Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Elżbieta Krępska
Nr albumu: 197882

A Service for Reliable Execution of Grid Applications

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr Janiny Mincer-Daszkiewicz
Instytut Informatyki

Lipiec 2006

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data   Podpis kierującego pracą

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data   Podpis autora pracy

Streszczenie

Computational grid is a service for sharing computational power and storage capacity over the Internet. It is a heterogeneous, vast, dynamic, distributed parallel computer. Due to its complexity and inherent unreliability, component failures happen frequently and applying fault-tolerance techniques becomes necessary. This thesis describes the Reliable Execution Service (RES), which reliably executes applications by detecting submission and execution errors, diagnoses them and recovers from them transparently to the user. The service does not execute applications directly but wraps them in an Executor, which prepares the execution, controls it (e.g. sends heartbeats) and handles the termination (e.g. detects exit code, intercepts uncaught exceptions). The service was realized using the Grid Application Toolkit – a simple, uniform, integrated grid API which provides independence of grid middleware as well as software and hardware diversity.

Słowa kluczowe

dependable computing, grid computing, fault tolerance, reliable execution, service, failure detection, failure diagnosis, failure classification, failure recovery

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D.4.5 [Operating systems]: Reliability – Fault-tolerance
H.3.4 [Information storage and retrieval]: Systems and Software – Distributed systems
C.4 [Computer Systems Organization]: Performance of systems – Fault tolerance
C.4 [Computer Systems Organization]: Performance of systems – Reliability, availability, and serviceability

Tytuł pracy w języku angielskim

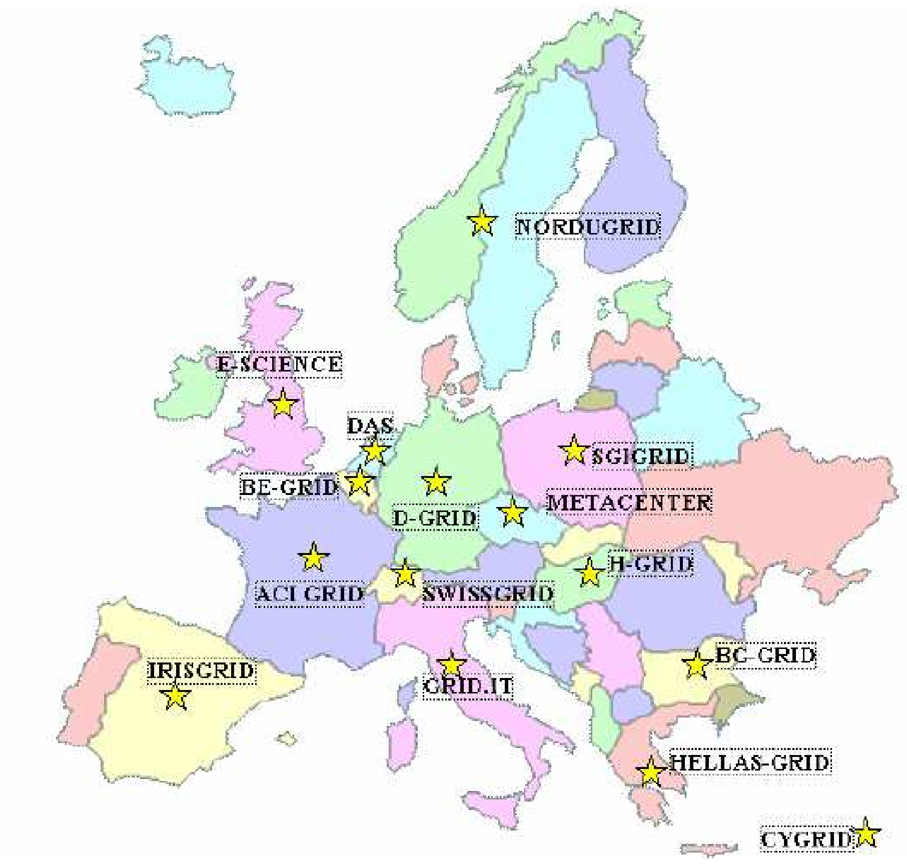
A Service for Reliable Execution of Grid Applications

Spis treści

Rozdział 1

Introduction

The CoreGRID "Network of Excellence" [10, 19] is a four year (2004-08) European project funded by the European Union's 6th Framework [18]. The project aims at advancing scientific and technological knowledge of grid and peer-to-peer technologies. These technologies have the potential to enable high-performance computing and efficient sharing of a set of widely distributed, connected to the Internet resources. In case of a computational grid the sharing is centralised, whereas in case of peer-to-peer networks – decentralised. However, programming such infrastructures is extremely chal-



Rysunek 1.1: Participants of CoreGRID Network of Excellence.

lenging and new programming paradigms, software tools and environments are required. Thus, the Network joins forty-two European research teams – major ones are depicted in the Fig. 1 – together to contribute to the area of grid and peer-to-peer technologies.

The Computer Systems Group at the Vrije Universiteit, Amsterdam, the Netherlands [42], participates in the CoreGRID virtual research Institute on "Grid Systems, Tools and Environments". The Institute aims at building a generic, component-based grid system that integrates lower level grid system/tools components and higher level application components [11, 12].

Naturally, to enable communication between the grid system and applications, an in-between layer exposing grid system capabilities to applications is needed – the *Mediator Components* [13] layