraint set of services. This applies in particular to the following Migol services: the Advance Reservation Service (ARS), the Job Broker Service (JBS), the Migration Service (MS) and the Checkpoint Replication Service (CRS). Usually multiple of these services are deployed within a Grid. The failure of a single service instance only affects a certain group of users. The remaining service instances are in general not effected.

Fault tolerance of medium-critical services is ensured by using the persistent resource properties of GT4 [292]. This interface is comparable to application-level checkpointing. It provides callbacks for writing resource states to a stable storage as well as for recovering resources. The store callback is used by the ARS, JBS, MS and CRS to save critical internal state information. After a container crash, the load callback is used to reinitialize the state of all persistent container resources.

These services deploy custom recovery routines to restart the service's background activities. For example, the JBS queries the state of all jobs submitted prior to the failure – if a job failed in the meantime, the submission process is re-executed. The CRS attempts to recover all formerly active transfers (for details see section 6.3.2 and 6.6.5).

Following the described end-to-end principle, the client is ultimately responsible for reliability. While Migol's architecture ensures that all services are recoverable - clients must anticipate possible transient and/or permanent failures. In case of a transient failure, the client can simply use a re-execution schema to compensate a short downtime, e. g. due to a server reboot. Clients can rely on the Migol library to execute automatic re-try operations, if desired.

7.2.3 Low-Critical Service: WS MDS

A less critical services for the overall Migol system is the WS MDS. The reliability of the MDS is ensured by deployment of multiple, hierarchically organized MDS services in the Grid. Each GT4 container contains a MDS instance, which aggregates local resource information of the container's services. In addition, sites and VOs usually run one or more MDS services, which again aggregate information from different local services. Since several MDS services cache the same data, the MDS does not represent a single point of failure. In case of a failure, the state of the WS MDS can be completely recollected after a failure. Both JBS and CRS depend on the WS MDS for resource discovery and selection. Both services can be configured to query and cache information of multiple MDS services. This ensures that failures of a single WS MDS service can be tolerated.

In contrast to the AIS, the information stored in the WS MDS is per definition incomplete and of an uncertain quality. Thus, the data obtained from WS MDS is solely used for the ranking of resources – information critical for the recovery of applications, such as the location of the checkpoint files and start parameters, is stored in the highly available AIS.

Having discussed the different service classes, the remainder of this chapter focuses on critically services and in particular on mechanisms to ensure the availability of these services.

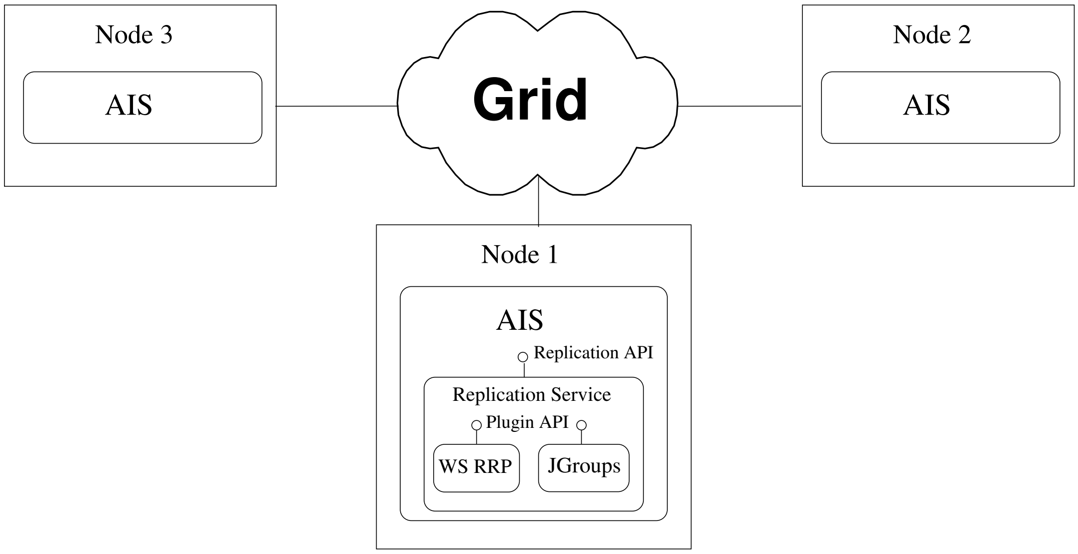


Figure 7.1: AIS Architecture

7.3 Service Replication in the Grid: Ensuring the Availability of the AIS

A well known concept for achieving fault tolerance of highly critical infrastructure services, such as the AIS, is active replication. A common approach for building an actively replicated system is the state machine approach introduced by Schneider [282]. All inputs are consistently distributed using a consensus algorithm. Each replica then updates its local state based on the input data.

For local networks, different algorithms for group communication and service replication have been proposed in literature [51]. The most popular ones are the centralized algorithm (also called Master-Slave), and the distributed algorithms Token Exchange on Demand (also called Floor-Passing) [72], and Totem Single Ring [25].

The design of a scalable replication service for a Grid is a complex task due to a myriad of reasons:

- Message latencies can be large and highly unpredictable.
- Failure detection in a Grid is less accurate than in a LAN.
- There is no efficient support for the broadcasting of messages.

Several investigations have shown that Totem provides a good performance and scalability especially with concurrent and distributed requests while offering strong consistency guarantees [285]. Thus, the AIS supports the replication via a variation of the Totem protocol, the Web Service based Ring Replication Protocol (WS RRP) [215].

Migol's Replication Service provides a high-level interface to the group communication capabilities of the underlying protocol (see Figure 7.1). All AIS requests, e. g. service registration or update requests, are forwarded to the Replication Service. The Replication Service (RS) propagates the request to all nodes using a group communication plugin, which ensures the total ordering. All nodes receive the same messages and can therefore maintain the same persistent state. Currently two different replication implementations for total ordering of updates and group membership are offered: The new Web service-based Ring Replication Protocol (WS RRP) for Grid environments, and JGroups [41] which is a Java-based toolkit for group communication developed at the Cornell University. Both plugins will be explained in the next sections.

7.3.1 Web Service Ring Replication Protocol (WS RRP)

The Web service-based Ring Replication Protocol (WS RRP) adapts Amir's Totem Single Ring Protocol [25] to provide total message ordering and group membership. WS RRP uses a token, which circulates on a logical ring, to ensure the ordering of messages and group membership. In contrast to Totem, where the token only controls access to the ring, WS RRP also distributes application messages, for example service update messages, via the token. Since existing Grids do not provide native multicast support, a multicast would have to be simulated by sending n-1 unicasts. Thus, the piggybacking of messages provides an efficient mean to distribute application messages to all members in a Grid environment.

The WS RRP service uses SOAP communication to replicate the content of the AIS. The main advantage of WS RRP is the interoperability gained by using Web service technologies. Further, access to the replication service and message communication can be secured by the Grid Security Infrastructure (GSI).

In the following the details of the ordering and membership protocol are explained.

Total Ordering Protocol

In this part the total ordering protocol used during regular operation for ensuring the consistency of all updates across the AIS service group is described. As explained, a token, which circulates around a logical ring, is used for the distribution of control data as well as all update messages. The token consists of the following information:

- `token_sequence`: The sequence number of the token. This field is used to detect e. g. duplicate tokens.
- `next_sequence`: Each update message is assigned a message sequence number. All messages are ordered according to this number. The next available message sequence number is transmitted via the token (high watermark vector).
- `rtr`: A list of retransmission requests. This field is used to request the re-transfer of an update message, e. g. in case an AIS instance rejoins a group.
- `message_list`: A list of update messages. Each message contains all required data to conduct a local state update of a group member.

If an application message arrives from the AIS, it will be dispatched to the WS RRP plugin. WS RRP queues all application messages until the next token arrives. Each token is processed with the following steps:

1. Update messages are applied to the local AIS, if all messages with a lower message sequence number than the update have been already handled. Otherwise, the updates are buffered.
2. In case of a gap between the sequence number of the last processed update and a received update, the missing messages are added to the retransmission list of the token.
3. If there is a retransmission request for a message, the respective message is appended to the message list and the request is deleted from the retransmission list.
4. If the token contains messages originated by this node, these messages are deleted since they have passed all nodes in the ring.
5. New application messages from the AIS are now added to the token. For each new message the sequence number is incremented.
6. Finally, the token sequence number is incremented and the token is forwarded.

Since only the current owner of the token is able to originate updates, all updates are serialized in the order of the assigned message sequence numbers. WS RRP provides an agreed delivery guarantee, i. e. it ensures that all replicas deliver all update messages in the same order. A safe delivery mode as proposed by the original Totem [25] provides additional guarantees by ensuring that all replicas have received a message before messages are delivered. However, this mode requires another token roundtrip and increases the latency. WS RRP currently only supports agreed delivery, but can easily be extended to safe delivery if higher consistency requirements arise. Potentially conflicting replica updates can always be resolved on application level using the GSO versioning schema (see section 6.9).

In addition to the ordering protocol, the failure detection as well as the group membership mechanism are vital parts of a group communication system. WS RRP uses timeouts to detect failures. Failures are reported to the membership service by initiating a reconfiguration. Both aspects are discussed in the following two sections.

Timeout Management

A challenge in distributed computing is the detection of failures. The only way to detect that a remote process has crashed are timeouts. Timeout management is critical: Low timeouts can lead to expensive reconfigurations, while high timeouts will not detect crashed members for some time. Especially under high loads continuous reconfigurations can lead to a malfunctioning system.

Figure 7.2 gives an overview of the different factors that influence the token roundtrip time. For one token circulation around a ring with n nodes, n messages are necessary. The token

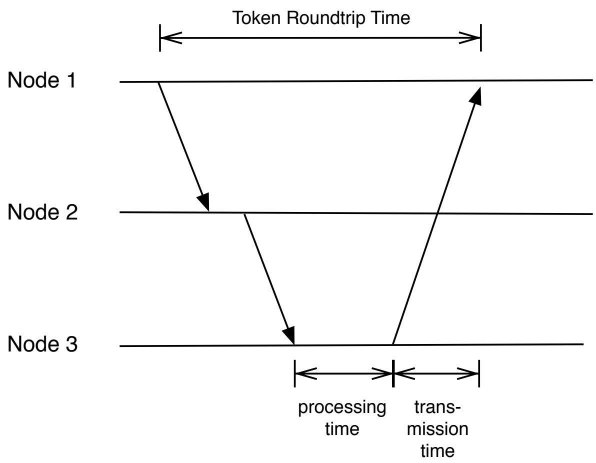


Figure 7.2: WS RRP: Timing Parameter

roundtrip time comprises of the token transmission times and the message processing times. The transmission time mainly comprises of the network latency as well as the protocol overhead, which can be mainly attributed to the SOAP and security stack. Since WS RRP transmits update messages via the token, the processing time greatly depends on the number of update messages in the token.

WS RRP relies on Jacobson's TCP approach [172] to forecast the token retransmission timeout based on continuous measurements of the token roundtrip times. The retransmission timeout is calculated using an exponential average of the mean roundtrip time and its standard deviation.

The roundtrip time is estimated using exponential averaging, i. e. the weighting for older measurements decreases exponentially, giving more importance to recent roundtrip times:

$$mean\_rtt = \alpha \cdot mean\_rtt + (1 - \alpha) current\_rtt$$

As a variation measure the mean deviation is used:

$$dev\_rtt = \alpha \cdot dev\_rtt + (1 - \alpha) \mid mean\_rtt - current\_rtt \mid$$

For α a value of $\frac{1}{8}$, which is close to the 0.1 proposed for TCP, is used. The token retransmission timeout is then calculated using the roundtrip estimation, the variation factor and the number of messages in the token:

$$timeout = mean\_rtt + 4 \cdot dev\_rtt + number\_msg \cdot MSG\_PROC\_TIME$$

The mean deviation is included with a factor of four as suggested by Jacobson, because the multiplication can be done by a single shift and only a small numbers of tokens arrive more than four standard deviations late. For each message in the token a constant factor for the message processing time (MSG_PROC_TIME) is added. The message processing can

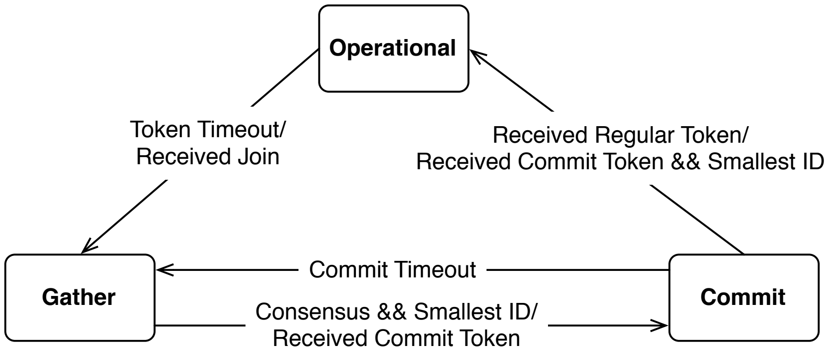


Figure 7.3: WS RRP: Membership States

be configured, per default a message processing time of 100 msec is assumed, which is the expected serialization and processing overhead for a token message on an average machine.

In case of a token retransmission timeout a reconfiguration using the Totem membership protocol is triggered. Additional to the token retransmission timer, WS RRP uses a consensus and commit token timeout, which indicate that a join round or the formation of a new ring has failed. For both timeouts similar sampling techniques are used. The next section describes the membership protocol of WS RRP in detail.

Membership Protocol

The membership protocol is responsible for maintaining a list of currently active processes. WS RRP adapts Totem's membership protocol to resolve processor and network failures. Figure 7.3 shows the finite state machine of the membership protocol. In the operational state the token continuously circulates around the ring. For failure detections timeouts are used as described previously. When a token retransmission timeout occurs, the node switches to the *Gather* state and sends a *join* message to all other nodes in its active node list. A join message contains the following data:

- `sender_id`: The originator of the join message.
- `active_node_list`: The active node list of the sender node also referred to as view of the node.

All nodes that receive a join message also switch to the gather state and start sending join messages containing their current view. Each node maintains a consensus map structure to track the views of all other group members. After reception of a join message this structure is updated using the view and sender id contained in the join message. When a node has received the same view from all other nodes, a consensus regarding the new ring topology has been reached. If the members did not agree on a new view, all nodes update their views with the information obtained from the other group members. The new view is then distributed via a set of join messages.

In case of a consensus the nodes with the smallest ID, emits a commit token. The commit token contains the proposed new membership list of the new ring as well as the last known message sequence number. The state changes to *Commit*. With the successful rotation of the commit token around the ring the membership of the group is confirmed. After completion of the commit round, the node with the smallest ID switches the state to *Operational* and creates a new regular token.

The commit token is also used for data recovery. Each node which receives the commit token, verifies the message sequence number contained in the token. If a higher sequence number is known, the number in the commit token is replaced. Nodes, which have missed messages, can request the retransmission of messages in the *Operational* state.

If a failed node has recovered and is ready to reenter the ring, it can trigger the membership protocol by sending a join message to a ring member.

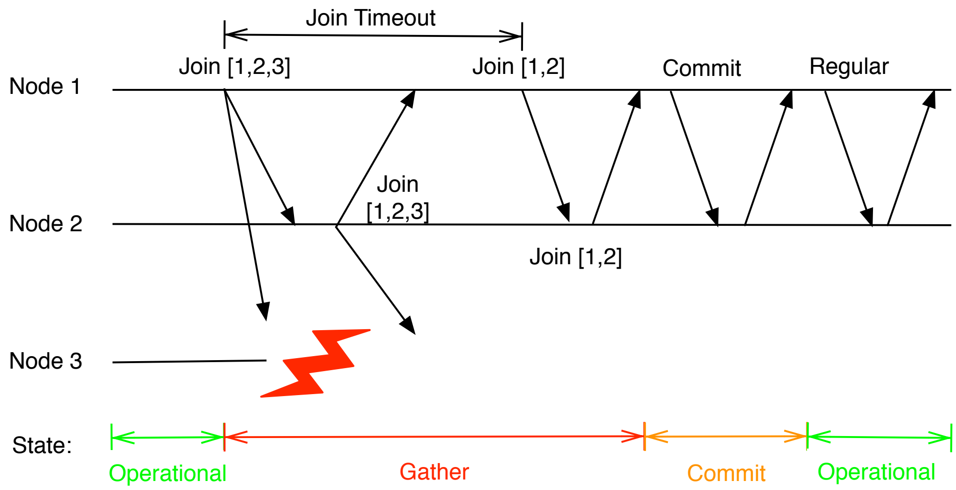


Figure 7.4: WS RRP: Membership Protocol Sequence

Figure 7.4 shows an example of the membership protocol using a ring configuration with three nodes: $\{1, 2, 3\}$. During the operation of the ring node 3 fails causing the initiation of the membership protocol. Depending on the timing and the precise failure behaviour either node 1 or 2 will detect this error. In the following it is assumed that node 1 notices the failure first due to a token timeout. Node 1 will switch to the *Gather* state by sending a join message with the view of $\{1, 2, 3\}$ to all other nodes within the view, i. e. node 2 and 3. Node 2 will respond with a join message containing the same view $\{1, 2, 3\}$. However, node 3 cannot respond to the join request due to the failure. After expiration of the join timeout, node 1 will adjust its view to $\{1, 2\}$. It will then send a new join message to node 2. Since node 2 did neither receive a join message from 3, it will respond with the same view $\{1, 2\}$. Now node 1 knows that a consensus about the new ring topology has been reached. At minimum two join rounds are necessary to reach consensus. However, in certain situations, additional join rounds might be necessary, e. g. in case node 3 fails inconsistently by still responding to a join from node 1, but not a join from node 3.

Now the node with the smallest ID, i. e. node 1, creates a commit token containing the view obtained from the consensus protocol and its message sequence number. The state is now switched to *Commit*. The commit token is send from node 1 to 2. After reception node 2 also moves to the *Commit* state and forwards the token to its successor in the ring. As soon as node 1 receives the commit token back, a new regular token is created and the state is switched to *Operational*.

WS RRP is also able to handle network partitions using a simple primary partition model. The protocol only permits nodes in the partition with the majority of nodes to remain active (cmp. majority consensus [307]). Nodes in the minority partition are halted. After the partition failure is resolved, the state of the AIS instances in the primary partition is transferred to the instances of the minority partition using retransmission requests. This model permits the continuous operation of the AIS as long as less then $\lfloor \frac{n}{2} \rfloor$ nodes fail.

Totem proposes the extended virtual synchronous model [230] for handling network partitions. This model permits both the primary and non-primary partition to make progress by introducing transitional configurations. However, the proposed protocol weakens the global consistency guarantees leading to potential conflicting updates. Other protocols supporting this model delay the execution of updates until a sufficient number of processes is available and are thus very costly [51]. Thus, WS RRP relies on the more robust primary partition model.

Having described the features and functionality of the WS RRP protocol, the JGroups replication framework, which can be used as replacement for WS RRP, is presented in the following.

7.3.2 JGroups Replication Framework

JGroups [41] is a Java-based group communication toolkit, which provides a flexible protocol stack with configurable semantics for group communication. JGroups aims to provide a group communication stack for Java applications. The framework was originally based on the Ensemble project [160] developed at the Cornell University.

Figure 7.5 shows the architecture of JGroups. Similar to Horus, JGroups is based on a micro-protocol stack. Each protocol provides a defined capability, e. g. group membership, flow control or ordering of messages. These micro-protocols can be combined to provide the required group communication properties, such as consistency, performance, security etc. The entry point to the JGroups toolkit is the `JChannel` API, a socket-style interface to the process groups. The group membership service maintains a list of group members as they join and leave. In general, the used stack comprises of different protocols for message ordering, reliability and failure detection. On the bottom layer, JGroups can use UDP/IP multicasts or TCP for message transmission. In Grid and WAN settings, commonly TCP is required due to the lack of support for native multicasts. The TCP micro-protocol uses a mesh of TCP connections and unicasts to distribute messages.

For total ordering JGroups provides a *sequencer* protocol [181, 51]. This protocol uses a distinguished, central process to determine the global message order. Figure 7.6 shows the protocol sequence: Messages are initially send to a sequencer process, the coordinator. The

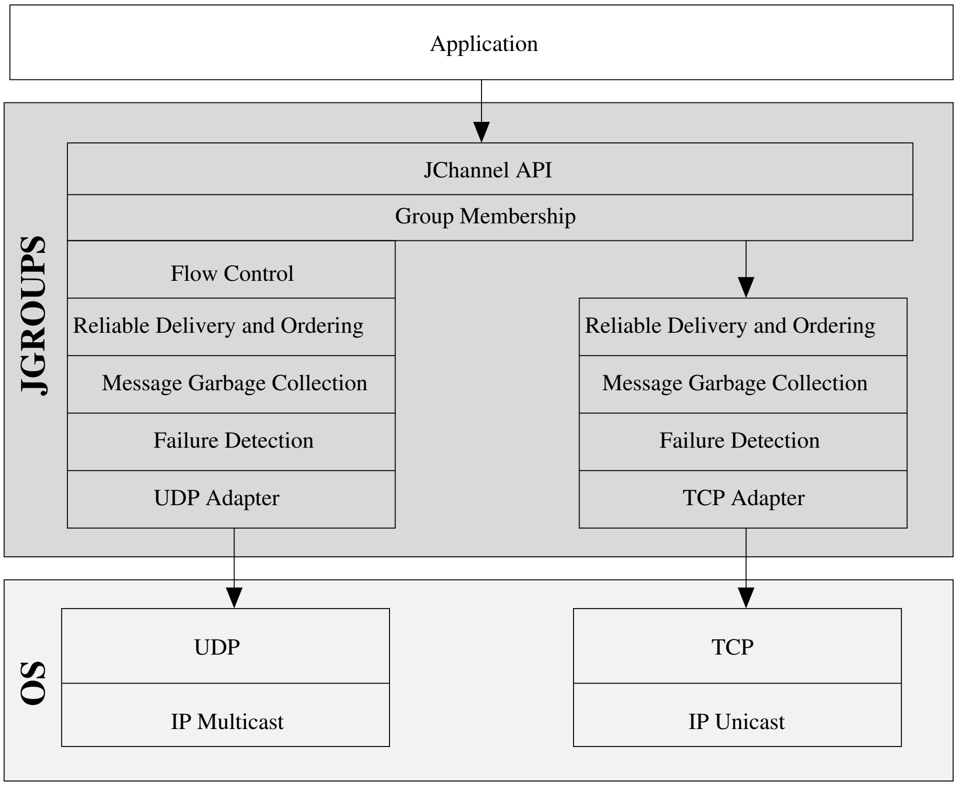


Figure 7.5: JGroups Architecture

coordinator allocates a sequence number and then broadcasts the message to all group members. The group members ensure that messages are delivered with respect to the assigned sequence number. The reliable delivery of messages is ensured by the NAKACK micro-protocol. NACK assigns sequence numbers to all messages and automatically requests the retransmission of missing messages. Former JGroups releases also supported additional protocols, such as Totem [56]. Unfortunately, the Totem implementation proofed not robust enough [215].

For group membership management, the JGroups Group Membership Service (GMS) uses a coordinator process, which manages the joins and the departures of nodes (cmp. [51, Chapter 15.1.2]). For failure detection, JGroups provides various micro-protocols, e. g. a heartbeat based failure detector (FD) and a socket-based failure detector (FD_SOCK).

The central coordinator concept used by the JGroups GMS is associated with a number of drawbacks. The coordinator can easily become a bottleneck. Further, the failure of the coordinator process must be considered. Upon such a failure the group membership protocol must select a new member as coordinator. The GMS uses a convention and always chooses the first member of the view list sorted by IP and port as coordinator. This approach simplifies the reconfiguration and avoids costly election algorithms, which are usually associated with costs in the magnitude of $O(n^2)$. In case a view with a different coordinator has been established, the sequencer protocol ensures that all unacknowledged messages are re-send to the new coordinator.

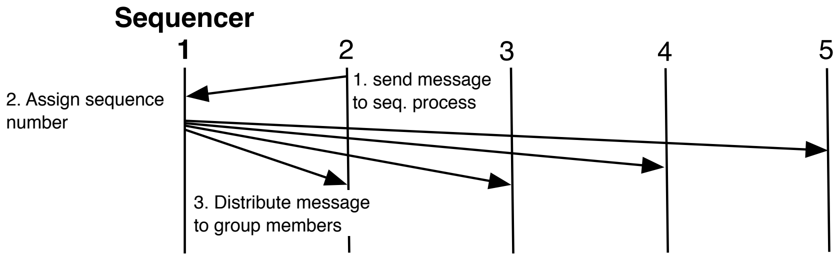


Figure 7.6: JGroups Sequencer Protocol

To avoid inconsistencies during network partitionings, the GMS can be extended to support the primary partition model. To achieve this Migol's Replication Service utilizes the view change event, which is published by the `Channel` object. Based on the current number of nodes within the view, the Replication Service can determine the primary component. The nodes in the non-primary partions are shut down. As with WS RRP, the availability of the AIS is maintained as long as less than $\lfloor \frac{n}{2} \rfloor$ processes fail.

Table 7.1 summarizes the protocol stack used for replication of the AIS. In addition to the described protocols, the STATE_TRANSFER protocol is used to transfer the current group state to new or re-joining group members. Further, the ENCRYPT and AUTH protocol provide important security properties, such as the authenticity and confidentiality of all messages.

In the following the properties of both WS RRP and JGroups are compared.

7.3.3 WS RRP and JGroups Sequencer Properties

Both WS RRP and the JGroups sequencer ensure the total ordering of all update messages. However, some properties found in local systems have be consciously weakened to provide an optimal performance. For example, both JGroups and WS RPP only ensure the total ordering of update messages – read messages are executed locally on the respective AIS instance. That means that the linearizable property is not guaranteed. However, the AIS uses an optimistic versioning schema to detect and resolve possible conflicts arising from dirty reads. Further, rather than relying on a more complex group membership algorithm to ensure a view synchronous delivery [51], both protocols rely on retransmissions and state transfers to derive a consistent state after the new view has been established.

In the following the response times and costs associated with both protocols are compared. Assuming a group consisting of n members, the WS RRP response time in the case of an agreed delivery comprises of the following factors:

- in average one half of the token roundtrip time for waiting on the token ($\frac{n}{2}$ messages),
- one token roundtrip for distribution of the application message (n messages),
- and the processing times at all nodes.

Neglecting the processing time, in total $n + \frac{n}{2}$ account for the average response time of WS RRP. The response time of the JGroups sequencer comprise of:

Micro- Protocol	**Description**
TCP	Uses TCP as communication protocol instead of UDP multicasts. Group members will be connected by a mesh of TCP connections.
TCPPING	Discovery protocol, which uses a host list to determine the initial group members.
FD	Failure Detection based on heartbeat messages.
VERIFY_SUSPECT	Verifies that suspected members are really dead before they are removed from the member list.
ENCRYPT	Uses encryption to secure messages.
NAKACK	Uses negative acknowledgement-based message retransmissions.
STABLE	This protocol detects whether a message has been delivered to all members and can therefore be garbage collected.
AUTH	Authentication of join requests to the membership service based on a X.509 certificate.
GMS	Group membership service, which is responsible for keeping track of joining and leaving processes.
SEQUENCER	Ensure total ordering of message delivery using a central sequencer process.
FC	Flow control based on credits.
FRAG2	Fragmentation of messages larger than 8192 bytes.
STATE_TRANSFER	Support state transfer so that new members can retrieve the state of the group from an existing member.
FLUSH	The protocol forces group members to flush their pending messages and to stop sending any new message. Flushing is done before reconfigurations and state transfers.

Table 7.1: JGroups Protocol Stack

- the sending of the message to the coordinator (1 message),
- the distribution of the message to all members via a unicast ($n - 1$ messages),
- and the processing time at all nodes.

Neglecting the processing time, in total n messages are required. Further, it is assumed that the sequencer protocol is deployed on top of the NACK protocol, i. e. no acknowledgements must be considered.

While both protocols are comparable from the response time point of view, the average costs per message differ strongly under high load. In this case the number of messages remains the same for the sequencer. For WS RRP the number of messages is reduced to a constant effort if each replica adds a single update message to the token. Assuming that in this scenario n

messages can be delivered with each token circulation (n messages) the total costs per message are reduced to 1.

In case of a node failure, WS RRP demands at least two join rounds and one token circulation for the commit phase (cmp. section 7.3.1). The number of required messages comprises:

- $(n-1)\cdot(n-1)$ messages for sending initial joins to all members of the old view,
- $(n-1)\cdot(n-2)$ messages for exchanging joins within the members of the new view,
- and n messages for commit token circulation.

Hence, the total costs for execution of the membership protocol are in the order of $O(n^2)$.

The GMS of JGroups similarly requires two rounds of message exchanges:

- $2\cdot(n-2)$ messages for sending the new view to all members and receiving the acknowledgements, and
- $2\cdot(n-2)$ messages for sending the commit message to all members and receiving the acknowledgements.

However, in contrast to WS RRP, messages must solely be exchanged between all members and the coordinator. Hence, the costs are only $O(n)$.

An important question for actively replicated systems concerns the number of replicas used for ensuring the availability of the system. This issue applies to WS RRP as well as JGroups. The next section discusses the different trade-offs that must be considered.

7.3.4 Degree of Replication

The aim of replication is to increase the reliability and availability of a system in case of failures. In general, with an increasing number of replicas the reliability of a system grows since more replicas that can mask a failure are available. The number of failures that can be tolerated by the Replication Service is mainly determined by the majority consensus protocol used for determining the primary partition. This protocol permits a continuous operation as long as less than $\lfloor \frac{n}{2} \rfloor$ fail. In case $\lceil \frac{n}{2} \rceil$ or more nodes fails, a total failure of the system occurs, i. e. a reparation of the system is required.

While with an increasing number of nodes the number of non-failure states increases, the probability of a failure also rises. Jalote [173] modeled the availability in dependency of the MTTR and the number of replicas (n). It was found that the optimal number of n is quite large if the MTTR is small. For example, assuming a MTTF of 30 days per node and a MTTR of 1 day, the optimal number of 15 replicas achieved an availability of 99.97 %. However, the model also showed that by reducing the MTTR, good results can be obtained with a significantly smaller number of replicas, e. g. 3 or 5.

Another important aspect is the response time, which is in general directly correlated to the number of involved nodes. Each update requires the transmission of the update message to

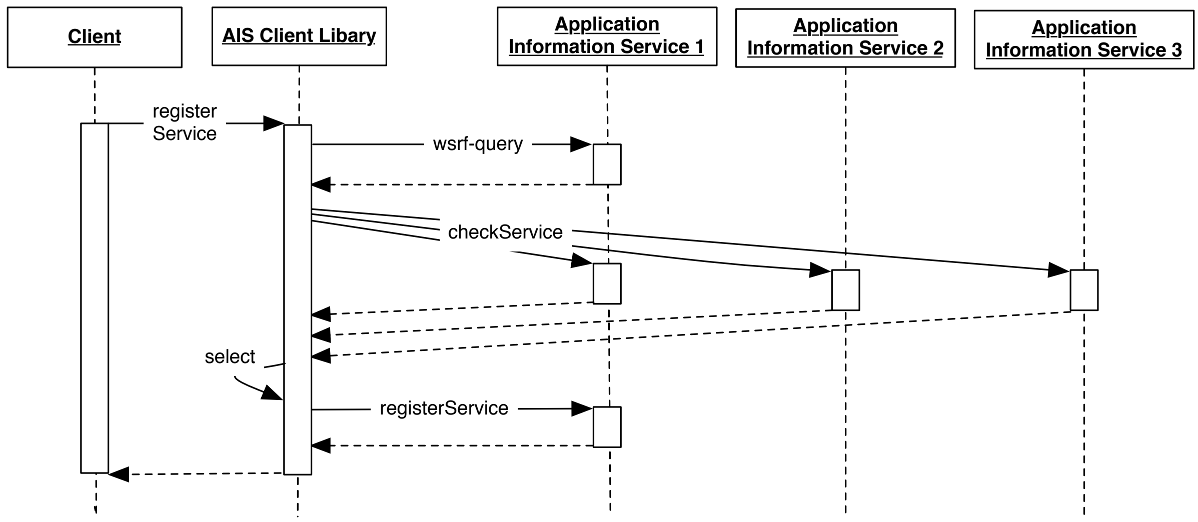


Figure 7.7: AIS Load Balancing

all $n-1$ nodes via the total ordering protocol as well as a fixed number of actions on each node. Depending on the protocol and the desired consistency guarantees at least linear costs are associated with replication. Thus, active replication is in general only scalable to a limited number of nodes. Several studies, such as done in context of the Chubby lock service [66] showed that using five replicas is a good trade-off between availability, reliability and performance of the system. This is also confirmed by the experiments conducted with the AIS presented in section 9.2.

The capability to discovery AIS instances is critical to ensure that clients can fail over to another AIS instance in case of a failure. Further, this feature enables the deployment of a load balancing mechanism. In the following section, both issues are described.

7.3.5 AIS Discovery and Load Distribution

Another important problem of replicated systems is the establishment of replication transparency [302]. In case of the AIS this is achieved by relying on a set of well-known and highly available AIS instances. The AIS client library requires that such a configured list is initially provided. Figure 7.7 illustrates the startup process: During the first initialization, a list of the current AIS group members is obtained via the resource property interface of the well-known AIS. This information is used to update the known AIS list, which is stored persistently in the user's home directory. Later, this list can be used to discover an alternative AIS in case of a failure.

Another benefit of active replication is that the load can be balanced among the group of AIS processes. Thus, the AIS library incorporates a simple algorithm to distribute requests among AIS server instances. The algorithm dispatches a request to all servers of the group in order to find the server with the fastest response time [179]. This server is likely a good choice taking into account indirect factors, such as the message latency and the load of the server. Using the current member list obtained before, the AIS client library sends a `checkService` request concurrently to all members of the group. The member with the shortest response time is chosen for all further requests. In the example presented in Figure 7.7 the `registerService` request is forwarded to this instance. Due to the overhead involved, load balancing is only done once per client – all subsequent requests are routed directly to the initially selected AIS.

7.3.6 Replication of the Monitoring and Recovery Service

Since recovery is essential for a fault tolerant Grid, the Monitoring Restart Service (MRS) also represents a critical component. To avoid a single point of failure, the MRS is deployed redundantly similar to the AIS. However, the replication of the MRS is complicated: An active replication would lead to additional monitoring traffic since all MRS instances would attempt to monitor all applications. Even worse, in case of a failure different MRS instances would attempt to recovery applications concurrently.

To avoid this issue the MRS relies on the primary-backup mode provided by the Replication Service (RS). This mode ensures that only one instance of the MRS is active at a time. To select the active MRS, the following convention is used: The RS with the smallest ID always activates the MRS – all remaining MRS instances are not activated and operate in stand-by mode.

7.4 Summary and Discussion

The Migol architecture aims to carefully balance the trade-off between performance and fault tolerance. Therefore, Migol reuses different preexisting fault tolerance techniques to create a fault resilient infrastructure for supporting the recoverability of Grid applications. The key concept for achieving fault tolerance is modularization and encapsulation of capabilities into separate services. All Migol services have been classified according to their criticality. Core services of Migol such as the Application Information Services (AIS) and the Monitoring Restart Service (MRS) are replicated. In particular, active replication is used to ensure the high availability of the AIS.

However, while active replication with strong consistency guarantees is a desirable feature, it must be carefully considered whether the increased complexity and overhead is beneficial to the overall goal. Not all services need to provide continuous operation – in most cases it is sufficient to support a backward recovery. Backward recovery is supported by all Migol services - in conjunction with the re-connection capabilities that have been build into the Migol library a reasonable level of fault tolerance can be achieved.

As described, Migol uses active replication to ensure the fault tolerance of its core service, the AIS. The AIS maintains the state of all Grid applications and services. For example, all users and Migol itself rely on the AIS for locating applications. Thus, the AIS is a

	WS RRP	**JGroups Sequencer**
Protocol	ring-based (totem)	central coordinator
Extendability	plugin architecture	micro-protocol stack
Security	GSI	proprietary protocol
Communication Protocol	SOAP	UDP/TCP
Response Time	$n + n/2$	n
Replication Costs	Best Case: $O(1)$ Worst Case: $O(n)$	$O(n)$
Membership Service Costs	$O(n^2)$	$O(n)$

Table 7.2: WS RRP/JGroups Sequencer Comparison

highly critical service. The Replication Service provides an extensible API for different group communication frameworks. Currently, two frameworks are supported the Web Service Ring Replication Protocol and the JGroups sequencer. WS RRP uses a token replication algorithm similar to Totem, while the JGroups sequencer stack relies on a central coordinator to ensure total ordering. Table 7.2 briefly summarizes the key features of the WS RRP and JGroups sequencer protocol. The main difference arises from the different technologies and protocols used within the frameworks. While WS RRP provides a plattform independent, secure Web service interface, JGroups relies on a proprietary protocol both for messaging and security. However, it must be noted that Web service based communication is associated with some overhead (see section 9.2), which is in particular critical for protocols, which require the exchanges of many messages.

From the algorithmic point of view, WS RRP can especially benefit from high loads, which reduces the replication costs to a magnitude of $O(1)$ – in contrast, the sequencer protocol always requires costs of $O(n)$. Both replication frameworks detect network as well as node failures using timeouts. Failures are resolved using a membership protocol. A majority consensus protocol ensures that in case of a network partition only one active ring exists.

While active replication is useful to achieve continuous availability, it is restricted to a certain fault model. Replication always assumes that faults occur independently - this assumption can be fulfilled e. g. for most faults if service instances are deployed on Grid sites in different administrative domains. However, with the distribution of the service instances, failure detection becomes more difficult due to the high latencies that must be expected. Thus, the management of timeouts becomes highly critical. In general, the described protocols are only able to handle inconsistent failures, such as slowdowns, to a certain extends. In the worst case, such failures can lead to an indefinite number of join rounds without agreement in the case of WS RRP.

Another concern often addressed in context of service replication is scalability [148]. As described, active replication is associated with different overheads for synchronization of replicas and group membership. Thus, it must be ensured that the performance of the replicated service can still meet the requirements of the application. In chapter 9 the performance of WS RRP and JGroups is extensively analysed.

Last, it must be considered that replication increases the overall complexity. In general, replicated services have a much more complex and unexpected failure behaviour. Thus, replication must be carefully considered. In the case of Migol only the most critical part of the infrastructure uses this fault tolerance schema.

"It is easier to write ten volumes on theoretical principles than to put one into practice."

Tolstoy

8 Implementation

In this section, the implementation of the Migol service framework is described. However, due to the complexity of the systems – Migol currently consists of 173 Java and C/C++ files, which account to about 45,000 lines of code (not including the WSDL files, generated code or any Migol application) – it is not possible to describe every detail.

After highlighting important implementation guidelines in section 8.1, the high level object and class design of a typical Globus Web service is described in section 8.2. As example Migol's Job Broker Service is used. Section 8.3 and 8.4 focus on the service reliability and availability aspects of the implementation. In particular, section 8.4 describes the implementation of the replication framework used within the AIS.

Another important part is the Migol library and the SAGA adaptor. Both are presented in section 8.5. Using these interfaces, Migol has been integrated into several MPI and non-MPI applications. In particular, the applications described in section 4.3, i. e. the Cellular Automaton, AMIGA, Cactus and the REMD-Manager, have been successfully ported to Migol. As representative example, section 8.6 describes the implementation of the REMD-Manager application. This application is especially interesting since it extensively uses Migol to distribute and monitor worker tasks. If a failed task is observed, Migol automatically handles the recovery of this task. In contrast, Cactus, AMIGA and the Cellular Automaton are rather monolithic from the Migol perspective: Although these three applications can be run across multiple nodes, they mainly rely on MPI to manage their distribution.

8.1 Implementation Considerations

For the implementation of the system, common practices, such as the known object oriented design patterns (see Gamma et. al. [134]), the Globus WSRF design recommendations as well as the fault tolerance principles described in section 7.1, are reused. All Migol services are composed of modular components. As described, modularization is a key principal for achieving a flexible, maintainable and reliable system [251]. By using a plugin architecture important modules can be easily exchanged. Further, defensive programing techniques are used, e. g. to ensure the consistency of passed parameters and messages.

When designing and implementing a distributed application framework, the different peculiarities of distributed systems must be considered. Migol's design presented in chapter 6 and 7 addresses complex important issues, such as failure detection, consensus, and security. However the implementation of such an infrastructure is a complex task:

- While the used Web service framework (Axis/Globus) provides a simple to use abstraction for exchanging messages between Grid sites, issues, such as failure detection and consensus, requires a great amount of experience and a lot of tuning effort. In particular, the management of timeouts is critical for the performance of the systems.

- In general, distributed architecture are highly concurrent. Adding the necessity to use threads for implementing algorithms such as WS RRP, the system becomes quite complex. While Globus/Java provides a simple to use abstraction such as the Work API, the developer still has to deal with deadlocks, race conditions etc. In particular, concurrent code requires a large amount of manual testing. Known test frameworks, such as JUnit, are only partially suitable for this.

- Despite the specification of a fail-stop fault model for Migol, experience showed that real world faults can be arbitrary manifold. In particular, during operation of the AIS within the D-Grid and LONI the importance of a broader failure model became obvious. In certain cases the slowdown of a single AIS node lead to an overall stalling systems. Only a manual tuning of the timeout parameters allowed the AIS to cope with such failure modes. However, extended test in production environments have been conducted, which lead to an optimization of Migol (see chapter 9).

Migol was initially developed on basis of the Globus Toolkit 3 and the Open Grid Services Infrastructure (OGSI) [311], the predecessor of WSRF. With the convergence of Grid and Web Services standards to WSRF and the release of the Globus Toolkit 4, Migol has been refactored [214].

The framework has been extensively tested under Mac OS 10.5 and Linux (using a 2.6 Kernel) and Java 5 SDK – however, it should be able to run on any GT 4.0 compliant machine. Due to the release policies of the Globus team, the newly released Globus Toolkit 4.2 is not backward compatible, i. e. at the moment Migol is not compatible with this version. However, due to the clear separation of application logic and the service interface, the porting to GT 4.2 should be straightforward.

8.2 WSRF Service Implementation: Job Broker Service

In the following, the important implementation steps for creating a WSRF service are presented:

- The service interface is defined using WSDL [81]. This involves in particular the usage of the WSRF interface [93], such as the WS-ResourceProperties port types, for exposing and manipulating of the service state.
- The WSDL description is used to generate stub and skeleton classes. In case of WS container of GT4, Java is used as implementation language. The generated classes implement the proxy pattern [134] and encapsulate the remote call details, i. e. they transparently handle the object marshalling, de-marshalling, communication as well as the dispatching to the respective remote method. This technique has formerly been described as remote procedure call (RPC) [55].
- Further, the service implementation in addition to different deployment parameters must be provided. Then, the service can be deployed in the Globus container.

The Web service container of the Globus Toolkit is based on Axis1 [38]. Globus extends Axis1 by several handlers and libraries to support e. g. WS-Security based on X509 proxy certificates. The service developer has various options to configure declaratively the used security mode for authentication, authorization and credential delegation. In addition, further service configuration data can be stored in the central naming service of Globus. These data can then be accessed via the Java Naming and Directory Interface API (JNDI) [174].

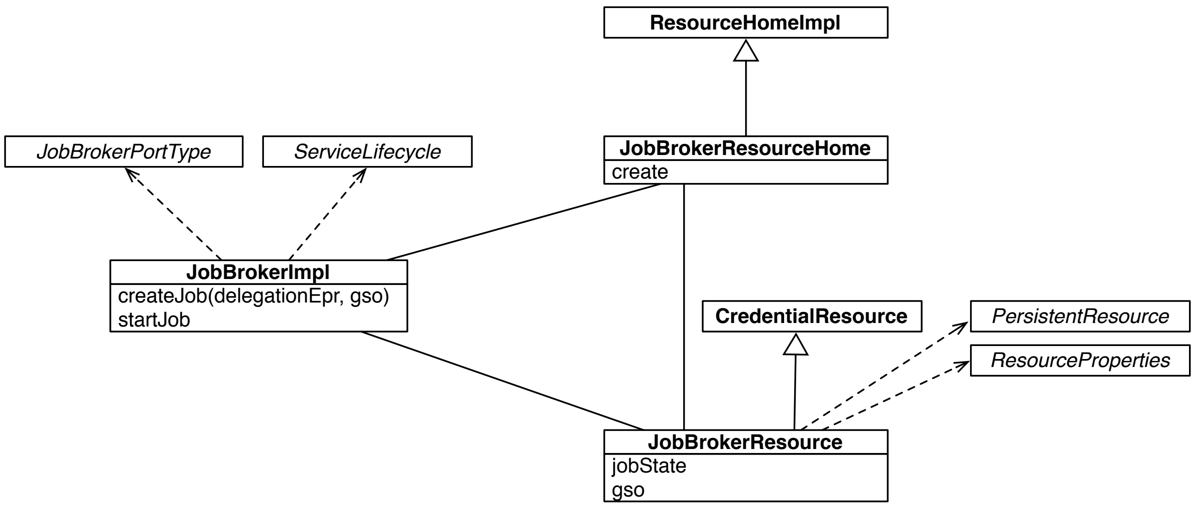


Figure 8.1: Job Broker Service: Core Service Classes

In general, all WSRF services of Migol follow the same implementation pattern. Figure 8.1 shows the class diagram of a typical WSRF service on example of the Job Broker Service. However, this similarly applies to the ARS, CRS, and MS.

The main application logic is provided by the `JobBrokerImpl` class. This class implements the service interface `JobBrokerPortType` generated from the WSDL service description. Further, the `ServiceLifecycle` interface is implemented – this interface is used to register further initialization and finalization logic. These callbacks are used e. g. to register and de-register the service with the AIS.

The `createJob` method of the service class is used as resource factory. Resources are created via the resource home class. In this example the `JobBrokerResourceHome` class is used to instantiate a `JobBrokerResource` (cmp. factory pattern [134]). After successful creation of a resource an endpoint reference (EPR) is returned to the user. An EPR contains both the service URL and the resource key, which uniquely identifies the resource. The EPR is used in all subsequent calls to interact with the service, e. g. to start the job or to query the job state. This pattern is also referred to as implied resource pattern [93].

The state maintained for each job mainly consists of the overall job state (see state chart in Figure 6.8) and the current Grid Service Object. It should also be noted that the entire state is also stored in the GSO of the application at the AIS. Thus, the GSO in the `JobBroker-Resource` solely represents a cached copy. This approach ensures, that the state is available even in case of a failure of a JBS service.

In general, the creation of a Grid job requires the delegation of the user credential. The credential is e. g. required to allow the submitted application to authenticate with the Migol backend. For performance reasons, the GT4 Delegation Service is used to pass the credential to the service container. The service resource is then able to obtain the credential from the container. This functionality is encapsulated in the base class `CredentialResource`.

The other methods of the service class provide the different service functionalities. For example, the `startJob` operation is used to start the job, i. e. the method initiates the scheduling and starting sequences illustrated in Figure 6.6. In addition to the classes depicted in Figure 8.1 several utility classes exists, which encapsule e. g. JNDI lookups, the loading of plugin classes or the communication with other Globus services such as the WS GRAM.

As emphasized in section 7 all Migol service are recoverable, i. e. the state of all resources can be re-created after a failure. In the following, some details regarding the implementation of the recoverability feature are given.

8.3 Service Recoverability

All Migol services, i. e. the ARS, CRS, JBS and MS, support recoverability. For this purpose, all resources implement the `PersistentResource` interface provided by the Globus Toolkit 4. The interface extends the three interfaces `Resource`, `ResourceIdentifier` (both not shown in Figure 8.1) and `PersistentCallback`. The `PersistenCallback` interface provides two methods for resource persistence: `load` and `store`. This interface can be compared to an application-level checkpointing interface. The resource implementation is responsible for persisting and loading the resource state. The JBS, MS, ARS and CRS simply use the Java object serialization mechanism to store the resource state in the `$HOME/.globus/persisted/` directory, which is also used by other Globus services. To save a resource state, the `store` method must manually be called after every modification.

After a container crash, the resource state is automatically recovered when a resource is accessed via its ResourceHome interface. However, it must also be ensured that background tasks are restarted after a failure. For example, the JBS recovery manager restarts the monitoring thread, which monitors the job starts and restarts a failed job if necessary. The CRS recovers the transfer monitor background thread, which periodically polls the transfer state from the RFT and restart transfers on demand.

8.4 AIS Replication Framework

The AIS represents the core of the Migol architecture. The AIS maintains the state of all applications and services in the Grid. To ensure the availability of the AIS, the service supports different replication schemas. Figure 8.2 shows the class diagram of the AIS implementation. Similarly, the service class AISImpl main class is derived from the generated AISPortType class. This class implements the business logic of the AIS, i. e. it provides methods for managing service registrations. In addition, several administrative operations for the maintenance of users are provided. All service states are stored persistently using a file-based storage or a relational database. Hibernate [163] is used for the object-relational mapping of data objects, i. e. the service and user object, onto tables of the relational database system. The framework supports several databases, e. g. MySQL, PostgreSQL and Oracle.

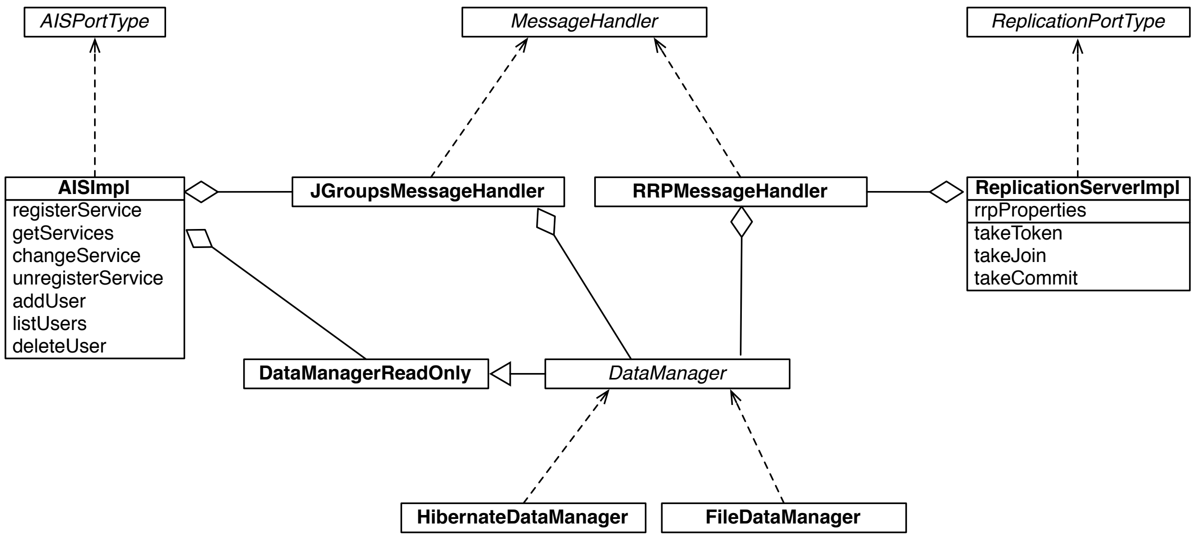


Figure 8.2: AIS Class Diagramm

Due to performance reasons, read operations are only conducted at the local AIS. For this purpose, the DataManagerReadOnly class is used. Update operations are executed virtual synchronously at all replicas. To achieve strong consistency all update messages must be distributed totally ordered to all process. The MessageHandler interface represents the abstraction toward the group communication system. The interface provides two simple methods for virtual synchronous execution of an operation, a blocking and a non-blocking one. Depending on the configured plugin the respective implementation is chosen. Currently, two plugins exists: the WS RRP and the JGroups plugin. Both are explained in the following sections.

8.4.1 WS RRP

WS RRP implements the Totem style protocol introduced in section 7.3.1. All message formats and interfaces of WS RRP, i. e. the format of join, commit and regular tokens, are defined within a WSDL file. The `ReplicationServerImpl` implements the generated `ReplicationPortType` interface, which is equivalent to the WSDL description. The state machine implementing the WS RRP protocol is provided within the `ReplicationServerImpl` class. State updates are conducted using the `RRPMessageHandler` class, which relies on the `DataManager` plugin for persistence.

Toke exchanges are conducted via the `takeToken` operation. For the implementation of the membership service the methods `takeJoin` and `takeCommit` are used. In case of an error, a join round is triggered by calling the join operation of all members. After reaching consensus, the commit token is distributed via the `takeCommit` operation.

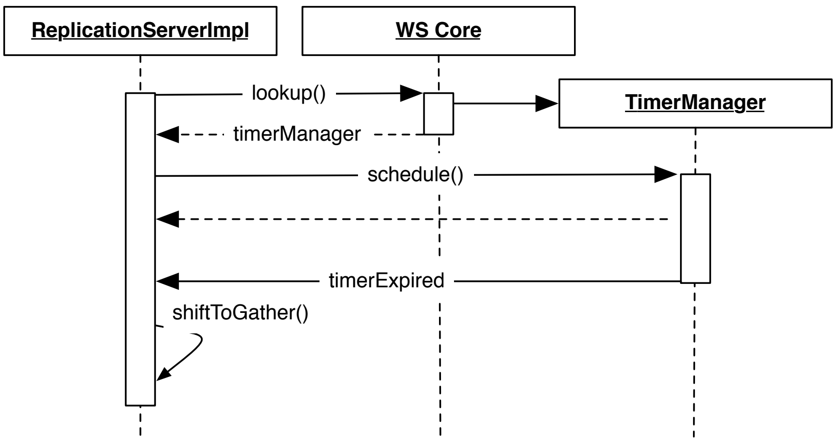


Figure 8.3: Replication Service: Timeout Management

A critical aspect when designing fault tolerant systems is error detection. As described in section 7.3.1 the timeouts for the different operations are determined dynamically. For enforcement of the timeouts, the JEE Concurrency API, for which a preliminary implementation is provided within Globus, is used. Figure 8.3 illustrates how this API is used. After obtaining a reference to the container's `TimerManager` class, a callback can be scheduled after a certain timeout. The timeout for the callback is determined based on the an exponential average of the last token roundtrip times and the number of messages. The timer manager ensures that the callback method is called after expiration of the timeout. In case of a timeout, a reconfiguration is initiated.

Another important aspect are the timeouts deployed for the remote calls for passing the regular token or sending join messages. Unfortunately, the default timeouts used by Globus/Axis

– during tests the detection of a failure took up to 10 minutes – are not suitable for WS RRP, which requires a more rapid error detection. Especially during join rounds such a long timeout leads to a long delay in which the AIS is blocked. Thus, the setting of a smaller timeout is required.

Globus provides the capability to adjust the timeout values of the remote procedure stubs. Listing 8.1 shows how custom timeout can be used.

```
int timeout = AISConfiguration.getTimeout();
ReplicationServiceAddressingLocator locator = new
   ReplicationServiceAddressingLocator();
ReplicationServerPortType repService = locator.
   getReplicationServerPortTypePort(endpoint);
((org.apache.axis.client.Stub) repService).setTimeout(timeout);
```

Listing 8.1: WS RRP Timeout Management

In the following section, the JGroups plugin, which can be used as alternative to WS RRP, is presented.

8.4.2 JGroups

JGroups provides a simple abstraction to a process group via the `Channel` object. Listing 8.2 shows the initialization of the JGroups stack in the `JGroupsMessageHandler` class. To setup the micro-protocol stack, the channel object must be initialized with an XML-based configuration file. The configuration file reflects the stack defined in Table 7.1

```
channel = new JChannel(aisConfig.getConfigFile());
messageListener = new MessageListenerAdapter(new AISMessageListener());
disp = new RpcDispatcher(channel, messageListener, this, this);
channel.connect(JGROUP_NAME);
```

Listing 8.2: JGroups Initialization

Further, a `MessageListenerAdapter` is registered – this message listener is used to initialize the `RpcDispatcher` component. The `disp` object can then be used to enable the invocation of a remote method by all group members at the same time. Listing 8.3 illustrates the invocation of the `handleMessage` method at all group members. The first parameter can be used to restrict the message receivers to a certain group member set. In the example null is used to address the entire group. The `callRemoteMethods` method will return, after all replicas have successfully completed the `handleMessage` method.

The message listener `AISMessageListener` is further used to receive notifications about message receptions and to enable a state transfer. In contrast to WS RRP, which allows new members to request old messages via the retransmission request field, JGroups requires the implementation of a custom state transfer mechanism. Listing 8.4 illustrates how the state transfer mechanism of JGroups works. After establishing a connection to the process

```
rsp_list = disp.callRemoteMethods(null, "handleMessage", param,
                        types, GroupRequest.GET_ALL, 0);
```

Listing 8.3: JGroups Update Propagation

group, new processes can obtain the group state via the `channel.getState()` method. Pre-requisite for this approach is that the message listener implements the `getState` and `setState` method properly.

```
boolean join = channel.getState(null, 1000001);
if (join) {
        log.debug("State␣transfer␣successful");
}
```

Listing 8.4: JGroups State Transfer

The JGroups plugin shows, how simple new group communication frameworks can be integrated into the AIS. Using this schema the AIS can easily be extended to support other group communication frameworks, such as Appia [226] or Spread [24].

8.5 Migol Client Library and SAGA

Further, the APIs and client libraries provided by Migol are important for the integration of Grid applications. Part of Migol is a lean C library as well as an adaptor for the SAGA C++ reference implementation. Both are described in the following sections.

8.5.1 Migol Client Library

Figure 8.4 shows the interface provided by the Migol library. The library provides routines for registering application metadata with the AIS and the triggering of checkpoint replications at the CRS. Further, the library starts a background thread, which is used for monitoring the

Migol Library
char* MIGOL_Init(char* guid);
void MIGOL_Finalize();
int check_point();
int check_service();
int destroy();
char* register_service(char *url, char *service_name, char *state);
int unregister_service(char *guid);
int change_service_state(char *guid, char *newState);
int register_checkpoint (char *guid, char *fileName);
int delete_checkpoint (char *guid, char *fileName);
int replicate_checkpoints(char *guid, int automaticReplication);
int delete_checkpoint_replicas(char *guid);

Figure 8.4: Migol Library Interface

application via a HTTP/SOAP endpoint. For the implementation of the SOAP endpoint the gSOAP toolkit [114] is used.

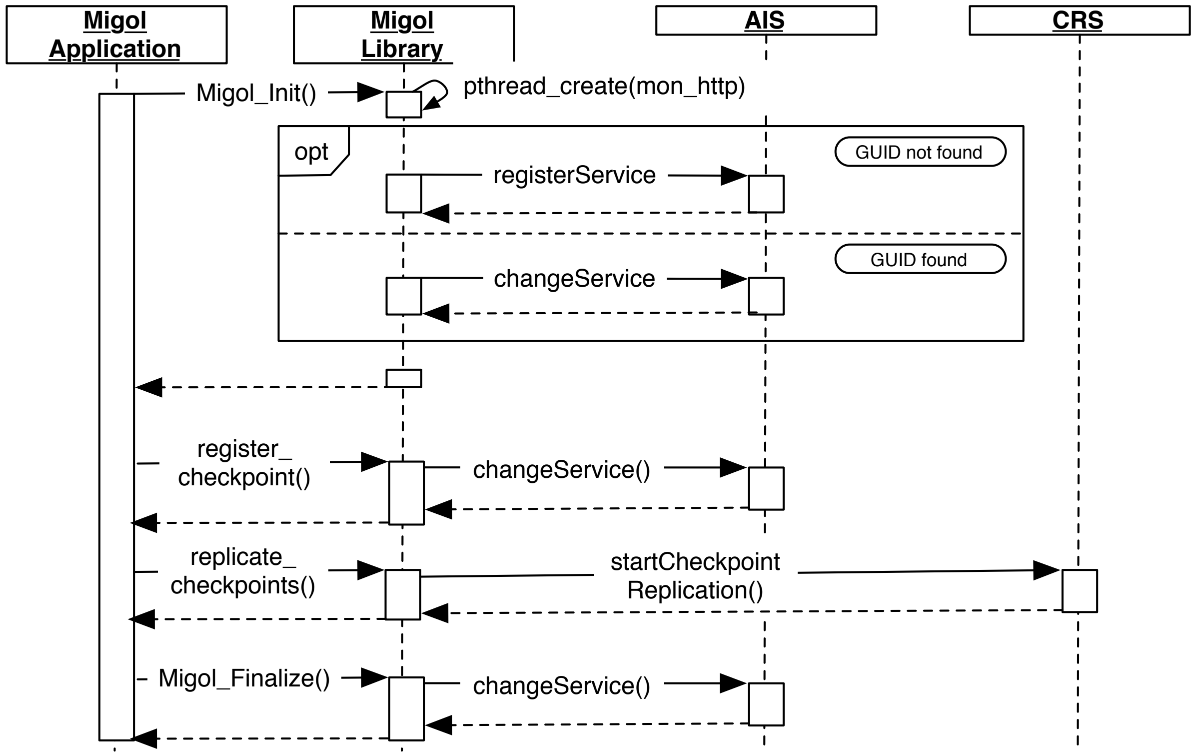


Figure 8.5: Interaction Application, Migol Library and Grid Service Infrastructure

Figure 8.5 illustrates the interactions between an application and the Migol library. With the call to `Migol_Init()` the Migol library starts a monitoring thread in the background. After this thread has been successfully started the new monitoring endpoint must be registered with the AIS. To obtain the current Grid Service Object, the Migol library tries to detect the GUID of the application. In general, the GUID can be passed as parameter to `Migol_Init` or via the environment variable `MIGOL_GUID`. If a GUID is found, the service state of the referenced GSO is changed to active and the new monitoring endpoint is added to the GSO. Otherwise a new application is registered at the AIS. During its runtime, an application can update its GSO by adding and removing checkpoints. The Migol library will modify the respective file profiles in the GSO of the application.

The Migol library is implemented in C, but relies on Java for realization of the communication with the Globus WS Core, which hosts the Migol services. This way the existing Java client libraries can be re-used. Further, all features such as credential delegation, which is not supported by the gSOAP GSI plugin, can be used. Since Java is an installation requirement for Globus, the portability of the Migol library is not restricted. The Java libraries are accessed via the Java Native Interface (JNI) [1].

The Migol library represents a minimal solution for accessing the Migol backend. A more comprehensive solutions is provided via the SAGA framework.

8.5.2 SAGA

The SAGA API consists of two parts: the Look & Feel and the API packages. Each API package defines an API for a certain aspect of a Grid, e. g. for job, file or replica management. The SAGA C++ reference implementation [182] uses a lightweight engine to decouple the API from the middleware specific implementation. All interactions with the Grid middleware is encapsulated in a so called adaptor. Each adaptor implements one or more Capability Provider Interface (CPI). A CPI represents the interface to an API package. The SAGA engine dispatches API calls to dynamically loadable adaptors.

The Migol adaptor implements the SAGA CPR package adaptor [223], which was inspired by the GridCPR draft [290] and Migol itself. The adaptor is designed as a wrapper for the existing Migol library. Figure 8.6 shows the initialization of the Migol adaptor. With the creation of a CRP service object (`saga::cpr::service`) the saga engine is initialized. The Migol adaptor CPI creates an instance of the `migol` object. The `migol` object starts the background monitoring thread, which runs the gSOAP HTTP server, in its constructor.

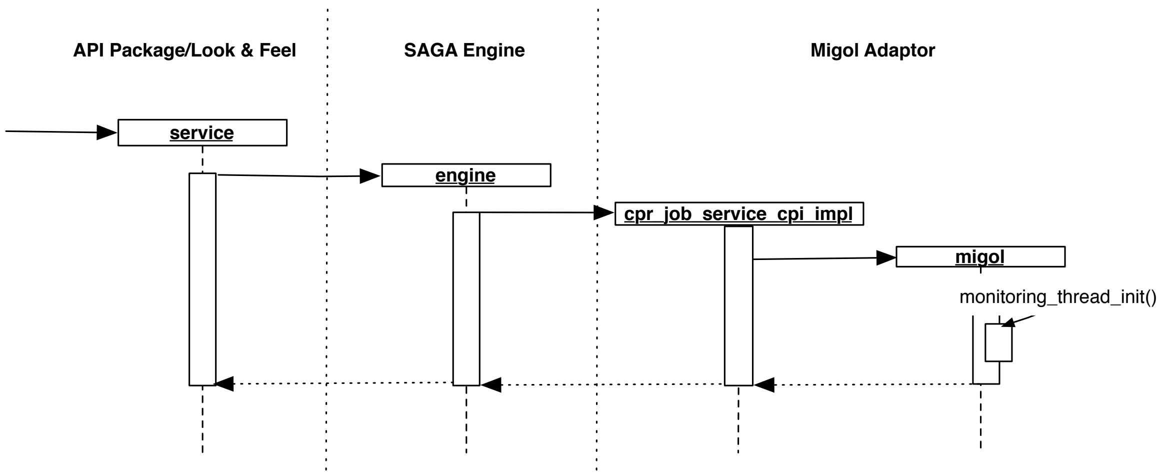


Figure 8.6: Initialization of the Migol Adaptor

Analogous to the Migol library, SAGA calls are routed via the Migol adaptor the the Migol backend, i. e. the AIS and CRS. For the management of checkpoint metadata the SAGA CPR API uses the namespace paradigm to hierarchically organize files and directories. Checkpoint directories are used to group checkpoints. Parallel applications often write one checkpoint per processor, i. e. in case of n processors a CPR checkpoint would consists of n files. Thus, a CPR checkpoint is designed as container for multiple physical or logical files. Listing 8.5 demonstrates how an application task registers a single checkpoint file with the CPR implementation, i. e. Migol.

```
saga::cpr::checkpoint remd_chkpt("remd_chkpt");
remd_checkpoint.add_file(saga::url("gsiftp://queenbee.loni.org/work/remd/
    remd_chkpt.dat"));
```

Listing 8.5: SAGA CPR: Register Checkpoint with Migol

In addition to the management of checkpoint metadata, the CPR API and the Migol adaptor also supports the management of Grid jobs via the API. For execution and management of recoverable jobs, SAGA defines a CPR job class. The management of CPR jobs is similar to a regular SAGA job: A job is defined by a job description. In contrast to normal jobs, CPR jobs require two job descriptions – one for starting and another one for restarting the application. Jobs are started using the job service. In addition to the normal job controls, a CPR job can be queried for checkpoint metadata, and it can be explicitly checkpointed or recovered. Listing 8.6 shows the start of a CPR enabled job.

```
saga::cpr::service js;
saga::cpr::description jd_start;
saga::cpr::description jd_restart;
//file out job description
...
//submit job
saga::cpr::job job = js.create_job (jd_start, jd_restart);
job.run();
```

Listing 8.6: SAGA CPR: Starting a Job with CPR Support

Having introduced the SAGA API for Migol, the REMD-Manager which uses this API for the management of application tasks is presented.

8.6 Applications Scenarios: REMD-Manager

Migol has been successfully used with the applications presented in section 4.3. The integration of Grid applications with Migol usually follows the same schema. Since the REMD-Manager heavily uses SAGA/Migol for distributing and monitoring of many applications tasks, the implementation of the REMD-Manager is detailed in this section.

As illustrated in Figure 8.7, the REMD framework comprises of three components: The task manager, also referred to as *REMD-Manager*, is deployed on the user's desktop and provides the user interface to the overall REMD run. The second component is the Migol infrastructure that submits, monitors, and if required, recovers simulations. The last element is the task agent, the *NAMD-Launcher*, that resides on the High Performance machines where MD simulations are carried out. The NAMD-Launcher is triggered by the Grid job and is responsible for spawning and monitoring the MD run. NAMD [257], a highly scalable, parallel MD code, is used to carry out the MD simulation corresponding to each replica run. It is important to mention that any other MD code could be used just as simply and effectively.

The *REMD-Manager* is the core of the framework; it orchestrates all replicas, i. e. the parameterization of replica tasks, file staging, job spawning and the conduction of the replica-

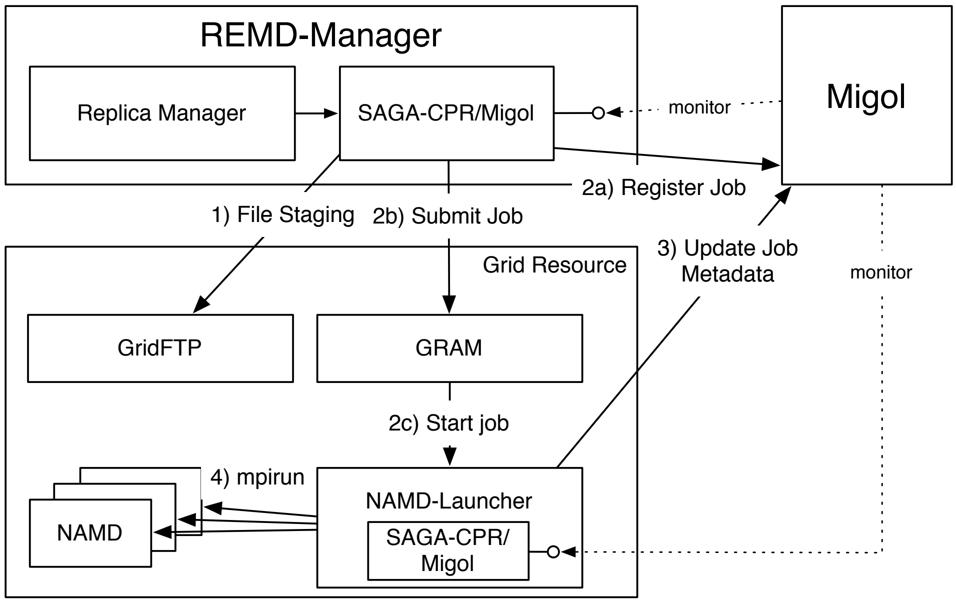


Figure 8.7: REMD-Manager Architecture

exchange itself. This component heavily relies on the SAGA File and CPR API as well as the Python bindings for the implementation of the RE logic.

Depending on the number of configured processes n, the REMD-Manager starts initially $\frac{n}{2}$ pairs of replicas. Each replica process is assigned a different temperature. Before launching a job the REMD-Manager ensures that all required input files are transfered to the respective resource. For this purpose, the SAGA File API and the GridFTP adaptor (step 1 in Figure 8.7) are used. The replica job is then submitted to the Grid resource using the CPR API and Migol/-GRAM (step 2a-2c). Migol ensures that the the job description of each replica is stored within the Migol backend to ensure a later recovery. Globus GRAM is used to start the application.

To integrate NAMD with the SAGA/Migol infrastructure a SAGA based task agent – *NAMD-Launcher*, is used. This agent is responsible for updating the metadata of the application, i. e. the state, monitoring endpoint and new checkpoint URLs, at the Migol backend. The agent then launches the actual NAMD job using MPI. During the entire runtime the replica process is monitored by Migol using the monitoring endpoint of the NAMD launcher. This NAMD-Launcher enables the flexible orchestration of multiple NAMD jobs through the REMD-Manager without modification of the NAMD source itself.

When paired replicas reach a pre-determined states (eg., after a fixed number of steps), the decision as to whether to exchange paired-replicas is determined by the Metropolis scheme. If successful, parameters such as the temperature, are swapped. Both jobs are then relaunched using the mechanisms described above.

8.7 Summary and Discussion

The Migol framework has been carefully implemented reusing existing patterns and frameworks whenever possible. For example, Migol heavily reuses the WSRF and GSI framework provided by the Globus Toolkit 4.0. At the same time, the implementation attempts to encapsule the core service logic in separate classes decoupling the service implementation as far as possible from the Grid middleware. This approach ensures that the framework can easily be ported to a new Globus version or another middleware platform.

The Migol framework has been integrated with several scientific applications, e. g. AMIGA, Cactus and the REMD-Manager. Grid applications have two options for accessing the services of Migol. Most flexible of all is the standardized SAGA programming API. With SAGA applications are able to reuse different well-defined programing abstractions, e. g. for job management and file transfers, in conjunction with the CPR interface for Migol.

The REMD-Manager is a representative example for the use of SAGA/Migol. SAGA allows the simple decoupling of the REMD application and orchestration logic from the underlying distributed infrastructure. At the same time the REMD-Manager remains general purpose and extensible, for example, using the Migol adaptor, the application can benefit from features, such as the automatic monitoring and the transparent recovery of failed tasks.

The components developed in this thesis have been extensively tested. For all critical classes JUnit [9] test cases have been developed. However, in particular for testing of concurrent parts of Migol, as e. g. the AIS, several manual tests have been necessary. As described in the following section, Migol has been deployed in various Grid settings, which significantly improved its robustness.

A theory is something nobody believes, except the person who made it. An experiment is something everybody believes, except the person who made it.

Albert Einstein

9 Grid Experiments

The performance und reliability of a system are closely related. In general, fault tolerance is associated with some costs in terms of performance. To evaluate the overhead associated with the Migol framework, several experiments within real production Grid environments, such as the German D-Grid and the LONI Grid, have been conducted.

This chapter will summarize the various experiments. Since Migol is based on the Globus Toolkit, it is particularly important to initially understand the performance impact of typical Web service operations. The results of this assessment is presented in section 9.1. In section 9.2, the performance of the AIS replication framework is investigated. This chapter is concluded with an evaluation of the overhead imposed by Migol on Grid applications. For this purpose, experiences with the REMD-Manager application are presented in section 9.3.

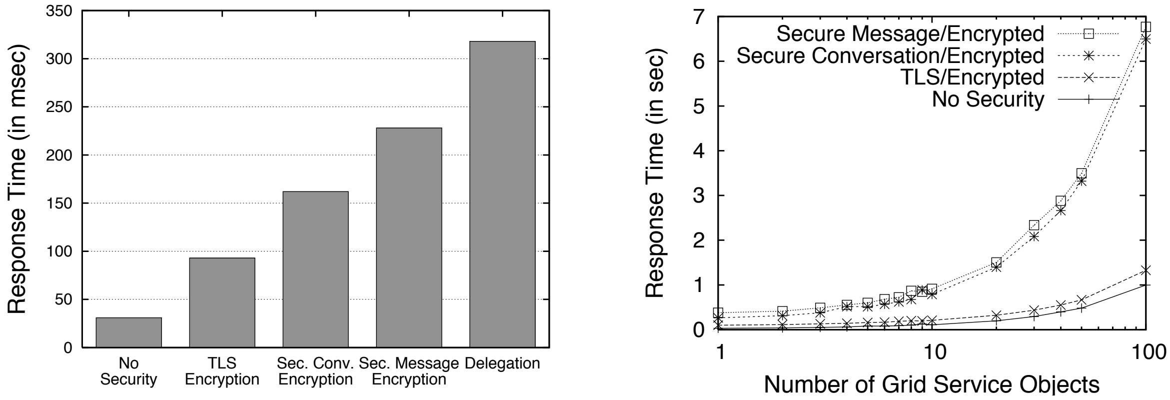
9.1 Web and Migol Service Performance

A challenge in such service-oriented environments consisting of many composable services, i. e. services which are based upon other services, is performance. Thus, the performance of the Web service stack is particularly important. For example, the submission of a Migol Grid job requires several service interactions for registering data at the AIS, reserving resources and the conduction of the actual job start.

In the following the performance of the GT4 SOAP implementation and of the Migol services is evaluated. For this purpose a dedicated Globus machine with 1.6 GHz AMD processor and 1 GB of RAM is used. The Migol services are deployed in Tomcat 5/Globus Toolkit 4.0 (GT4) containers running on Linux with a Kernel version 2.6. The Java Virtual Machine (Java 5) is configured with a heap space of 512 MB.

9.1.1 Globus Web Service Stack

A common Migol use case is the transfer of Grid Service Objects (GSO) as input or output parameter for a service. The used micro benchmark consists of a Web service with a simple echo operation, which accepts a list of GSOs as parameter. This service is called with different numbers of GSOs as input; each GSO has a serialized size of 3.1 KB.



(a) Transport-level versus Message-level Security

(b) Message-size Dependency

Figure 9.1: Security Benchmark Results

Figure 9.1a shows the overhead imposed by encryption and delegation using transport-level respectively message-level security (Secure Conversation and Secure Message) with a message size of one Grid Service Object.

Needless to say that the best performance is achieved without security. With only about 62 msec overhead, the best secure invocation was done with transport-level security (see TLS Encryption in Figure 9.1a). The two message-level security modes performed with overheads of up to 200 msec significantly worse (see Sec. Conv. and Sec. Message Encryption in Figure 9.1a). Due to a possible context reuse Secure Conversation can perform better when invoking multiple consecutive operations. But, this scenario was not in the scope of this evaluation.

Another cost-intensive operation is the delegation of credentials. Delegation is only supported in the Secure Conversation mode. The delegation process involves, in addition to several roundtrips, the creation of a new private/public key pair, which makes it a very performance intensive operation.

Figure 9.1b shows the impact of the message size on the performance. Obviously, the response times correlate to the message size. Especially large messages with message-level protection lead to unacceptable response times of over 6 s. Further, the processing of large messages (>100 Grid Service Objects) usually causes a close to 100 % CPU saturation. Clearly, this is a limitation of the current Web service and XML processing stack of Globus.

Obviously, especially secure Web service operations are associated with some overhead. In the following section, the impact on these results on different real services, such as the Globus

GRAM and the Migol services is investigated.

9.1.2 Migol Services

In this section the response times of the Job Broker Service (JBS), Migration Service (MS), GRAM2 and GRAM4 are compared. To ensure a better comparability the fork job manager is used for all submission tests, i. e. no queue time is measured. The resource manager selective flooding scheduler with two sites has been used for all JBS/MS tests.

Figure 9.2 presents the results of these experiments. Surprisingly, the GRAM4 client performed only slightly slower than the GRAM2 client (see GRAM2, GRAM4 (C) in Figure 9.2). In contrast to the GRAM2, the GRAM4 does not require delegation. If delegation is disabled, the GRAM4 C client even performs better than the legacy GRAM implementation. The performance of the GRAM4 has significantly improved compared to the GT3 GRAM (not depicted, for details see [212]). GT4 especially benefits from the availability of a C framework: the C client performs, with about 6 sec submission time, 10 sec faster than the Java client (see GRAM4 (Java) in Figure 9.2). A reason for this behavior is the necessary startup time of the Java Virtual Machine.

Furthermore, the performance of Migol's Job Broker Service and Migration Service has been measured (see JBS and MS in Figure 9.2). The overhead of the JBS compared to the GRAM4 (Java) amounts to about 18 sec, which is comparable to the performance of GT3 Migol implementation. A reason for the longer execution time are the necessary interactions with other services. For example, several service calls must be conducted to obtain information from the WS MDS and the AIS. Further, the application is submitted sequentially to two sites, which requires in average 9 sec per site using the Java interface to the GRAM4.

The performance of the MS was determined by measuring the time for stopping and restarting the service. File staging was disabled for this test to determine the overhead for file staging more precisely. The checkpoint file was provided via a NFS share. Since the MS uses the JBS for job submission, the JBS overhead also applies to the MS. In addition, several service calls are necessary, e. g. to authorize users, to perform the delegation-on-demand process with the AIS, to trigger checkpointing, and to update the corresponding Grid Service Object at the AIS. Especially the delegation-on-demand operation is very performance intensive.

After the initial performance evaluation we optimized the Migol services. The Web Service calls were reduced by consolidating operations, e. g. the create resource and start operations of the JBS and MS. Since delegation is a very expensive operation, we minimized the necessary effort by utilizing the GT4 Delegation Service [140]. This service enables the reuse of delegated credentials from any service within the same container. After delegation of the user credential to the Delegation Service, an endpoint reference to the credential is returned to the JBS respectively the MS client. With the credential reuse, the overhead for job submission and recovery was significantly reduced. The performance of the JBS and MS was improved by 6 sec respectively 13 sec (see JBS and MS (optimized) in Figure 9.2).

Another important infrastructure service is the AIS. In particular, the performance of the replication routines is evaluated in the next section.

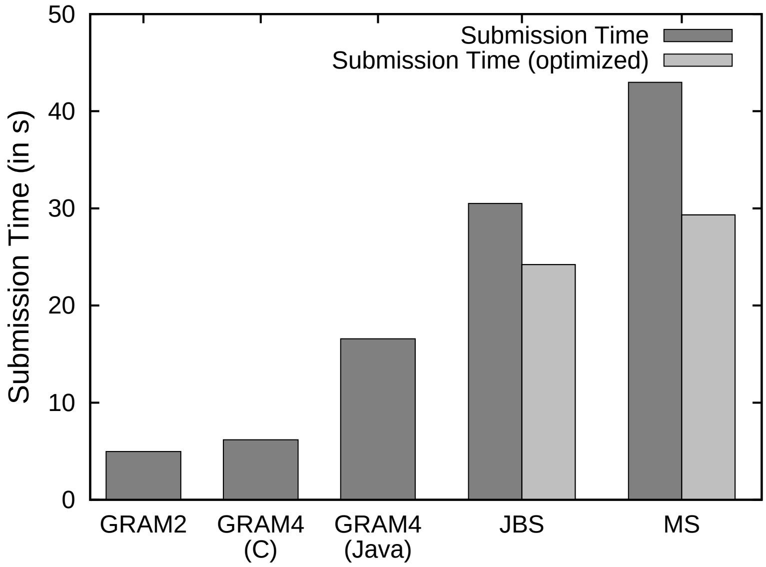


Figure 9.2: Migration and Job Broker Service Overhead

9.2 Replication Experiments

A major concern for the AIS is the performance associated with the replication of the AIS across the Grid. To demonstrate the robustness of the AIS in a real Grid scenario with high latencies, several experiments in the German D-Grid [95] and the Amazon EC^2 Grid have been performed.

In these experiments the performance of the two different group communication plugins, the Web service-based WS RRP and JGroups, is evaluated. For the study a micro application benchmark, which sequentially conducts a service registration using a GSO with a size of 3.1 kb at all active AIS nodes, is used. Each registration involves the synchronous replication of the registration data to all AIS instances, i. e. the service call does not return until the replication has been finished. The response time of each registration was measured. Every experiment was repeated 50 times. The following factors were considered:

- Number of AIS nodes [2, 4, 8]
- Network connectivity [Cluster, Grid]
- Number of concurrent users [2, 4, 8, 16]

All Grid nodes are running Java 5 on a Linux 2.6 kernel. The Globus Toolkit 4.0 container was deployed in a Tomcat 5.5 server. Further, the JGroups 2.5 Sequencer stack has been used.

As cluster setup the Einstein cluster of the Potsdam University was used. For the Grid scenario we utilized machines of the German D-Grid and the Amazon EC^2 Grid. Table 9.1 summarizes the latencies and bandwidths initially measured in the Grid. The D-Grid nodes

Site	Roundtrip Time	Bandwidth
edison.babylon (D-Grid)	0.24 msec	94.0 Mbit/s
buran.aei.mpg.de (D-Grid)	3.2 msec	77.5 Mbit/s
c3grid.pik-potsdam.de (D-Grid)	1.45 msec	53.4 Mbit/s
Amazon EC^2	103 msec	2.58 Mbit/s

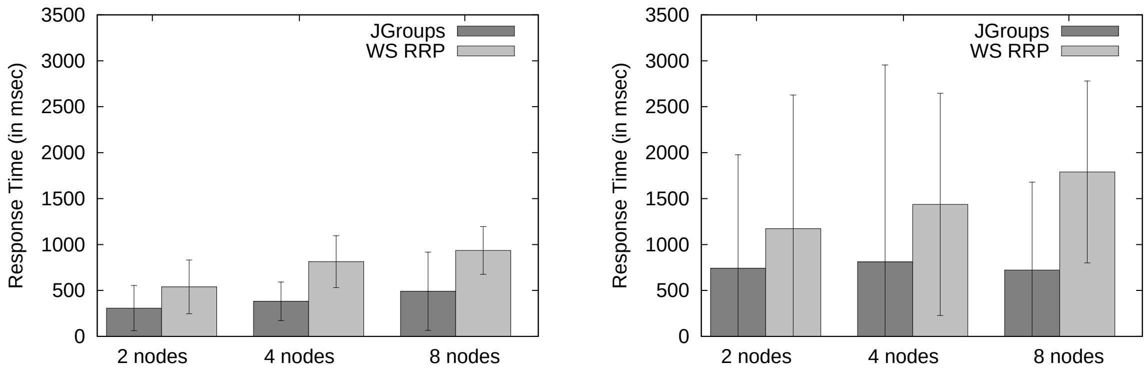
Table 9.1: Latencies and Bandwidths measured from the University of Potsdam

are situated in Germany and are connected with high bandwidth and latencies. The EC^2 site is located in Seattle, USA. Especially concerning for an active replication are the high roundtrip times between the D-Grid and EC^2. All Grid tests were performed with $\frac{n}{2}$ nodes in the D-Grid and $\frac{n}{2}$ nodes in the EC^2 Grid.

9.2.1 AIS Response Times

In the initial setup the response times of the new WS RRP protocol and the sequencer-based JGroups protocol stack were compared. Both plugins differ in the chosen replication strategy, but provide about the same consistency guarantee. The experiments were conducted using the security mechanisms described before: WS RRP was configured with GSI authentication and encryption in transport-level security mode, while JGroups was setup with its group communication friendly security schema using certificate-based `AUTH` and `ENCRYPT` micro-protocol with dynamic key generation.

In the first scenario, the AIS response time within the Einstein cluster was measured (see Figure 9.3a). JGroups showed response times of 308-491 msec with a standard deviation of up to 427 msec. As expected WS RRP performed worse with response times of 539-936 msec and a standard deviation of up to 292 msec.



a) Einstein Cluster

b) D-Grid/EC^2

Figure 9.3: WS RRP versus JGroups Sequencer: Response Times

A reason for this behavior is the overhead caused by the Web service stack and the GSI security features of WS RRP. WS RRP uses SSL for all messages. This provides a higher level of security, but is also associated with a performance penalty: for every point-to-point connection a session key must be negotiated. JGroups uses one shared cluster session instead of point to point SSL sessions. It is therefore only necessary to generate and exchange a single key, which can be used for multiple messages.

As expected, the replication across the Grid leads to an increase of the response time (see Figure 9.3b): JGroups increased by 65 %, WS RRP by 91 %. This effect is mainly caused by the high network latency (cmp. table 9.1). The standard deviation of WS RRP climbed to 1454 msec – JGroups showed an even higher standard deviation of 2144 msec. As explained, the WS RRP overhead is caused mainly by the SOAP stack.

9.2.2 AIS Loadtest

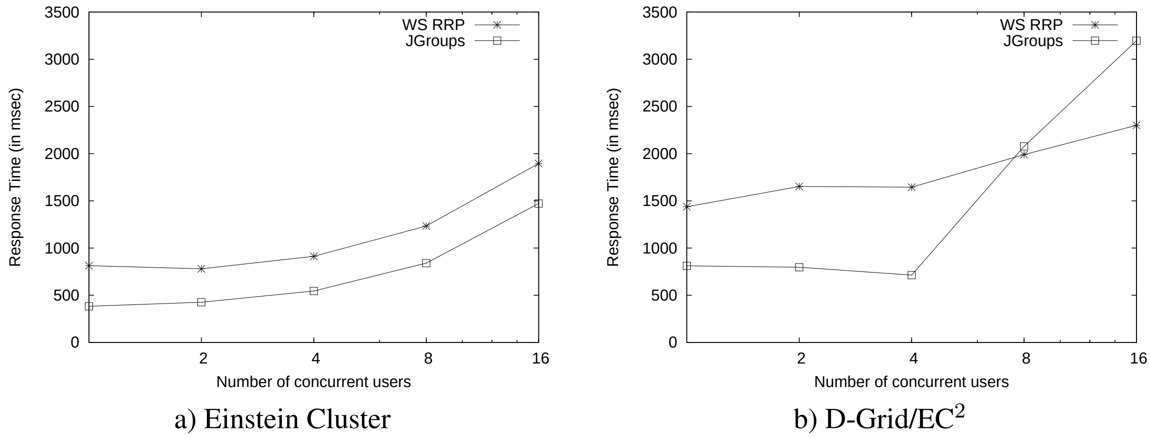


Figure 9.4: WS RRP versus JGroups Sequencer: Load Test with 4 Nodes

To evaluate the robustness of the AIS, a stress test was conducted. During the test a high user load was put on a four node AIS configuration. Each concurrent user performs a service registration sequentially at each of the four AIS nodes. Figure 9.4 shows the results.

In a cluster setup WS RRP and the JGroups Sequencer protocol scale equally well. Figure 9.4a) shows that the response times of WS RRP and JGroups increase proportionally to the number of concurrent clients. The Grid scenario diagramed in Figure 9.4b) shows a quite different picture. While the response times of the Sequencer implementation rapidly climb with eight and more concurrent clients, the response times of WS RRP only increase moderately despite the high security overhead of GSI observed in earlier experiments. WS RRP respectively Totem requires the exchange of less messages than a central Sequencer-based protocol. Especially in a Grid each message is subject to high latencies. Also, the central Sequencer process represents a natural bottleneck under high loads. Another aspect is the noticeable high standard deviation of the JGroups stack under load: some requests showed peak response times of 120 sec. Therefore, the standard deviation reached up to 34 sec with 16 concurrent clients.

Having analysed the performance of the Migol services, in the following section the overhead imposed on an Grid application by Migol is evaluated.

9.3 Monitoring and Migration Experiments

In the following, the REMD-Manager presented in section 4.3.3 is used as example application to study the overhead of Migol. The REMD experiments have been conducted on the LONI Grid [10]. The REMD-Manager is used to deploy tasks to the LONI clusters: QueenBee (QB), Poseidon and Eric.

Initially the runtime of a single replica run is effected by Migol's active monitoring mechanism and the required checkpoint registration. For this purpose, a medium-size NAMD job was started with and without Migol support. Monitoring intervals between 20 s and 2 minutes were chosen to study the effect of monitoring frequency on runtimes. The update interval for checkpoint metadata was set to five minutes. Since the time measured for a checkpoint update operation was on average only 1.3 seconds, this operation is not expected to be a critical factor.

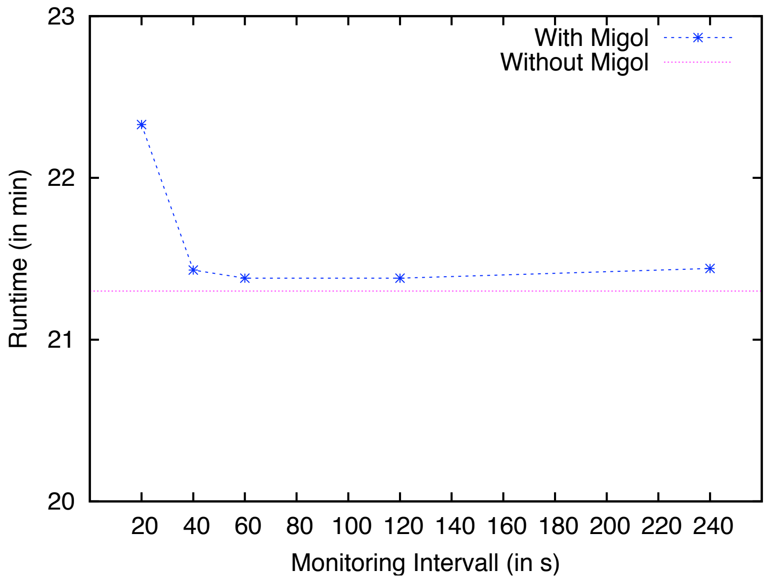


Figure 9.5: Migol Monitoring overhead

Figure 9.5 illustrates the results of this experiment. The runtime of the NAMD job on QueenBee without CPR/Migol amounted to 21.3 minutes. With Migol at worst a 1 minute overhead was observable with a monitoring interval of 20 s. With a lower monitoring interval, the overhead was further reduced, e.g. the runtime overhead with a 2 minute monitoring interval is only 10 s, which is only slightly higher than the variance of typical NAMD runtimes.

Having analysed the runtime of a single NAMD job, a more complex REMD using the REMD-Manager was evaluated. The REMD-Manager was configured to run a simulation

Number of NAMD steps	100
Number of MPI processes per NAMD run	16
Required staging files/-size	6 files 10 MB
Number of replica processes	2-16
Total number of replica-exchange steps	16

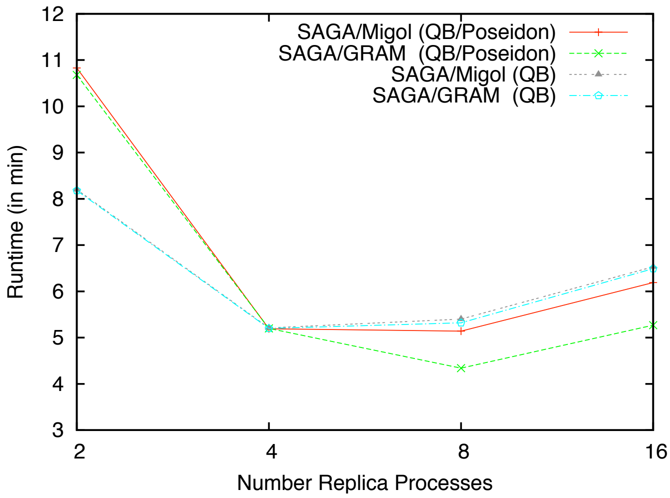


Figure 9.6: REMD Application Characteristics and Runtimes

with 2 to 16 replica processes and 16 replica-exchange steps. To stress test the Migol infrastructure very short NAMD tasks with only 100 steps have been used. The runtime of a REMD simulation depends to a great extend on the queuing time at the local resource management system. Thus, the queueing times during the experiment have been minimized. However, as the results show, small queueing delays could not always be avoided.

Figure 9.6 shows the REMD configuration used as well as the runtimes measured within the LONI Grid. Since the total number of replica exchange steps remained constant, the runtime decreases the more replica processes are used. With more than four replica processes a slight decrease of the efficiency can be observed. The more replica processes, the more dominant the sequential overhead at the REMD-Manager becomes. To emulate the most general case, where each exchange step requires the staging of different files, in the setup, six files with the total size of about 10 MB had to be staged. This transfer required approximately 5-10 s on the LONI network. Due to the small problem set computed by each replica (only 100 NAMD steps, which require 35 seconds computation time on QB), this bottleneck becomes very evident. However, in more realistic scenarios with larger problem chunks this issue does not exist.

On average, with all factors considered, the SAGA/Migol adaptor added a total runtime overhead of about 15 seconds to the time-to-completion. It is important to note that this does not change significantly with neither the number of replicas, the number of replica-exchanges, nor the runtime of each replica. These results indicate that the SAGA/Migol overhead is acceptable.

Figure 9.6 also shows that the REMD-Manager can be employed to orchestrate multiple resources concurrently, as well as different resources (QB/Poseidon) individually. During the coupled distributed run, one quarter of the processes were allocated to the smaller machine Poseidon, while the bulk stayed on QueenBee. As the number of replicas gets larger, the concurrent distributed runs have a lower time-to-completion than when QueenBee was used in isolation. However it is important to note, that in particular Poseidon showed long queuing times leading to high variance in the overall time-to-completion. This overhead is significant for short-running tasks and less important for longer running tasks.

Since the probability of a failure during a 10 minute run on a few resources is rather low, the reliability of the proposed framework was validated by introducing faults into the systems. For this purpose, selected replica processes have been kill. During this experiment the time required by Migol to restart the task was measured. Due to the selected monitoring interval of one minute and failure threshold of 2 tries, the failure detection time averages to 2.5 minutes.

The recovery time required for the restart of the job is ~ 42 seconds. This corresponds to the earlier results presented in Figure 9.2 As explained before, this overhead is mainly caused by the complex interactions conducted by the Migol backend. The monitoring service initializes the restart at the JBS. A major performance penalty is the delegation-on-demand mechanism required to obtain the credential of the user from the AIS – this procedure demands the creation of a public-private key pair, which is very costly. Further, the resource discovery and selection mechanisms used by Migol's JBS are designed with a focus on long-running applications, and currently show some substantial overhead, especially when used for short-running tasks.

9.4 Summary and Discussion

While these results indicate that the usage of Migol is associated with some overhead, we believe that this is acceptable compared to the benefits a fault-tolerant, self-healing infrastructure offers. In addition, it must be noted that further simple yet effective optimizations are possible. For example, by introducing an efficient caching mechanism for resource information at the JBS expensive discovery queries at the MDS can be minimized or done as background task. Further, the job submission process of the JBS can be optimized – in particular the usage of the ARS incurs some overhead compared to a plain GRAM4 submission.

Further, it can be concluded that the overhead incurred by Migol on the runtime of Grid applications is hardly measurable. Using a relative small monitoring intervall of one minute, REMD tasks incurred an overhead of about 10 s, which was less than 1 % of the total runtime. This overhead is well acceptable taking into account the benefits of Migol's application-level failure detector.

"No technology can make everyone happy."

Jim Gray

10 Related Work

Fault tolerance within distributed systems has been extensively researched. Various systems address reliability similarly to Migol. This section aims to give a survey about these approaches, while highlighting the benefits of Migol at the same time.

This chapter is structured as follows: Part 10.1 focuses on fault tolerance frameworks that can be utilized by Grid applications to enhance their reliability. In particular, this includes checkpointing, automatic failure detection and recovery techniques. Section 10.2 surveys different approaches for ensuring the availability of critical services. This includes a survey of different group communication systems and replication frameworks as well as utilities for building highly available cluster systems.

10.1 Application Fault Tolerance

As pointed out in this thesis, fault tolerance is in particular for long-running application a major issue. After an overview of checkpointing techniques in section 10.1.1, section 10.1.2 describes different application management frameworks that address the reliability of Grid and cluster applications. Since most of these systems rely on checkpoints, the availability of these checkpoints must be ensured; section 10.1.3 will survey different systems for checkpoint replication.

10.1.1 Checkpointing

Backward recovery based on checkpoints is one of the most used fault tolerance mechanisms. Checkpointing has been extensively researched in the context of local and cluster systems. Condor [210], libckpt [259], the Berkley Lab Checkpoint Restart (BLCR) [57] and PBeam [256] provide more or less transparent checkpointing of operating system processes for different Unix systems. However, these approaches are all restricted to checkpointing of single processes.

A challenge for distributed applications with communicating processes is the gathering of a consistent application state. Several checkpointing and logging protocols have been proposed (for a survey refer to Elnozahy et. al. [113, 103]). Distributed checkpointing has also been extensively researched in the context of the Message Passing Interface (MPI) [231] standard. For a survey of checkpointing and rollback implementations for MPI refer to [113, 49]. CoCheck [297] and LAM/MPI [278] e. g. support the coordinated checkpointing of MPI applications. While CoCheck reuses the Condor checkpoint and restart system, LAM/MPI relies on BLCR for the gathering of local process states. OpenMPI [169] extends the checkpoint mechanisms of LAM/MPI. The checkpoint and restart module of OpenMPI provides an abstraction for different checkpointing protocols. The current implementation supports BLCR-based and application-level based checkpointing. MPICH-V [60] supports coordinated checkpointing as well as uncoordinated checkpointing in conjunction with message logging.

Unfortunately, all checkpoint implementations for MPI require the installation of a custom MPI distribution – some also depend on certain operating system libraries, which restricts their usage in a Grid. The landscape of a Grid consists of a variety of MPI implementations - a special MPI implementation installed on all systems cannot be assumed.

Checkpointing is also heavily used within Grid systems to support the recovery of applications. While some Grid frameworks, such as GRADS [315]), rely on user-level checkpointing most Grid frameworks, e. g. Cactus [142] and InteGrade [101], utilize application-level checkpointing to support a recovery among heterogeneous resources. Various tools for supporting the creation of application-level checkpoints exists: For example, Bronevetsky et. al. [63] proposed a compiler-based framework for instrumenting MPI application to perform application-level state-saving. Cactus supports almost transparent application-level checkpointing through different IO thorns.

While the described systems mainly focus on the state saving aspects, advance features, such as the management of checkpoints, failure detection and automatic recovery are not addressed. In particular, these issues must be addressed to provide a comprehensive fault tolerance solution for Grid applications. In the following, an overview of such frameworks is given.

10.1.2 Fault Tolerant Grid Middleware Platforms

Process recovery and migration has been extensively studied in the area of cluster computing (for a survey see Milojičić et. al. [225]). In the following, a survey of different reliable Grid and cluster frameworks is given.

Globus Heartbeat Monitor

The Globus Heartbeat Monitor (HBM) [296] implements the unreliable fault detector pattern [70] for fail-stop failures. Application can register themselves together with the local monitor to the HBM service using the client registration API. The local monitor periodically reports the status of the application to the data collector of the HBM. The data collector then updates its state information and generates callbacks for missing heartbeats to previously registered application-specific consumers. The callback consumer can then implement a defined fault handling strategy, e. g. the re-scheduling of the application. However, a lot of manual effort on application-level, such as the selection of a new resource, the conduction of the file staging etc., is required to facilitate a restart after a failure. Further, the HBM service was part of the Globus Toolkit 2, but is not supported with GT4 anymore.

Jin/Zou [178] proposed a fault tolerant Grid architecture, the Fault Tolerant Grid Platform (FTGP), that extends the original HBM service. The framework extends the GT2 HBM by the possibility of fault recovery. Services and applications can be automatically restarted in case of a failure. The fault tolerance of the infrastructure is considered: The framework e. g. proposes a primary-backup replication schema for the MDS. Further, results can be redirected to the MDS in case of a resource manager failure. However, FTGP does not address state persistence or checkpointing at the moment. Further, the entire framework is based on the deprecated HBM service.

Condor

The Condor/PGRADE system [194] consists of a checkpointing mechanism for PVM applications, which is used in conjunction with Condor-G [132] for the management of the application run. PGRADE relies on user-level checkpointing, which limits its usage to homogeneous resources. The fault detection mechanism of Conder-G [132] simply relies on polling the job state at the LRMS. This allows the detection of some errors; however, an application-level failure detector as used by Migol can detect much more complex errors. In case a failure is detected, the failed job is simply re-executed. The framework does not address the fault tolerance of the service infrastructure itself.

EasyGrid

With EasyGrid, Silva/Rebello [96] propose a hierarchical infrastructure for implementing fault tolerant master-worker applications. The framework utilizes various workarounds to enable a fault tolerant execution of MPI programs. For example, to avoid the abortion of a MPI program due to sending a message to a failed node, EasyGrid sends a non-MPI keep-alive message to the remote process before every `MPI_Send`. The EasyGrid infrastructure can handle certain infrastructure faults. However, the Global Manager, a core component of the system, represents a single point of failure. While EasyGrid addresses similar shortcomings of current MPI implementations (as described in section 5), it is restricted to single cluster machines running LAM/MPI. Further, the framework is only suitable for loosely coupled master-worker applications.

HPC4U

The HPC4U [165] middleware also supports the checkpoint based migration of MPI applications. However, the framework can solely be used in conjunction with the resource management system Computing Center System (CCS) [185]. In contrast to other LRMSs, which use a queue-based scheduling approach, CCS is a pure plan-based systems. However, the restriction on a single LRMS makes the system less suitable for Grids. Also, HPC4U relies on kernel-level checkpointing, which limits the use of the framework with heterogenous resources as found in a Grid.

Cactus

The migration of Cactus applications [200, 201] in the Grid was demonstrated by Lanfermann et. al. in early experiments in GT2-based Grids. The results of this work provided useful insight for the design of the Migol Grid middleware. In contrast to this early work, Migol emphasizes OGSA compliance as well as the fault tolerance of the infrastructure services. Further, Migol is not limited to Cactus applications and supports much more applications, e. g. MPI or SAGA applications.

The BBH factory [283] is another Cactus-based application management framework. The main focus of the BBH factory is the management of cluster configurations required for building and running Cactus applications. However, it also provides rudimentary support for the recovery of Cactus applications. If a recovery is requested, the BBH factory automatically creates a new directory structure containing new versions of all output and checkpoint files before it resubmits the Cactus application. Compared to Migol, BBH factory lacks certain features, such as automatic resource discovery, application monitoring and automatic recovery. However, BBH factory provides useful functionalities that can easily be deployed in conjunction with Migol.

Condor-G, Nimrod-G and Legion

While the described frameworks mainly focus on long-running MPI applications, several frameworks for loosely coupled applications have been developed. Condor-G [132], Nimrod-G [67], and Legion [74] are some examples for such frameworks. Basic fault tolerance support is provided by automatic re-scheduling of failed tasks. Advanced features such as the management of checkpoints however, are not supported. Further, these frameworks rely on a very simple failure detection mechanism – usually by simply polling the job state at the Globus gatekeeper. This allows the detection of some errors, but application-level failure detectors as used by the Migol library can detect much more complex errors. Also, these frameworks do not follow an overall system-wide approach to fault tolerance as Migol and do not consider the fault tolerance of the infrastructure services.

Another useful feature for fault tolerant Grid applications is the ability to automatically replica checkpoint files. In the following section different frameworks addressing this problem are described.

10.1.3 Checkpoint Replication

Data replication has been extensively researched in the context of Grids. Current Grid middleware platforms, such as the Globus Toolkit 4 [123] provide base services for managing replicas: The Globus Replica Location Service (RLS) [76] is a simple information service, which maintains information about physical locations of file copies (replicas) using a unique logical filename. RLS instances can be hierarchically aggregated: Local Replica Catalogs (LRCs) store replica information for a local site, while Replica Location Indices (RLIs) consolidate data of a set of LRCs. While this setup provides some fault tolerance, it lacks strong consistency guarantees since different RLIs are not required to have the same consistent view of the current Grid state. Further, the data model of the RLS currently does not support all required aspects: for example, it is not possible to store the state of a replica in the RLS. Especially in a Grid it can happen that certain physical files become unavailable - thus, storing the state of a replica is a critical feature.

The Globus Data Replication Service (DRS) [141] offers a higher level Grid service for automatic replica creation. The DRS automatically conducts file transfers to a specified resource and creates the respective RLS entries. But, the service lacks autonomic capabilities, such as the automatic selection of a destination Grid resource or the detection of a new checkpoint file to replicate.

Colesa et al. [86] propose an automatic and reliable data management framework on top of the Globus Toolkit. Since the framework is based on the RLS and DRS, the same disadvantages apply. While the framework provides some autonomic features, such as a new file detection daemon, it still misses features, such as the capability to automatically select a replica target site based on the available bandwidth. Further, the framework aims to offer a reliable replica service, but does not address the fault tolerance of the infrastructure itself.

Having described the features of different fault tolerant frameworks from the application point of view, the following section focuses on the provisioning of highly available infrastructure services.

10.2 Highly Available Services

This section reviews related work with respect to supporting the high availability of infrastructure services. Service replication is a fundamental principle to achieve high availability. As described, group communication is a common technique used to replicate services. Section 10.2.1 introduces different group communication frameworks. After this overview, the usage of replication technologies within CORBA (section 10.2.2) and in Grid/Web service frameworks (section 10.2.3) is discussed. The remainder of the section focuses on highly available clusters. While section 10.2.4 gives a general overview of cluster fault tolerance, section 10.2.5 introduces RSerPool, a protocol suite for high available clusters.

10.2.1 Group Communication Frameworks

Various systems that address group communication within clusters have been proposed. Several frameworks have been developed at Cornell University, such as Isis [53], Horus [316] and Ensemble [160]. Isis was the first group communication system, which introduced the process group abstraction. However, the architecture of Isis was very monolithic and inflexible. Horus introduced more flexibility by providing a stack consisting of micro-protocols. In this stack each micro-protocol handles a small set of guarantees. Ensemble is a successor of Horus, which was written in Objective CAML. A special emphasis of the Ensemble project is the formal verification of certain protocol properties, such as virtual synchrony [52].

A Horus/Ensemble inspired micro-protocol architecture is also used by different other frameworks, such as JGroups [41] and Appia [226]. Another popular group communication toolkit is Spread [24]. Spread was as the Totem protocol developed by Amir and uses a similar token protocol for replication of messages.

Migol's Replication Service uses JGroups as one group communication framework for the replication of the AIS (see section 7.3.2). JGroups was originally based on Ensemble, but has been rewritten to plain Java. The framework is heavily used within distributed middleware systems. For example, JBoss uses JGroups for replication of parts of JEE applications, e. g. HTTP sessions or stateful session beans. The main reasons for using JGroups are:

- JGroups is widely used in production infrastructures, e. g. as part of the JBoss application server.
- The framework is written in Java and can be seamlessly integrated in the Globus Web service framework, which Migol is based on.
- If required, the micro-protocol stack can be extended to meet application requirements.

However, it must be noted that Migol's Replication Service (RS) is not limited to JGroups. Further, group communication frameworks can be simply integrated via a plugin API if a particular requirement arises.

10.2.2 Replication within CORBA

The *Common Object Request Broker Architecture (CORBA)* [242] is a well-known specification for communication in object-oriented distributed systems. The core of CORBA is the Object Request Broker (ORB), which is the runtime system responsible for the transparent invocation of objects. Different group communication systems have been integrated with CORBA. For example, Electra [217] uses the Horus framework for transparent replication of CORBA objects.

Further, CORBA provides a standardized Object Group Service (OGS) as building block for fault tolerant applications. The OGS provides an abstraction for the invocation of methods on a set of CORBA objects. For object groups CORBA introduces so called interoperable object group references (IOGR). The fault tolerant CORBA specification considers a primary-backup, an active as well as a voting style replication mode. The CORBA standard demands strong consistency, i. e. the runtime system must ensure that all replicas have the same state all the time. Implementors of the specification are free to choose an appropriate method, e. g. group communication, which ensures the specified semantic. An implementation of the fault tolerant CORBA standard has been provided e. g. in the Eternal system [235].

However, the CORBA specification has some limitations: For example, the standard does not address heterogeneity or network partitionings [117]. Also it must be considered that current Grids are build on top of Web services a successor technology of CORBA – unfortunately, a standardized replication service has not been proposed for Web services yet.

10.2.3 Web and Grid Service Replication

Following the Web service architecture, the Universal Description, Discovery and Integration service (UDDI) [314] represents the central entry point to a service-oriented environment. Hence, ensuring the availability of UDDI services is critical. To achieve fault tolerance, the UDDI specification defines a simple replication process protocol [85]. Unfortunately, the replication model has a major disadvantage: it relies on a lazy update propagation schema, i. e. replica updates are propagated asynchronously to other nodes. To avoid inconsistencies, changes can only be performed at the node, which the service originally registered to, the so called operator node. If this host becomes unreachable, no changes to the service are possible. Thus, the operator node represents a single point of failure.

Zhang et. al. [338] investigated the feasibility of primary-backup replication for Globus Toolkit 3 Grid Services. The proposed architecture utilizes an eager update propagation schema, i. e. updates are transferred to all backups before a client request returns. In contrast to active replication, the client is only allowed to send its request to the primary component. Thus, the primary component is the natural bottleneck – depending on the number of backups and concurrent users it may take some time to forward the updates. Further, no load balancing is possible. Update propagation is done using the Globus Toolkit 3 notification framework, which results in a significant performance penalty.[1] Thus, the framework has been re-developed using a socket-based protocol in conjunction with Paxos [199, 337]. However, the framework still lacks important features, such as the authentication of processes and message protection.

Different frameworks explore the usage of group communication in the context of Web services. WS-Replication [274] supports the active replication of Web services in a wide area network. The framework consists of a replication component, which implements the replication state machine on the respective node, and a multicast component, which is responsible for group communication. WS-Replication builds upon the JGroups toolkit and extends it with a SOAP micro-protocol. Osrael et. al. provide a similar framework for replication of Axis2 based Web services [246]. The framework extends the Axis2 SOAP engine by a special handler, which transparently replicates all inputs using group communication. For this purpose, the Spread toolkit is reused. However, both frameworks are currently not available for download. Further, they neither support Web Service Resource Framework (WSRF) conform Web

[1] A similar worse performance of the GT3 Grid service stack has been observed in Luckow/Schnor [213].

services as provided by GT4 nor do they address security related aspects.

10.2.4 High-Available Compute Clusters

Highly available cluster solutions have been well researched. In general, a HA cluster combines a set of technologies:

- Replication of critical cluster services (e. g. using a primary-backup schema)
- Fault tolerant storage (e. g. using RAID [253])
- Fault tolerant network setup (e. g. by using multiple paths)

Failover transparency is, in general, ensured using an IP address takeover. For state sharing often a shared, fault tolerant storage media is used.

HA-Linux [269] provides a collection of tools for building highly available Linux clusters. The framework can be used to build active/active as well as active/passive clusters. The core of HA-Linux is *heartbeat*, which is used to monitor configured cluster nodes and services. The *fake* tool is used to take-over the IP address in case of a failure. The transfer of the application state is in general accomplished using a shared storage. Similar cluster solutions exist for commercial operating systems: Windows 2003 Server provides integrated clustering features [14]. Open High Availability Cluster is an open source HA cluster framework for Solaris [12].

A solution that particularly addresses HPC clusters is HA-OSCAR [155]. HA-OSCAR provides an integrated bundle of cluster software that can easily be configured to operate within a HA cluster setting. The setup of HA-OSCAR is based on the primary-backup schema. The system is able to automatically checkpoint different services, e. g. it supports the state saving of LRMS queues. Between all HA servers a heartbeat mechanism is applied for failure detection. In case the primary server fails the backup server takes over all services. HA-OSCAR is currently limited to the GT2 services: MDS, GridFTP and the GRAM Gatekeeper. Further, the LRMS PBS and SGE are supported [207]. A drawback of HA-OSCAR is the restriction to the OSCAR [245] and Rocks [249] Linux distribution.

The described clustering frameworks do not provide out-of-the-box support for Globus based services. However, since these services mainly rely on standard tools, such as a Tomcat servlet container and/or a MySQL database, the inclusion of these services in a cluster setup is in general straightforward.

A HA cluster setup can be considered complementary to Migol. Ensuring the availability of critical cluster services, such as the WS GRAM and the LRMS is an important building block to avoid expensive job recoveries on Grid level. In general, the earlier a fault is handled, the lower is the impact of a fault.

Having described various existing and proven HA cluster solutions, the RSerPool standards aims to provide a comprehensive framework for building highly available clusters based on the SCTP transport protocol.

10.2.5 RSerPool

The *RSerPool* [313] framework aims to provide a standardized protocol suite for building highly available, clustered services. Using the RSerPool notation, servers that offer the same service are grouped together into a server pool. Each server is referred to as pool element. The core of the RSerPool architecture are the pool registrar (PR) servers, which provide a central resolution service for mapping pool handles to transport addresses. This is an essential step for accessing a service. Clients are referred to as pool users.

Pool elements (PE) can register themselves at a PR using the Aggregate Server Access Protocol (ASAP) [262]. Failover between pool elements is supported via a client based synchronization protocol, i. e. no synchronization between elements of a service pool is conducted. While this architecture is conform with the end-to-end principle [275], it has several disadvantages. In addition to its lack of transparency, a possible failure of the client is not considered in the architecture.

Due to their central role, pool registrars are clustered using the Endpoint Handlespace Redundancy Protocol (ENRP) [335]. ENRP is based on the primary-backup schema. Each pool element is associated with a home ENPR server. Registration and updates are done at the home PR server. The ENPR server ensures that the update is propagated to all other ENPR servers (using the `ENRP_HANDLE_UPDATE` message). Failure detection is done using a heartbeat protocol. In case the failure of a server is detected, the server, which detected the error, initiates a take-over. To conduct the take-over the ENRP server must acquire a majority of votes from all other ENRP servers. During the take-over process the new ENRP claims the ownership of all pool elements maintained by the failed ENRP server. Then, all clients of the old ENRP are notified about their new home ENRP server.

RSerPool heavily utilizes the Stream Control Transmission Protocol (SCTP) [298] to provide fault tolerance against network failures, e. g. by using redundant paths. While SCTP provides interesting reliability features, it limits the usage of such a RSerPool. For example, commonly Grid services do not support SCTP as transport-level protocol.

Migol and RSerPool address different scenarios, while Migol aims to support the automatic recovery of long-running Grid applications, RSerPools aims to provide application-independent high availability. However, Migol and RSerPool share different architecture elements: The AIS provides similar to the RSerPool PR a central registry for locating services. In contrast to RSerPool, Migol provides an extensible architecture. Per default, active replication is used to ensure the availability of the AIS. Especially in a loosely coupled environment, active replication has some advantages: Changes can be done at any service instance, e. g. an instance that is close to the current location of the application. Further, recovery times are in general less than in primary-backup replication schemes. The fail-over mechanism of RSerPool heavily relies on client-side state saving – for data intensive application as addressed by Migol this is not an option. For infrastructure services, Migol relies on server-side state saving, which is more transparent and performs very well especially in case of transient failures.

10.3 Summary and Discusion

Several fault tolerant frameworks for Grids and clusters have been proposed. However, usually these frameworks only address single aspects, such as checkpointing within MPI applications, in particular well. Many of these systems claim to provide fault tolerance support, e. g. by supporting the monitoring and re-execution of Grid jobs. But, these frameworks usually only offer quite simple failure detection mechanisms, such as polling the job state. Migol's application-level monitoring approach is able to detect much more complicated errors. Further, these systems often introduce single point of failures in their infrastructure, e. g. by not considering a possible failure of the resource broker or the information service. Thus, it is of major importance to address Grid reliability from a overall system-wide point of view. Similarly, this has been concluded by Dabrowski [97].

Migol aims to fill this gap by addressing application and infrastructure fault tolerance at the same time. The architecture carefully incorporates fault tolerance techniques to meet the reliability requirements of every services. Critical and central service, such as the AIS are designed with respect to high availability. Different group communication systems for supporting the replication of applications exist. As extensively elaborated, JGroups represents the most suitable framework for the replication of Java-based Web services.

However, it must be noted that Migol can be used in conjunction with many of the proposed frameworks. For example, it is possible to combine Migol with the Cactus checkpointing thorn or a fault tolerant MPI implementation. Further, Migol can benefit from HA cluster frameworks, which can significantly reduce the downtime of Grid resources, leading the fewer necessary recoveries.

"If it's important, how can you say it's impossible if you don't try?"
Jean Monnet

11 Conclusion and Future Work

Developing Grid applications is a complex and difficult task for various reasons. Distributed computing environments are inherently unreliable and prone to failures. In particular for long-running applications, reliability is a major concern. Being able to deal with failure as an intrinsic property is a key pre-requisite to efficiently utilize distributed infrastructures such as Grids. Section 11.1 summarizes the contributions of this thesis with respect to the vision of achieving a reliable Grid platform.

The proposed Migol framework is closely bound to the WSRF Web service standard and the Globus Toolkit. While the usage of Web services offers many benefits, there have been some drawbacks and controversies in particular related to limited interoperability and penetration of WSRF. Section 11.2 summarizes the experiences made with Globus/WSRF during the development of the Migol framework.

During the writing of such a thesis, different new topics directly and/or indirectly related to this thesis arise. These future research issues have been collected in section 11.3.

11.1 Achieved Results and Discussion

The objectives of this thesis have been achieved. Migol provides an integrated framework for addressing fault tolerance within distributed systems – from the application layer up to the Grid infrastructure. This thesis makes the following contributions:

- **Motivation and Requirements:** The need for more fault tolerance within Grid infrastructures and applications has been extensively motivated. Based on different complex use cases the requirements with respect to a reliable Grid infrastructure have been presented.

- **Self-Healing Infrastructure for the Automatic Recovery of Grid Applications:** Migol provides a generic, reliable infrastructure for managing Grid applications. The framework addresses the complete lifecycle of an application from the deployment and start to the monitoring of the application run. In case of a failure, applications are automatically recovered.

- **Design of a Reliable Grid Architecture:** While the Open Grid Service Architecture (OGSA) envisions the Grid as a highly reliable and scalable infrastructure, it does not provide any details about how to achieve this dependability goal. Migol addresses Grid reliability from an overall perspective including all components from the Grid applications itself to the service infrastructure. Using this approach several problematic architectural issues, such as single point of failures in the infrastructure, could be identified and resolved. Such issues are usually not apparent when solely considering individual services.

- **Intrinsic Infrastructure Fault Tolerance:** While the design identified several potential points of failures, every service and potential failure source was carefully assessed with respect to its criticality. Based on this evaluation, suitable fault tolerance techniques have been selected trading off the costs associated with fault tolerance, such as additional required resources, performance penalties, scalability, the probability of a failure, and the benefits of the enhanced reliability.

- **Service Replication:** A particular contribution with respect to infrastructure fault tolerance is the service replication framework developed for ensuring the availability of Migol's core service the AIS. The extensible framework supports the active replication of the AIS based on the newly developed WS RRP protocol and the JGroups framework.

- **Development:** An open implementation of the proposed architecture has been developed.

- **Evaluation:** The resulting system has been evaluated with several real life applications within different production Grid environments, such as the German D-Grid and the US LONI Grid.[1] Applications tested with Migol include AMIGA, Cactus and the molecular dynamics code NAMD. During this evaluation, the overhead of the reliable infrastructure on the application run-time, the restart and recovery costs as well as the re-configuration ability of the infrastructure have been assessed. Obviously, fault tolerance is associated with some overhead; however, the identified overhead is acceptable compared to the benefits of fault tolerance.

- **Fault Tolerant Design Pattern:** Several key principles for building a reliable infrastructure have been identified and collected. These patterns are naturally not restricted to Migol and can be applied to the design of other reliable distributed systems.

- **Standardization:** Migol supports the open Grid standards developed within the OGF. All services expose a WSRF compliant interface. Grid applications can access Migol via the SAGA CPR application programming interface.

[1] The LONI Grid is also part of the US TeraGrid.

Migol provides a proven infrastructure for the supporting the reliability of computational Grid applications. However, Migol is also subject to the theoretical limitations that exist in distributed systems. As most practical systems, Migol heavily relies on timeouts for failure detection. Timeouts are associated with several drawbacks: As pointed out in section 2.5.2 it is not possible to differentiate between the slow-down of a process and a crash, or between the failure of the network and the failure of the process. Hence, it is only possible to detect processor slowdowns within the bounds of the chosen timeout thresholds.

While there are theoretical models, which eventually guarantee the accurate detection of failures (see e. g. Chandra/Toueg [70]), the described algorithms are usually not suitable for a rapid failure detection [75]. For real systems sophisticated schemas to estimate good timeout values remain the only option for rapid failure detections. However, in general such mechanisms are always a trade-off between the failure detection time and accuracy. A short timeout decreases the failure detection time – at the same time the rate of mistakenly detected failures rises. Further, the network traffic, performance overhead caused by the failure detector and the scalability of the approach must be considered.

This thesis demonstrated that Migol is capable of handling crash faults of applications very well. That means that for most use cases, e. g. for the monitoring of Grid applications, this model is sufficient. At worst, an error is mistakenly detected, which potentially leads to an unnecessary recovery. However, except the needlessly allocated resources, the system is not negatively affected. But, in particular when dealing with high available services, such as the AIS, this limited failure model can have severe consequences. Our experiences showed that when deploying the AIS across different Grid sites very arbitrary failures may occur. For example, the data of a single AIS instance can be corrupted by a defect RAID controller. This kind of error does not lead to an immediate failure; however it can cause a Byzantine type of failure later on. A single corrupt AIS instance can block the system for some time e. g. by replying respectively not replying to joins. Voting protocols, e. g. Paxos [199], can deal with such Byzantine failures. In the future, such a protocol should be further evaluated on its suitability for Grids.

Another issue is that must be further addressed in the future is scalability. While the experiments showed that Migol is sufficiently scalable for current Grids, a further grows of infrastructure sizes can be expected. With the increase in the Grid sizes and number of applications, measurements to further scale Migol must be taken. However, there is sufficient potential to improve the scalability. For example, it is easily possible to partition the monitoring traffic across the already redundantly deployed MRS instances.

As described, Migol is based upon the Globus Toolkit 4 and WSRF. In the following, the experiences made with these frameworks are critically reflected.

11.2 Lessons Learned from OGSA, OGSI, WSRF and GT4

OGSA envisions the Grid as service-oriented environment in which the capabilities of Grid resources are provided via well-defined service interfaces. However, the SOA and Web service paradigm has some drawbacks, as pointed out by the following quotation:

> "Give scientists an API they are happy, give scientists a SOA they are confused." Steve Newhouse

In particular, this criticism applies to the complexity that arises from SOAP services. As a consequence, SOAs often focus on the design of the Web service interface rather than on the capabilities a Grid service should provide. In the following, a brief overview of the Grid standardization efforts and the implementation of these standards within the Globus platforms is given.

11.2.1 Which Standard is the Right One?

> "The good thing about standards is that there are so many to chose from." Andrew Tanenbaum

Since the initial OGSA proposal in 2001, several standards addressing the issue of stateful Web services have been proposed.

- In 2003 the *Open Grid Service Infrastructure (OGSI)* [311] standard was finalized within the Global Grid Forum (the predecessor the Open Grid Forum). OGSI was the first Web service standard, which introduced the concept of stateful Web services. However, OGSI was heavily criticized for its complexity and the lack of tooling support in particular for languages other than Java.
- The *Web Service Resource Framework (WSRF)* [42] was announced as successor of OGSI in 2004. WSRF extracted the ideas of OGSI into different substandard. The standardization body was moved to the OASIS consortium mainly to gain more industry acceptance. WSRF aimed to provide the "convergence" between the Grid and Web service community. However, the complexity did not decrease. At the same time, WSRF lacks certain aspects, which have been originally addressed by OGSI, such as a handle resolution service, which is an important building block for reliable distributed systems.
- In 2006 the WSRF successor *Web Service Resource Transfer (WS-RT)* [266] – the third incarnation of stateful Web services – was announced. Currently, no tooling support for WS-RT exists. No Grid middleware platform substantially invested into WS-RT yet. Thus, if WS-RT interfaces are required, the WS-RT porttypes must be implemented manually.

Unfortunately, none of these standards has reached a great amount of adaptation neither within nor outside the Grid community. Having a mean time to a new standard of less then two years, removes the focus from the service capabilities to the design of the Web service interface. Another concern, that has been raised with every iteration of these standards, is the lack of tooling support.

11.2.2 Web Service Tooling

Since the WSRF is currently the most widely adopted Web service standard, the tooling for WSRF is examined in the following. Different Web service stack with WSRF support currently exists: GT4 (Java and C container), Unicore WSRFLite, WSRF.NET Apache MUSE, WSRF::Lite. The aim of these tools is to provide an Remote Procedure Call (RPC) [55] like abstraction, i. e. stub/skeleton functions, for consuming and creating WSRF services. Unfortunately, the complexity of handling all WSDL datatype, XML namespaces, different message styles, Web service security etc. makes the building of such tools very difficult. While the Java-based tools have gained some maturity, especially the WSRF frameworks for C/C++ are very unsatisfactory. Hence, the integration of C/C++ based applications as found in high performance computing into a WSRF Grid in general requires a lot of manual effort.

While the Globus Java WS core implements a full set of features, it is also associated with some drawbacks especially concerning the underlying SOAP implementation. The framework is still based on Axis1 [38] – the newer Axis2 [39] provides many improvements in terms of usability and performance. For example, Axis2 supports the JAX-WS programming model [6], which simplifies the development of Web services by relying on annotations. The performance improvements can mainly be attributed to the new XML parsing model StAX (Streaming API for XML) [295].

While the Grid community placed a great emphasis on different heavy weight Web service standars, REST as lightweight alternative for Web services became increasingly popular.

11.2.3 REST – Leightweight Alternative to WSRF?

The *Representational State Transfer (REST)*, first described by Fielding [119], addresses similar as WSRF the problem of modeling stateful Web services using the resource abstraction. However, the architecture of REST differs fundamentally from WSRF.

While WSRF relies on endpoint references (as specified by WS-Addressing) to uniquely identify resources, REST utilizes the well-know URI syntax. That means that REST encodes the state identifier into the URI. In contrast to an EPR, which is essentially an XML document stored in a file, an URI is easier to handle, e. g. when referring to a resource in a command line interface. On the other side, EPRs are extensible and allow the storage of arbitrary metadata. This feature is heavily utilized by standards such as WS-Security, WS-Naming and WS-ReliableMessaging.

As described, WSRF specifies different WSDL porttypes for modification of resource states. In contrast, REST simply uses standard HTTP [120] methods to express read and/or write operations on resources. However, clients and services need to agree on the representation format in order to communicate. This can be done e. g. with a XML schema.

REST also has limitations: for security REST services in general rely on transport-level security, i. e. SSL/TLS [105]. End-to-end security as provided by WS-Security is not supported. Further, REST services are accessed via the exchange of XML. Unfortunately, the REST approach does not address the aspect of service description.

Before deploying a complex Web service stack, it must be carefully considered whether the overhead is appropriate. WSRF provides a lot of features related to state management, which might not be required by all use cases, e. g. the support for partial state queries with XPATH or the resource lifetime management (soft state). While the soft state abstraction, i. e. the maintenance of the state for a limited time, is useful to ensure that claimed resources are freed, many services do not require this feature. For example, Migol Job Broker Service solely maintains the state of the user's job until the job has started and does not lock any large amounts of local resources. In this case lifetime management is not necessarily required – a stateless service interface would be sufficient.

Many Cloud offers explicitly prefer REST services to their SOAP counterparts. For example, Google's GData API [7], which can be used to programmatically consume different Cloud services offered by Google, is entirely based on REST.

Further, different Grid middleware platforms, e. g. gLite [112], KnowARC [190] and GridSAM [3], do not use WSRF for state modelling. While these platforms still use SOAP, the state is modelled explicitly within the service interface. That means that a reference to the state is explicitly passed with every call to the service (state id pattern [122]). This approach is easy to implement and can be considered more interoperable than WSRF-style services.

11.2.4 Globus Toolkit 4

The Globus Toolkit 4 offers, especially in comparison with its predecessor the GT3, a mature environment for developing WSRF services. The Grid Security Infrastructure (GSI) provides effective and extensible security services, which address important issues, such as authentication, authorization, message protection and credential delegation. The GT4 services as well as all Migol services can benefit from this comprehensive framework. Further, important Globus services, such as the WS GRAM, have been reengineered and show despite the SOAP overhead a solid performance.

Still, there is room for improvement: Currently, the RFT still causes a significant overhead when staging files with the WS GRAM. Further, the conducted evaluation indicated the severe overhead of the WS-Security stack. Transport-level security and services such as the Delegation Service provide a way to circumvent this performance penalty on the costs of interoperability.

The development of new services is still a complex issue – this is mainly attributed to the complexity of WSRF. Although, the GT4 aims to hide most of the WSRF specific interfaces, a solid understanding of WSRF is the pre-requisite for creating own services.

GT4 provides some support for the creation of fault tolerant services. The persistent resource interface can be used to store the resource state on a stable storage to allow a later recovery. However, the developer is required to serialize and de-serialize the state of the resource manually. Further, there is no support for the fail-over of resources to a backup server. This would be an easy way to increase the fault tolerance of medium and low critical services, such as the WS GRAM. Further, important standards with respect to reliability such as WS-ReliableMessaging [99] and WS-Naming [149] are not supported. Although the predecessor GT3 provided a resolution service [277] for Grid service references (the EPR equivalent of

OGSI), this feature, despite its usefulness for implementing location transparency, is not part of GT4.

Migol demonstrates the usability of the WSRF/Globus toolset to create a fault tolerant environment. However, more lightweight solutions, e. g. REST, are a serious design alternative. In particular, the focus on simplicity avoids many pitfalls associated with the Globus WS-* stacks. Further, a lower complexity also reduces the probability of faults and is therefore an important step towards a reliable infrastructure.

11.3 Future Work

This thesis comprehensively addressed the fault tolerance of distributed Grids focusing on long-running MPI applications. However, the Migol framework itself as well as the described patterns are applicable to many other use cases in distributed computing. Unfortunately, it is nearly impossible to address all possible use cases in a single thesis. In the following, several aspects for future research are highlighted.

As introduced in section 3.5, Cloud Computing describes a new distributed computing paradigm. In particular, the term Cloud is often used to describe the on-demand provision of virtual machines. While this provides an interesting abstraction for applications and services, the fault tolerance of such environments has not been addressed. At the same time, many of the described principles, such as failure detection and recovery, can also be applied to Cloud environments and applications.

Complex application scenarios require the management of various dependencies between tasks. Often, a workflow engine is used to orchestrate different application tasks. Different workflow engines, such as Pegasus [104], the Grid Workflow Execution Service [236], and Swift [339] have been proposed. In general, workflow engines introduce a higher level language, which is used to described the workflow. The individual workflow tasks are executed using a Grid execution management, such as the GRAM. Usually these systems provide basic fault tolerance support by simply polling the job state at the local resource manager. If a failure is detected, the task is usually resubmitted. This approach allows the detections of some errors, however these applications could clearly benefit from an application-level monitoring approach as provided by Migol. At the same time these systems often do not consider the possible failure of the workflow service itself. In the future, the usage of Migol in conjunction with a workflow engine could significantly enhance the reliability of scientific workflows.

Bibliography

[1] Java Native Interface Specification. http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html, 2003.

[2] The Cellular Automaton. http://www.cs.uni-potsdam.de/bs/cellularautomat/, 2006.

[3] GridSAM – Grid Job Submission and Monitoring Web Service. http://gridsam.sourceforge.net/2.0.1/index.html, 2007.

[4] HPC Grid Interoperability Demonstration at SC07. http://www.ogf.org/rotate_headers/documents/HPCBP_Data_Sheet_final1.pdf, 2007.

[5] JSDL SPMD Application Extension, Version 1.0. Open Grid Forum Document GFD.115, 2007. Open Grid Forum.

[6] JSR 224: Java API for XML-Based Web Services (JAX-WS) 2.0. http://jcp.org/en/jsr/detail?id=224, 2007.

[7] Google Data API. http://code.google.com/apis/gdata/, 2008.

[8] Grid Interoperability Now (GIN) Community Group. http://forge.ogf.org/sf/projects/gin, 2008.

[9] Junit.org. http://www.junit.org/, 2008.

[10] LONI – Louisiana Optical Network Initiative. http://www.loni.org, 2008.

[11] OASIS Consortium. http://www.oasis-open.org/, 2008.

[12] Open High Availability Cluster. http://www.opensolaris.org/os/community/ha-clusters/ohac/, 2008.

[13] W3C Consortium. http://www.w3.org/, 2008.

[14] Windows 2003 Server Cluster. http://www.microsoft.com/windowsserver2003/enterprise/clustering.mspx, 2008.

[15] R. Alfieri, R. Cecchini, V. Ciaschini, L. dell'Agnello, Á. Frohner, A. Gianoli, K. Lörentey, and F. Spataro. VOMS, an Authorization System for Virtual Organizations. In F. Fernández Rivera, Marian Bubak, A. Gómez Tato, and Ramon Doallo, editors, *European Across Grids Conference*, volume 2970 of *Lecture Notes in Computer Science*, pages 33–40. Springer, 2003.

[16] W. Allcock. GridFTP: Protocol Extensions to FTP for the Grid. Open Grid Forum Document GFD.22, 2002. Open Grid Forum.

[17] W. Allcock, I. Foster, and R. Madduri. Reliable Data Transport: A Critical Service for the Grid. In *Building Service Based Grids Workshop*, 2004.

[18] G. Allen, P. Bogden, T. Kosar, A. Kulshrestha, G. Namala, S. Tummala, and E. Seidel. Cyberinfrastructure for Coastal Hazard Prediction. *CTWatch Quarterly*, 4(1), 2008.

[19] G. Allen, T. Dramlitsch, I. Foster, N. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus. In *Proceedings of Supercomputing 2001*, Denver, USA, 2001.

[20] G. Allen, T. Goodale, M. Russell, E. Seidel, and J. Shalf. Classifying and Enabling Grid Applications. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 555–578. Wiley, 2003.

[21] P. Alsberg and J. Day. A Principle for Resilient Sharing of Distributed Resources. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, pages 562–570, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[22] Amazon EC^2 Web Service. `http://ec2.amazonaws.com`, 2008.

[23] Amazon S3 Web Service. `http://s3.amazonaws.com`, 2008.

[24] Y. Amir, C. Danilov, and J. Stanton. A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, pages 327–336, Washington, DC, USA, 2000. IEEE Computer Society.

[25] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. The Totem Single-Ring Ordering and Membership Protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, 1995.

[26] R. Ananthakrishnan, M. D'Arcy, and T. Howe. Globus WS Core and Tools. In *Talk at Open Source Grid & Cluster Conference*, 2008.

[27] Steve Anderson, Jeff Bohren, Toufic Boubez, Marc Chanliau, Giovanni Della-Libera, Brendan Dixon, Praerit Garg, Phillip Hallam-Baker, Maryann Hondo, Chris Kaler, Hal Lockhart, Robin Martherus, Hiroshi Maruyama, Nataraj Nagaratnam, Andrew Nash, Rob Philpott, Darren Platt, Hemma Prafullchandra, Maneesh Sahu, John Shewchuk,

Dan Simon, Davanum Srinivas, Elliot Waingold, David Waite, Doug Walter, and Riaz Zolfonoon. Web Service Trust Language (WS-Trust). ftp://www6.software.ibm.com/software/developer/library/ws-trust.pdf, 2005.

[28] S. Andreozzi, F. Ehm, L. Field, and B. Kónya. GLUE Specification. https://forge.gridforum.org/sf/projects/glue-wg, 2007.

[29] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Specification (WS-Agreement). Open Grid Forum Document GFD.107, 2007. Open Grid Forum.

[30] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. McGough, and D. Pulsipher. Job Submission Description Language (JSDL) Specification 1.0. http://www.gridforum.org/documents/GFD.56.pdf, 2005.

[31] Apache Sandesha. http://ws.apache.org/sandesha/, 2008.

[32] D. C. Arnold, S. S. Vahdiyar, and J. J. Dongarra. On the Convergence of Computational and Data Grids. *Parallel Processing Letters*, 11(2–3):187–202, 2001.

[33] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[34] AstroGrid-D – Use Cases. http://www.gac-grid.de/project-documents/UseCases.html, 2008.

[35] A. Avižienis. Fault-Tolerance and Fault-Intolerance: Complementary Approaches to Reliable Computing. In *Proceedings of the international conference on Reliable software*, pages 458–464, New York, NY, USA, 1975. ACM.

[36] A. Avižienis. Fault-Tolerant Systems. *IEEE Transactions on Computers*, 25(12):1304–1312, 1976.

[37] A. Avižienis, J.-C. Laprie, and B. Randell. Dependability and its Threats - A Taxonomy. In René Jacquart, editor, *IFIP Congress Topical Sessions*, pages 91–120. Kluwer, 2004.

[38] Axis Web Services. http://ws.apache.org/axis/, 2006.

[39] Axis2 Web Services. http://ws.apache.org/axis2/, 2008.

[40] R. Badia, R. Hood, T. Kielmann, A. Merzky, C. Morin, S. Pickles, M. Sgaravatto, P. Stodghill, Nathan Stone, and Heon Y. Yeom. Use Cases for Grid Checkpoint and Recovery. Grid Forum Document GFD.92, May 2007. Open Grid Forum.

[41] B. Ban. Design and Implementation of a Reliable Group Communication Toolkit for Java. `http://www.jgroups.org/javagroupsnew/docs/papers/Coots.ps.gz`, 1998.

[42] T. Banks. *Web Services Resource Framework (WSRF) – Primer v1.2*. OASIS, 2006.

[43] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[44] Jim Basney, Marty Humphrey, and Von Welch. The MyProxy Online Credential Repository. `http://www.ncsa.uiuc.edu/~jbasney/myproxy-spe.pdf`, 2005.

[45] D. Battré, O. Kao, and K. Voss. Implementing WS-Agreement in a Globus Toolkit 4.0 Environment. `http://www.assessgrid.eu/fileadmin/AssessGrid/usermounts/publications/papers/Implementing-WS-Agreement-Abstract.pdf`, 2008.

[46] Marc-Ellian Bégin. Grids and clouds – evolution or revolution. `https://edms.cern.ch/file/925013/3/EGEE-Grid-Cloud.pdf`, 2008.

[47] E. Benson, G. Wasson, and M. Humphrey. Evaluation of UDDI as a Provider of Resource Discovery Services for OGSA-Based Grids. *ipdps*, 0:18, 2006.

[48] D. Berry, A. Luniewski, and M. Antonioletti. OGSA Data Architecture. Open Grid Forum Document GFD.121, 2007. Open Grid Forum.

[49] B. Bieker, E. Maehle, G. Deconinck, and J. Vounckx. Reconfiguration and Checkpointing in Massively Parallel Systems. In *European Dependable Computing Conference*, pages 353–370, 1994.

[50] K. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, 1993.

[51] K. Birman. *Reliable Distributed Systems – Technologies, Web Services and Applications*. Springer, New York, USA, 2005.

[52] K. Birman, R. Constable, M. Hayden, J. Hickey, C. Kreitz, R. Van Renesse, O. Rodeh, and W. Vogels. The Horus and Ensemble Projects: Accomplishments and Limitations. Technical report, Ithaca, NY, USA, 1999.

[53] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. Technical report, Ithaca, NY, USA, 1987.

[54] K. Birman and T. A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[55] A. Birrell and B. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[56] V. Blagojevic. Implementing Totem's Total Ordering Protocol in JavaGroups Reliable Group Communication Toolkit. `http://www.jgroups.org/javagroupsnew/docs/papers/totaltoken.ps.gz`, 2000.

[57] Berkeley Lab Checkpoint/Restart (BLCR). `http://ftg.lbl.gov/CheckpointRestart/CheckpointRestart.shtml`, 2008.

[58] P. S. Bogden, T. Gale, G. Allen, J. MacLaren, G. Almes, G. Creager, J. Bintz, L. D. Wright, H. Graber, N. Williams, S. Graves, H. Conover, K. Galluppi, R. Luettich, W. Perrie, B. Toulany, Y. P. Sheng, J. R. Davis, H. Wang, and D. Forrest. Architecture of a Community Infrastructure for Predicting and Analyzing Coastal Inundation. In *Marine Technology Society Journal*, volume 41, pages 53–71, 2007.

[59] D. Booth, H. Haas, F.McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, editors. *Web Services Architecture*. W3C, 2004.

[60] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. MPICH-V: A Multiprotocol Fault Tolerant MPI. *International Journal of High Performance Computing and Applications*, 20 (3):319–333, 2006.

[61] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), October 1989.

[62] J. Bresnahan, M. Link, G. Khanna, Z. Imani, R. Kettimuthu, and I. Foster. Globus GridFTP: What's New in 2007. In *Proceedings of the First International Conference on Networks for Grid Applications*, 2007.

[63] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated Application-Level Checkpointing of MPI Programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 84–94, New York, NY, USA, 2003. ACM.

[64] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. The Primary-Backup Approach. pages 199–216, 1993.

[65] S. Burke, S. Campana, P. Lorenzo, C. Nater, R. Santinelli, and A. Sciabà. *gLite 3.1 User Guide*. EGEE/CERN, 2008.

[66] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 24–24, Berkeley, CA, USA, 2006. USENIX Association.

[67] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid, 2000.

[68] N. Carr. *The Big Switch: Rewiring the World, from Edison to Google*. W. W. Norton, January 2008.

[69] N. Carr. Crash: Amazon's S3 utility goes down. `http://www.roughtype.com/archives/2008/02/amazons_s3_util.php`, 2008.

[70] T. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.

[71] K. Chandy. A Survey of Analytic Models of Rollback and Recovery Stratergies. *Computer*, 8(5):40–47, 1975.

[72] K. Chandy. A Mutual Exclusion Algorithm for Distributed Systems. Technical report, University of Texas, 1982.

[73] K. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[74] S. J. Chapin, D. Katramatos, J. F. Karpovich, and A. S. Grimshaw. The Legion Resource Management System. In *IPPS/SPDP '99/JSSPP '99: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 162–178, London, UK, 1999. Springer-Verlag.

[75] W. Chen, S. Toueg, and M.K. Aguilera. On the Quality of Service of Failure Detectors. *IEEE Transactions on Computers*, 51(5):561–580, 2002.

[76] A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunszt, M. Ripeanu, B. Schwartzkopf, H. Stockinger, K. Stockinger, and B. Tierney. Giggle: A Framework for Constructing Scalable Replica Location Services. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–17, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[77] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets, 1999.

[78] M. Chetty and R. Buyya. Weaving Computational Grids: How Analogous Are They with Electrical Grids? *Computing in Science and Engg.*, 4(4):61–71, 2002.

[79] G. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[80] T. Chou. Beyond Fault Tolerance. *IEEE Computer*, 30(4):47–49, 1997.

[81] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *Web Services Description Language (WSDL) 1.1*. W3C, 2001.

[82] C.T. Chu, S.K. Kim, Y.A. Lin, Y.Y. Yu, G. Bradski, A.Y. Ng, and K. Olukotun. Map-Reduce for Machine Learning on Multicore. 2006.

[83] D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 106–114, New York, NY, USA, 1988. ACM.

[84] J. Clark and S. DeRose. *XML Path Language (XPath) Version 1.0 XML Path Language (XPath) Version 1.0*. W3C, 1999.

[85] UDDI Version 2.03 Replication Specification. `http://uddi.org/pubs/Replication_v2.pdf`, 2002.

[86] A. Colesa, T. Pop, I. Ignat, and C. Ardelean. Automatic and Reliable Distribution of Data in Grids over Globus Toolkit. In *SYNASC '07: Proceedings of the Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 310–316, Washington, DC, USA, 2007. IEEE Computer Society.

[87] W3C Consortium. SOAP Version 1.2. `http://www.w3.org/TR/soap12-part1/`.

[88] CREAM-BES. `http://grid.pd.infn.it/omii/cream-bes`, 2008.

[89] F. Cristian. Agreeing on Who is Present and Who is Absent in a Synchronous Distributed System. pages 206–11, 1988.

[90] F. Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, 34(2):56–78, 1991.

[91] F. Cristian. Synchronous and Asynchronous Group Communication. *Communications of the ACM*, 39(4):88–97, 1996.

[92] China Research and Development Environment over Wide-Aread Network (CROWN). `http://www.crown.org.cn/en/`, 2004.

[93] K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe. The WS-Resource Framework. `http://www.oasis-open.org/committees/download.php/6796/ws-wsrf.pdf`, 2005.

[94] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. In *IPPS/SPDP '98: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, London, UK, 1998. Springer-Verlag.

[95] D-Grid – German Grid Initiative. `http://www.d-grid.de/`, 2008.

[96] J. da Silva and V. Rebello. Low Cost Self-healing in MPI Applications. In *Proceedings of the 14th EuroPVM/MPI*, volume 4757 of *Lecture Notes in Computer Science*, pages 144–152, Berlin, Heidelberg, September/October 2007. Springer-Verlag.

[97] C. Dabrowski. Reliability in Grid Computing Systems. *Concurrency and Computation: Practice and Experience (OGF Special Issue)*, 2008.

[98] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a Partitioned Network: A Survey. *ACM Comput. Surv.*, 17(3):341–370, 1985.

[99] D. Davis, A. Karmarkar, G. Pilz, S. Winkler, and U. Yalcinalp. Web Services Reliable Messaging (WS-ReliableMessaging) Version 1.1. `http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf`, 2007.

[100] dCache Homepage. `http://www.dcache.org/`, 2008. 05/2008.

[101] R. De Camargo, F. Kon, and A. Goldman. Portable Checkpointing and Communication for BSP Applications on Dynamic Heterogeneous Grid Environments. *17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'05)*, 0:226–234, 2005.

[102] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 137–150, Berkeley, CA, USA, 2004. USENIX Association.

[103] G. Deconinck, J. Vounckx, R. Cuyvers, and R. Lauwereins. Survey of Checkpointing and Rollback Techniques. Technical report, Katholieke Universiteit Leuven, Belgium, 1993.

[104] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. Berriman, J. Good, A. Laity, J. Jacob, and D. Katz. Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Sci. Program.*, 13(3):219–237, 2005.

[105] T. Dierks and C. Allen. The TLS Protocol Version 1.0. `http://www.ietf.org/rfc/rfc2246.txt`, 1999.

[106] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[107] B. Dillaway, M. Humphrey, C. Smith, M. Theimer, and G. Wasson. HPC Basic Profile, Version 1.0. Open Grid Forum Document GFD.114, 2007. Open Grid Forum.

[108] Distributed Management Task Force, Inc. . Common Information Model (CIM) Standard. DTMF Web Page, 2007.

[109] A. Downey. Using Queue Time Predictions for Processor Allocation. In *IPPS '97: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 35–57, London, UK, 1997. Springer-Verlag.

[110] J. Duell. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical report, Lawrence Berkeley National Laboratory, November 2003.

[111] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *J. ACM*, 35(2):288–323, 1988.

[112] EGEE Middleware Architecture. `https://edms.cern.ch/file/476451/1.0/architecture.pdf`, 2004.

[113] E. Elnozahy, L. Alvisi, Y.-M. Wang, and D. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[114] R. Van Engelen and K. Gallivan. The gSOAP Toolkit for Web Services and Peer-to-Peer Computing Networks. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 128, Washington, DC, USA, 2002. IEEE Computer Society.

[115] G. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London, and J. Dongarra. Extending the MPI Specification for Process Fault Tolerance on High Performance Computing Systems. In *Proceedings of ISC2004*, Heidelberg, Germany, June 2004.

[116] D. Fallside. XML schema part 0: Primer. first edition of a recommendation, W3C, May 2001. http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/.

[117] P. Felber and P. Narasimhan. Experiences, Strategies, and Challenges in Building Fault-Tolerant CORBA Systems. *IEEE Transactions on Computers*, 53(5):497–511, 2004.

[118] M. Feller, I. Foster, and S. Martin. GT4 GRAM: A Functionality and Performance Study. In *Proceedings of the Teragrid 2007 Conference*, Madison, WI, USA, 2007.

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

[120] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, 1999.

[121] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with one Faulty Process. *J. ACM*, 32(2):374–382, 1985.

[122] I. Foster. How Do I Model State? Let Me Count the Ways. `http://sorma.fzi.de/images/9/90/Fos05c.pdf`, 2005.

[123] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *Proceedings of IFIP International Conference on Network and Parallel Computing*, pages 2–13. Springer-Verlag LNCS 3779, 2006.

[124] I. Foster. Service-Oriented Science: Scaling eScience Impact. In *IAT '06: Proceedings of the IEEE/WIC/ACM international conference on Intelligent Agent Technology*, pages 9–10, Washington, DC, USA, 2006. IEEE Computer Society.

[125] I. Foster, D. Gannon, H. Kishimoto, and J. Von Reich. Open Grid Service Architecture Use Cases. Open Grid Forum Document GFD.29, 2004. Open Grid Forum.

[126] I. Foster, A. Grimshaw, P. Lane, W. Lee, M. Morgan, S. Newhouse, S. Pickles, D. Pulsipher, C. Smith, and M. Theimer. OGSA Basic Execution Service – Version 1.0. 2007.

[127] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In *Proceedings of the International Workshop on Quality of Service*, 1999.

[128] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid. `http://www-unix.globus.org/toolkit/3.0/ogsa/docs/physiology.pdf`, 2002.

[129] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *ACM Conference on Computer and Communications Security*, pages 83–92, 1998.

[130] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid. `http://www.globus.org/research/papers/anatomy.pdf`, 2001.

[131] I. Foster, Kishimoto, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. Von Reich. The Open Grid Services Architecture, Version 1.5. Grid Forum Document GFD.80, 2006. Global Grid Forum.

[132] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5(3):237–246, July 2002.

[133] E. Gabriel, G. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[134] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[135] Ganglia homepage. `http://ganglia.info/`, 2008.

[136] H. Garcia-Molina. Elections in a Distributed Computing System. *IEEE Transactions on Computers*, 31(1):48–59, 1982.

[137] M. Gardner. Mathematical Games: The Fantastic Combinations of John Conway's new Solitaire Game "life". *Scientific American*, 223:120–123, 1970.

[138] D. Gifford. Weighted Voting for Replicated Data. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162, New York, NY, USA, 1979. ACM.

[139] Globus Alliance. GT Security (GSI). URL: `http://www.globus.org/toolkit/security/`, 2005.

[140] Globus Alliance. GT4 Delegation Service. `http://www.globus.org/toolkit/docs/4.0/security/delegation/`, 2005.

[141] Globus Alliance. Data Replication Service (DRS). `http://www.globus.org/toolkit/docs/4.0/techpreview/datarep/`, 2008.

[142] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf. The Cactus Framework and Toolkit: Design and Applications. In *Vector and Parallel Processing - VECPAR '2002, 5th International Conference*. Springer, 2003.

[143] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. von Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf. SAGA: A Simple API for Grid Applications, High-Level Application Programming on the Grid. *Computational Methods in Science and Technology*, 12(1):7–20, 2006.

[144] J. Gray. Why Do Computers Stop and What Can Be Done About It? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[145] J. Gray. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4):409–418, 1990.

[146] J. Gray. Distributed Computing Economics. *Computer Research Repository (CoRR)*, cs.NI/0403019, 2004.

[147] J. Gray and M. Anderton. Distributed Computer Systems: Four Case Studies. *Proceedings of the IEEE*, 75(5):719–726, May 1987.

[148] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM.

[149] A. Grimshaw, M. Morgan, and K. Sarnowska. WS-Naming: Location Migration, Replication, and Failure Transparency Support for Web Services. *Concurrency and Computation: Practice and Experience (OGF Special Issue)*, 2008.

[150] W. Gropp and E. Lusk. Fault Tolerance in MPI Programs. *High Performance Computing and Applications*, 2002.

[151] M. Grottke and K. Trivedi. Fighting bugs: Remove, retry, replicate, and rejuvenate. *Computer*, 40(2):107–109, 2007.

[152] Web Services Secure Conversation Language (WS-SecureConversation). `ftp://www6.software.ibm.com/software/developer/library/ws-secureconversation.pdf`, 2005.

[153] I. Gupta, T. Chandra, and G. Goldszmidt. On Scalable and Efficient Distributed Failure Detectors. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 170–179, New York, NY, USA, 2001. ACM.

[154] M. Gusowski, A. Luckow, B. Schnor, and M. Schütte. Experiences with a Resilient, MPI-based Master-Worker Application in a Failure-Prone Grid Environment. Technical report, University of Potsdam, Potsdam, 2008.

[155] I. Haddad, C. Leangsuksun, and S. Scott. HA-OSCAR: The Birth of Highly Available OSCAR. *Linux Journal*, 2003(115):1, 2003.

[156] Hadoop: Open Source Implementation of MapReduce. `http://hadoop.apache.org/`, 2008.

[157] N. Hallama, A. Luckow, and B. Schnor. Grid Security for Fault Tolerant Grid Applications. In *ISCA 19th International Conference on Parallel and Distributed Computing Systems*, pages 76–83, San Francisco, USA, 2006.

[158] R. Hanmer. *Patterns For Fault Tolerant Software*. John Wiley & Sons Ltd, West Sussex, England, 2007.

[159] U. Hansmann. Parallel Tempering Algorithm for Conformational Studies of Biological Molecules. In *Chemical Physics Letters*, volume 281, pages 140–150, 1997.

[160] M. Hayden. *The Ensemble System*. PhD thesis, Department of Computer Science, Cornell University, 1998.

[161] R. Henderson and D. Tweten. Portable Batch System: External Reference Specification. Technical report, NASA Ames Research Center, 1996.

[162] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[163] Hibernate Homepage. `http://www.hibernate.org/`, 2008.

[164] R. Housley, W. Ford, W. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. `http://www.ietf.org/rfc/rfc3280.txt`, 2002.

[165] M. Hovestadt. Fault Tolerance Mechanisms for SLA-aware Resource Management. *11th International Conference on Parallel and Distributed Systems*, 02:458–462, 2005.

[166] Y. Huang and C. Kintala. Software Fault-Tolerance in the Application Layer. In M. R. Lyu, editor, *Software Fault Tolerance*, pages 231–248. John Wiley & sons, 1995.

[167] E. Hudo, R. Montero, and I. Llorente. The GridWay Framework for Adaptive Scheduling and Execution on Grids. In *Scalable computing: practice and experience (SCPE)*, volume 6, 2005.

[168] M. Humphrey, C. Smith, M. Theimer, and G. Wasson. JSDL HPC Profile Application Extension, Version 1.0. Open Grid Forum Document GFD.115, 2007. Open Grid Forum.

[169] J. Hursey, J. Squyres, and A. Lumsdaine. A Checkpoint and Restart Service Specification for Open MPI. Technical Report TR635, Indiana University, Bloomington, Indiana, USA, July 2006.

[170] A. Iosup, M. Jan, O. Sonmez, and D. Epema. On the Dynamic Resource Availability in Grids. In *8th IEEE/ACM International Conference on Grid Computing (Grid 2007)*, April 2007.

[171] D. Jackson, Q. Snell, and M. Clement. Core Algorithms of the Maui Scheduler. *Lecture Notes in Computer Science*, 2221:87, 2001.

[172] V. Jacobson. Congestion Avoidance and Control. In *ACM SIGCOMM '88*, pages 314–329, Stanford, CA, August 1988.

[173] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, USA, 1998.

[174] JSR 244: Java Platform, Enterprise Edition 5 (Java EE 5) Specification. `http://jcp.org/en/jsr/detail?id=244`, 2008.

[175] J. Jeske, A. Luckow, and B. Schnor. Reservation-based Resource-Brokering for Grid Computing. In *Proceedings of German e-Science Conference*, Baden-Baden, Germany, 2007.

[176] S. Jha, H. Kaiser, Y. El Khamra, and O. Weidner. Design and implementation of network performance aware applications using saga and cactus. In *Accepted for 3rd IEEE Conference on eScience2007 and Grid Computing, Bangalore, India.*, 2007.

[177] S. Jha, A. Merzky, and G. Fox. Using Clouds to Provide Grids Higher-Levels of Abstraction and Explicit Support for Usage Modes. *Concurrency and Computation: Practice and Experience (OGF Special Issue)*, 2008.

[178] H. Jin, D. Zou, H. Chen, J. Sun, and S. Wu. Fault-Tolerant Grid Architecture and Practice. *Journal of Computer Science and Technology*, 18(4):423–433, 2003.

[179] A. Jindal, S. Lim, S. Radia, and W.-L. Chang. Load Balancing for Replicated Services. `http://www.freepatentsonline.com/6324580.html`, 2001.

[180] D. Johnson and W. Zwaenepoel. Recovery in Distributed Systems using Asynchronous Message Logging and Checkpointing. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 171–181, New York, NY, USA, 1988. ACM.

[181] M. Kaashoek, A. Tanenbaum, and S. Hummel. An Efficient Reliable Broadcast Protocol. *SIGOPS Operating Systems Review*, 23(4):5–19, 1989.

[182] H. Kaiser, A. Merzky, S. Hirmer, and G. Allen. The SAGA C++ Reference Implementation. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'06) - Library-Centric Software Design (LCSD'06)*, Portland, OR, USA, October, 22-26 2006.

[183] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *CoRR*, cs.DC/0206040, 2002. informal publication.

[184] K. Keahey, I. Foster, T. Freeman, and X. Zhang. Virtual Workspaces: Achieving Quality of Service and Quality of Life in the Grid. *Scientific Programming*, 13(4):265–275, 2005.

[185] A. Keller and A. Reinefeld. CCS Resource Management in Networked HPC Systems. In *Proc. Heterogenous Computing Workshop HCW#98 at IPPS*, pages 44–56, Orlando, Florida, 1998. IEEE Comp. Society Press.

[186] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[187] Omid Khalili, Jiahua He, Catherine Olschanowsky, Allan Snavely, and Henri Casanova. Measuring the Performance and Reliability of Production Computational Grids. In *GRID*, pages 293–300. IEEE, 2006.

[188] A. Knebe. AMIGA Project Homepage. http://www.aip.de/People/AKnebe/AMIGA/, 2008.

[189] A. Knebe and C. Power. On the Correlation between Spin Parameter and Halo Mass. *The Astrophysical Journal*, 678:621–626, May 2008.

[190] KnowARC – Grid-enabled Know-how Sharing Technology Based on ARC Services and Open Standards. `http://www.knowarc.eu/`, 2008.

[191] G. Kola, T. Kosar, and M. Livny. Faults in Large Distributed Systems and What We Can Do About Them. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par*, volume 3648 of *Lecture Notes in Computer Science*, pages 442–453. Springer, 2005.

[192] N. Kolettis and N. Fulton. Software Rejuvenation: Analysis, Module and Applications. In *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, page 381, Washington, DC, USA, 1995. IEEE Computer Society.

[193] M. Komann, C. Kauhaus, and D. Fey. Calculation of Single-File Diffusion Using Grid-Enabled Parallel Generic Cellular Automata Simulation. In *Proceedings of the 12th EuroPVM/MPI*, pages 528–535, 2005.

[194] J. Kovács and P. Kacsuk. A Migration Framework for Executing Parallel Programs in the Grid. In Marios D. Dikaiakos, editor, *European Across Grids Conference*, volume 3165 of *Lecture Notes in Computer Science*, pages 80–89. Springer, 2004.

[195] H. Kung and J. Robinson. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[196] R. Ladin, B. Liskov, and L. Shrira. Lazy Replication: Exploiting the Semantics of Distributed Services. In *PODC '90: Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 43–57, New York, NY, USA, 1990. ACM.

[197] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[198] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[199] L. Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[200] G. Lanfermann. *Nomadic Migration – A Service Environment for Autonomic Computing on the Grid.* PhD thesis, University of Potsdam, Germany, 2003.

[201] G. Lanfermann, G. Allen, T. Radke, and E. Seidel. Nomadic Migration: Fault Tolerance in a Disruptive Grid Environment. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 280, Washington, DC, USA, 2002. IEEE Computer Society.

[202] G. Lanfermann, B. Schnor, and E. Seidel. Grid Object Description: Characterizing Grids. In *Eighth IFIP/IEEE International Symposium on Integrated Network Management (IM 2003)*, pages 519–532, Colorado Springs, Colorado, USA, 2003.

[203] J.C. Laprie, A. Avižienis, and H. Kopetz, editors. *Dependability: Basic Concepts and Terminology*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1991.

[204] Craig Lee and Domenico Talia. Grid programming models: Current tools, issues and directions. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 555–578. Wiley, 2003.

[205] P. A. Lee. Software-Faults: The Remaining Problem in Fault Tolerant Systems? In *Workshop on Hardware and Software Architectures for Fault Tolerance: Experiences and Perspectives*, pages 171–181, London, UK, 1994. Springer-Verlag.

[206] Hui Li, David Groep, Lex Wolters, and Jeff Templon. Job Failure Analysis and Its Implications in a Large-Scale Production Grid. *e-science*, 0:27, 2006.

[207] K. Limaye, B. Leangsuksun, V. Munganuru, Z. Greenwood, S. Scott, R. Libby, and K. Chanchio. Grid-Aware HA-OSCAR. *19th International Symposium on 19th International Symposium on 19th International Symposium on 19th International Symposium on High Performance Computing Systems and Applications*, 00:333–339, 2005.

[208] R. Lindsay. Dealers in the dark after systems failure dealers in the dark after systems failure. `http://business.timesonline.co.uk/tol/business/markets/article2828050.ece`, 2007.

[209] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, 1988.

[210] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the COndor Distributed Processing System. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[211] C. Liu and D. Booth. Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. W3C recommendation, W3C, June 2007. http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626.

[212] A. Luckow and B. Schnor. Migol: A Fault Tolerant Service Framework for MPI Applications in the Grid. In *15th European PVM/MPI User's Group Meeting - Sorento, Italy*, pages 258–267, 2005.

[213] A. Luckow and B. Schnor. Migol: A fault-tolerant service framework for mpi applications in the grid. In *Euro PVM/MPI 2005*, volume 3666/2005, pages 258–267. Springer, October 2005.

[214] A. Luckow and B. Schnor. Migol: A Fault Tolerant Service Framework for Grid Computing – Evolution to WSRF. In *Proceedings 2. Workshop: Grid-Technologie für den Entwurf technischer Systeme*, pages 47–54, Dresden, 2006.

[215] A. Luckow and B. Schnor. Service Replication in Grids: Ensuring Consistency in a Dynamic, Failure-Prone Environment. In *Proceedings of Fifth High-Performance Grid Computing Workshop in conjunction with IEEE International Parallel & Distributed Processing Symposium*, Miami, USA, 2008.

[216] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 8–8, Berkeley, CA, USA, 2004. USENIX Association.

[217] S. Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. In *COOTS'95: Proceedings of the USENIX Conference on Object-Oriented Technologies on USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 10–10, Berkeley, CA, USA, 1995. USENIX Association.

[218] I. Mandrichenko, W. Allcock, and T. Perelmutov. GridFTP v2 Protocol Description. Open Grid Forum Document GFD.47, 2005. Open Grid Forum.

[219] Satoshi Matsuoko. To Distribute or Not to Distribute, That is the Question in Petascale and Beyond (Keynote). `http://www.mardigrasconference.org/slides/a4-matsuoka.pdf`, 01 2008.

[220] J. McCarthy. MIT Centennial Speech. 1961. *Cited in* Architects of the Information Society: Thirty-five Years of the Laboratory for Computer Science at MIT. S. L. Garfinkel (Ed), MIT Press, Cambridge MA, 1999.

[221] M. S. Memon, A. S. Memon, M. Riedel, B. Schuller, D. Mallmann, B. Tweddell, A. Streit, S. van de Berghe, D. Snelling, V. Li, M. Marzolla, and P. Andreetto. Enhanced Resource Management Capabilities Using Standardized Job Management and Data Access Interfaces within UNICORE Grids. *13th International Conference on Parallel and Distributed Systems*, 2:1–6, 2007.

[222] Roger Menday. The web services architecture and the unicore gateway. *aict-iciw*, 0:134, 2006.

[223] A. Merzky. SAGA CPR Draft, 2008.

[224] A. Merzky and S. Jha. A Requirements Analysis for a Simple API for Grid Applications. Open Grid Forum Document GFD.71, 2006. Open Grid Forum.

[225] D. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process Migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

[226] H. Miranda, A. Pinto, and L. Rodrigues. Appia: A Flexible Protocol Kernel Supporting Multiple Coordinated Channels. *21st IEEE International Conference on Distributed Computing Systems (ICDCS'01)*, 00:0707, 2001.

[227] Kenichi Miura. Overview of Japanese Science Grid Project NAREGI. http://www.nii.ac.jp/pi/n3/3_67.pdf, 2006.

[228] R. S. Montero. OpenNebula: Open Source Virtual Machine Manager for Cluster Computing. In *Talk at Open Source Grid & Cluster Conference*, 2008.

[229] M. Morgan and A. Grimshaw. Genesis II - Standards Based Grid Computing. *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, 00:611–618, 2007.

[230] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal. Extended Virtual Synchrony. In *The 14th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, 1994.

[231] *MPI: A Message-Passing Interface Standard*. Message Passing Interface Forum, 1995.

[232] *MPI-2: Extensions to the Message-Passing Interface*. Message Passing Interface Forum, 1997.

[233] MPICH2. MPICH2 Project Homepage. http://www-unix.mcs.anl.gov/mpi/mpich2/, 2008.

[234] Jarek Nabrzyski, Jennifer M. Schopf, and Jan Weglarz, editors. *Grid Resource Management: State of the Art and Future trends*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[235] P. Narasimhan. *Transparent Fault Tolerance for CORBA*. PhD thesis, 1999. Chair-Louise E. Moser.

[236] Falk Neubauer, Andreas Hoheisel, and Joachim Geiler. Workflow-based Grid Applications. *Future Generation Computer Systems – The International Journal of Grid Computing: Theory, Methods and Application*, 22(1):6–15, 2006.

[237] John Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966.

[238] A. Nguyen-Tuong, A. Grimshaw, G. Wasson, M. Humphrey, and J. Knight. Towards Dependable Grids. http://www.cs.virginia.edu/~techrep/CS-2004-11.pdf.

[239] OASIS. SAML V2.0 Standard. URL: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security#samlv20, 2005.

[240] OASIS. Web Services Security: SOAP Message Security 1.1. URL: http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf, 2006.

[241] Open Grid Forum. http://www.ogf.org, 2008.

[242] OMG. Common Object Request Broker Architecture (CORBA) Specification, Version 3.1. http://www.omg.org/spec/CORBA/3.1/, 2008.

[243] OpenMP Application Program Interface – Version 2.5. http://www.openmp.org/mp-documents/spec25.pdf, 2005.

[244] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do Internet Services Fail, and What can be Done About it? In *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[245] OSCAR – Open Source Cluster Application Resource. http://oscar.openclustergroup.org/, 2008.

[246] J. Osrael, L. Froihofer, M. Weghofer, and K. Goeschka. Axis2-Based Replication Middleware for Web Services. *icws*, 0:591–598, 2007.

[247] C. D. Ott, E. Schnetter, G. Allen, E. Seidel, J. Tao, and B. Zink. A Case Study For Petascale Applications in Astrophysics: Simulating Gamma-Ray Bursts. In *MG '08: Proceedings of the 15th ACM Mardi Gras conference*, pages 1–9, New York, NY, USA, 2008. ACM.

[248] M. Palankar, A. Onibokun, and Adriana Iamnitchi. Amazon S3 for Science Grids: A Viable Solution? In *Proceedings of 4th USENIX Symposium on Networked Systems Design & Implementation*, Cambridge, MA, USA, 2007.

[249] P. Papadopoulos, M. Katz, and G. Bruno. NPACI Rocks: Tools and Techniques for Easily Deploying Manageable Linux Clusters. *cluster*, 00:258, 2001.

[250] M. Parashar and J. C. Browne. Conceptual and implementation models for the grid. In *Proceedings of the IEEE, Special Issue on Grid Computing*, volume 93, pages 653–668. IEEE Press, March 2005.

[251] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[252] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical report, Berkeley, CA, USA, 2002.

[253] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In Haran Boral and Per-Åke Larson, editors, *SIGMOD Conference*, pages 109–116. ACM Press, 1988.

[254] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, 1980.

[255] Satish Penmatsa, Anthony T. Chronopoulos, Nicholas T. Karonis, and Brian R. Toonen. Implementation of Distributed Loop Scheduling Schemes on the TeraGrid. *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, page 361, 2007.

[256] S. Petri, B. Schnor, and H. Langendörfer. PBeam – Fehlertoleranz für verteilte Anwendungen mittels Migration und Checkpointing. In Clemens H. Cap, editor, *siwork96*, pages 91–102, Universität Zürich, Institut für Informatik, May 1996. vdf Hochschulverlag AG an der ETH Zürich.

[257] J. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. Skeel, L. Kale, and K. Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26:1781–1802, 2005.

[258] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure Trends in a Large Disk Drive Population. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[259] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of USENIX Winter1995 Technical Conference*, pages 213–224, New Orleans, Louisiana/U.S.A., January 1995.

[260] J. Postel and J. K. Reynolds. RFC 959: File transfer protocol, October 1985. Status: STANDARD.

[261] Jon Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981.

[262] R. Stewart and Q. Xie and M. Stillman and M. Tüxen. Aggregate server access protocol (asap). Technical report, September 2007.

[263] B. Randell. System Structure for Software Fault Tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[264] K. Ranganathan and I. Foster. Identifying Dynamic Replication Strategies for a High-Performance Data Grid. In *GRID '01: Proceedings of the Second International Workshop on Grid Computing*, pages 75–86, London, UK, 2001. Springer-Verlag.

[265] Ralf Ratering, Alexander Lukichev, Morris Riedel, Daniel Mallmann, A. Vanni, C. Cacciari, S. Lanzarini, K. Benedyczak, M. Borcz, R. Kluszcynski, Piotr Bala, and Gert Ohme. Gridbeans: Support e-science and grid applications. *e-science*, 0:45, 2006.

[266] Brian Reistad, Bryan Murray, Doug Davis, Ian Robinson, Raymond McCollum, Alexander Nosov, Steve Graham, Vijay Tewari, and William Vambenepe. Web Services Resource Transfer (WS-RT). Technical Report 1.0, IBM, HP, Microsoft, Intel, August 2006.

[267] RESERVOIR - Resources and Services Virtualization Without Barriers. `http://www.reservoir-fp7.eu/`, 2008.

[268] M. Riedel, E. Laure, T. Soddemann et. al. Interoperation of World-Wide Production e-Science Infrastructures. *Concurrency and Computation: Practice and Experience (OGF Special Issue)*, 2008.

[269] A. Robertson. Linux-HA Heartbeat System Design. In *ALS'00: Proceedings of the 4th conference on 4th Annual Linux Showcase & Conference, Atlanta*, pages 20–20, Berkeley, CA, USA, 2000. USENIX Association.

[270] M. Roehrig, W. Ziegler, and P. Wieder. Grid Scheduling Dictionary of Terms and Keywords. Open Grid Forum Document GFD.11, 2002. Open Grid Forum.

[271] Mathilde Romberg and Dietmar W. Erwin. Unicore-uniformes interface für computer ressourcen. *Praxis der Informationsverarbeitung und Kommunikation*, 24(2), 2001.

[272] Jothy Rosenberg and David Remy. *Securing Web Services with WS-Security*. SAMS Publishing, 2004.

[273] S. M. Sadjadi, S. Shimizu, J. Figueroa, R. Rangaswami, J. Delgado, H. Duran, and X. J. Collazo-Mojica. A Modeling Approach for Estimating Execution Time of Long-Running Scientific Applications. In *Proceedings of Fifth High-Performance Grid Computing Workshop in conjunction with IEEE International Parallel & Distributed Processing Symposium*, Miami, USA, 2008.

[274] Jorge Salas, Francisco Perez-Sorrosal, Marta Patino-Martinez, and Ricardo Jimenez-Peris. WS-Replication: A Framework for Highly Available Web Services. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 357–366, New York, NY, USA, 2006. ACM Press.

[275] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in System Design. pages 195–206, 1988.

[276] P. Sanders and T. Worsch. *Parallele Programmierung mit MPI - ein Praktikum -*. Logos Verlag, Berlin, 1997.

[277] T. Sandholm and J. Gawor. Globus Toolkit 3 Core – A Grid Service Container. `http://www.globus.org/toolkit/3.0/ogsa/docs/gt3_core.pdf`, 2003.

[278] S. Sankaran, J. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, Winter 2005.

[279] A. Schiper. Group Communication: From Practice to Theory . In Jirí Wiedermann, Gerard Tel, Jaroslav Pokorný, Mária Bieliková, and Julius Stuller, editors, *SOFSEM*, volume 3831 of *Lecture Notes in Computer Science*, pages 117–136. Springer, 2006.

[280] R. Schlichting and F. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems*, 1(3):222–238, 1983.

[281] L. Schneidenbach, D. Böhme, and B. Schnor. Performance Issues of Synchronisation in the MPI-2 One-Sided Communication API. In *Proceedings of the 15th EuroPVM/MPI*, to appear 2008.

[282] F. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[283] E. Schnetter. The BBH Factory: Herding Simulations. Talk given to LSU relativity group in Baton Rouge, LA, October 2007.

[284] B. Schnor. *Scheduling and Migration Strategies for Parallel Applications on Distributed Systems*. Shaker Verlag, Aachen, 1999.

[285] B. Schnor, S. Petri, and M. Becker. Scalability of Multicast Based Synchronization Methods. In *Proceedings of the 24th EUROMICRO Conference, Västerås, Sweden, August 25-27, 1998*, pages 969–975. IEEE Computer Society, August 1998.

[286] J. Schopf, L. Pearlman, N. Miller, C. Kesselman, I. Foster, M. D'Arcy, and A. Chervenak. Monitoring the Grid with the Globus Toolkit MDS4. In *Journal of Physics: Conference Series – Proceedings of SciDAC*, 2006.

[287] B. Schroeder and G. Gibson. A Large-Scale Study of Failures in High-Performance Computing Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2006)*, 2006.

[288] B. Schroeder and G. Gibson. Understanding Failures in Petascale Computers. *Journal of Physics: Conference Series*, 78:012022 (11pp), 2007.

[289] Sun Grid Engine. `http://gridengine.sunsource.net/`, 2008.

[290] D. Simmel and T. Kielmann. An Architecture for Grid Checkpoint and Recovery Services. Grid Forum Document GFD.93, May 2007. Open Grid Forum.

[291] W. Smith, I. Foster, and V. Taylor. Predicting Application Run Times Using Historical Information. In *IPPS/SPDP '98: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 122–142, London, UK, 1998. Springer-Verlag.

[292] B. Sotomayor and L. Childers. *Globus Toolkit 4 – Programming Java Services*. Morgan Kaufmann Publishers, 2005.

[293] V. Springel. The Cosmological Simulation Code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, 364:1105, 2005.

[294] J. Staten, S. Yates, F. Gillett, W. Saleh, and R. Dines. Is Cloud Computing Ready for the Enterprise. *Forrester Research*, 2008.

[295] JSR 173: Streaming API for XML. `http://jcp.org/en/jsr/detail?id=173`, 2007.

[296] P. Stelling, I. Foster, C. Kesselman, C. Lee, and G. von Laszewski. A Fault Detection Service for Wide Area Distributed Computations. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, pages 268–278, Chicago, IL, 28-31 July 1998.

[297] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

[298] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. Technical report, May 2002.

[299] E. Stokes. Execution Environment and Basic Execution Service Model in OGSA Grids. Open Grid Forum Document GFD.119, 2007. Open Grid Forum.

[300] A. Streit, O. Wäldrich, P. Wieder, and W. Ziegler. On Scheduling in UNICORE - Extending the Web Services Agreement based Resource Management Framework. In Gerhard R. Joubert, Wolfgang E. Nagel, Frans J. Peters, Oscar G. Plata, P. Tirado, and Emilio L. Zapata, editors, *PARCO*, volume 33 of *John von Neumann Institute for Computing Series*, pages 57–64. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[301] Y. Sugita and Y. Okamoto. Replica-Exchange Molecular Dynamics Method for Protein Folding. In *Chemical Physics Letters*, volume 314, pages 141–151, 1999.

[302] A. Tanenbaum and M. van Steen. *Distributed Systems – Principles and Paradigms*. Pearson Prentice Hall, 2 edition, 2007.

[303] J. Taylor. e-Science Definition. `http://www.nesc.ac.uk/nesc/define.html`, 1999.

[304] EGEE – European Grid Infrastructure. http://public.eu-egee.org/, 2008.

[305] TeraGrid – American Grid Infrastructure. http://www.teragrid.org/, 2008.

[306] The Institute of Electrical and Electronics Engineers (IEEE). *IEEE 90: IEEE Standard Glossary of Software Engineering Terminology*. 1990.

[307] R. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.

[308] B. Toonen. MPIg Homepage. http://wiki.ngs.ac.uk/index.php?title=MPIg, 2008.

[309] Torque Resource Manager. http://www.clusterresources.com/pages/products/torque-resource-manager.php, 2008.

[310] P. Tröger, H. Rajic, A. Haas, and P. Domagalski. Standardization of an API for Distributed Resource Management Systems. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 619–626, Washington, DC, USA, 2007. IEEE Computer Society.

[311] S. Tuecke, I. Foster, and C. Kesselman. Open Grid Service Infrastructure. http://www-unix.globus.org/toolkit/draft-ggf-ogsi-gridservice-33_2003-06-27.pdf, 2003.

[312] S. Tuecke, V. Welch, D. Engert, L. Pearlman, and M. Thompson. RFC 3820: Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile. http://www.ietf.org/rfc/rfc3820.txt, 2004.

[313] M. Tuexen, Q. Xie, R. Stewart, M. Shore, J. Loughney, and A. Silverton. Architecture for Reliable Server Pooling. Technical report, March 2006.

[314] UDDI Version 3.0.2 – UDDI Spec Technical Committee Draft. http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm, 2004.

[315] Sathish S. Vadhiyar and Jack J. Dongarra. Self adaptivity in Grid computing. *Concurrency and Computation: Practice and Experience*, 17(2-4):235–257, 2005.

[316] R. van Renesse, K. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, 1996.

[317] S. Venugopal, R. Buyya, and K. Ramamohanarao. A Taxonomy of Data Grids for Distributed Data Sharing, Management, and Processing. *ACM Computing Surveys*, 38(1):3, 2006.

[318] W. Vogels. A Head in the Cloud - The Power of Infrastructure as a Service. Talk given at the Open Grid Forum 23, 2008.

[319] W3C. XML Encryption Syntax and Processing. URL: http://www.w3.org/TR/xmlenc-core/, 2002.

[320] W3C. XML-Signature Syntax and Processing. URL: http://www.w3.org/TR/xmldsig-core/, 2002.

[321] W3C. Web Services Addressing (WS-Addressing). http://www.w3.org/Submission/ws-addressing/, 2004.

[322] Web Service Notification (WSN) TC. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn, 2006.

[323] O. Weidner and J.-C. Bidal. Shrimp Farming on the Grid. In *MG '08: Proceedings of the 15th ACM Mardi Gras conference*, pages 1–1, New York, NY, USA, 2008. ACM.

[324] Globus Toolkit Version 4 Grid Security Infrastructure: A Standards Perspective. http://www-unix.globus.org/toolkit/docs/4.0/security/GT4-GSI-Overview.pdf, 2005.

[325] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for Grid Services. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, page 48, Washington, DC, USA, 2003. IEEE Computer Society.

[326] P. Wieder, O. Wäldrich, and W. Ziegler. A meta-scheduling service for co-allocating arbitrary types of resources. In *Proceedings of the 6th International Conference, Parallel Processing and Applied Mathematics, PPAM 2005,*, volume 3911 of *LNCS*, pages 782 – 791, Poznan, Poland, September 2005. Springer. Also published as CoreGRID Technical Report TR0010.

[327] Marianne Winslett. Bruce Lindsay Speaks Out. *SIGMOD Rec.*, 34(2):71–79, 2005.

[328] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, Champaign, Ill, 2002.

[329] R. Wolski. Dynamically forecasting network performance using the Network Weather Service. In *Cluster Computing*, pages 119–132, 1998.

[330] R. Wolski. EUCALYPTUS: An Open Source Service Infrastructure for Elastic Computing Research. In *Talk at Open Source Grid & Cluster Conference*, 2008.

[331] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, 1999.

[332] WS-I Basic Profile Version 1.1. http://www.ws-i.org/Profiles/BasicProfile-1.1.html, 2006.

[333] Web Services Resource Lifetime 1.2 (WS-ResourceLifetime). Committee specification, OASIS, 2006.

[334] Web Services Resource Properties 1.2 (WS-ResourceProperties). Committee specification, OASIS, 2006.

[335] Q. Xie, R. Stewart, M. Stillman, M. Tüxen, and A. Silverton. Endpoint Handlespace Redundancy Protocol (ENRP). Technical Report, Internet-Draft Version 17, IETF, RSerPool Working Group, September 2007.

[336] Y. Xie and Y. Teo. State Management Issues and Grid Services. In H. Jin, Y. Pan, N. Xiao, and J. Sun, editors, *GCC*, volume 3251 of *Lecture Notes in Computer Science*, pages 17–25. Springer, 2004.

[337] X. Zhang, F. Junqueira, M. Hiltunen, K. Marzullo, and R. Schlichting. Replicating Nondeterministic Services on Grid Environments. *15th IEEE International Symposium on High Performance Distributed Computing*, 0:105–116, 2006.

[338] X. Zhang, D. Zagorodnov, M. Hiltunen, K. Marzullo, and R. Schlichting. Fault-tolerant Grid Services using Primary-Backup: Feasibility and Performance. In *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 105–114, Washington, DC, USA, 2004. IEEE Computer Society.

[339] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *IEEE SCW*, pages 199–206. IEEE Computer Society, 2007.

[340] S. Zhou. LSF: Load Sharing in Large-scale Heterogenous Distributed Systems. Workshop on Cluster Computing, 1992.

Glossary

Advance Reservation	Contract, which commits a particular resource over a defined time interval to a resource consumer.
AIS	Application Information Service – core Migol service, which maintains meta-data about all running services.
ARS	Advance Reservation Service – Migol's service providing advance reservation support on top of the Globus Toolkit.
DRMS	Distributed Resource Management System – A DRMS, also referred to as meta-scheduler or Grid scheduler, provides an uniform acces to multiple Grid resources. The DRMS distributes jobs across multiple clusters in a Grid.
GRAM	Grid Resource Allocation Manager – Grid resource management system of the Globus Toolkit. The GRAM provide a common API for starting and monitoring of jobs across different local resource management systems. Different versions of the GRAM are currently supported, the GRAM2 (also referred to as pre-WS GRAM) and the GRAM4 (also referred to as WS GRAM).
GridFTP	High performance, secure data transfer service for wide-area networks.
GUID	Global Unique Identifier – ID used to unambiguously identity Grid Service Objects.
HPC	High Performance Computing – Computing paradigm addressing data- and compute-intensive applications running on high-end systems like compute clusters.
HTTP	Hypertext Transfer Protocol – Request/response protocol for transferring information on the Internet. HTTP is used for accessing Web applications as well as different kinds of Web services.
JBS	Job Broker Service – Migol's Grid scheduler: the JBS provides a single entry point to the Grid. It automatically discovers and selects resources for an application and starts the job.

JEE	Java Enterprise Edition – Platform specification consisting of several APIs defining the runtime environment for enterprise applications.
JNDI	Java Naming and Directory Interface – API specified within the Java Enterprise Edition (JEE) framework for accessing directory or naming services from within JAVA.
JSDL	Job Submission Description Language – Normative XML schema for expressing requirements of computational jobs for submission to resource management systems.
LDAP	Lightweight Directory Access Protocol – Protocol for updating and querying directory services. LDAP is e. g. used in the pre-WS MDS of the Globus Toolkit.
LRMS	Local Resource Management System – A local resource management system provides an uniform access to resources of a compute cluster. The main functionalities of a local resource management system are: job queuing, scheduling and execution.
MDS	Monitoring and Discovery Service – information service of the Globus Toolkit. Two version currently exist: the LDAP-based MDS 2 and the WSRF-based MDS4 (also referred to as WS MDS).
Migration	Process of transferring an application from one machine to another machine.
MRS	Monitoring Restart Service – Service for monitoring all Grid applications via a heartbeat protocol. In case of a failure, a restart is initiated by the MRS.
MS	Migration Service – Service supporting the automatic relocation of applications in case of a failure.
MTBF	Mean Time Between Failure – The average time between two consecutive system failures.
MTTF	Mean Time To Failure – The average time between the repair of a system and its failures.
MTTR	Mean Time To Repair – The average time need to recover a system from a failure.
NWS	Network Weather Service – System for monitoring network and operating system performance metrics in distributed systems.

OGF	Open Grid Forum – The OGF is an organization, which aims to provide a forum for grid innovations and the development of grid standards.
OGSA	Open Grid Service Architecture – Defines a blueprint for a standardized Grid architecture.
RFT	Reliable File Transfer Service – Web service for management of GridFTP transfers.
RS	Replication Service used within Migol to actively replicate the AIS.
SaaS	Software as a Service – Software delivery through the Web browser
SAGA	Simple API for Grid Applications – High-Level API for the development of grid-aware applications.
SLA	Service Level Agreement – Contract between service provider and consumer, which defines the terms under which a resource can be used.
SOA	Service-oriented Architecture – Loosely coupled architecture consisting of independent well-defined services.
SOAP	Platform and language independent protocol based on XML for exchanging Web service messages.
WSDL	Web Service Description Language – W3C Standard, which specifies an XML format for describing Web service interfaces.
WSRF	Web Service Resource Framework – OASIS Standard, which addresses the management of stateful resources using a Web service interface.

Index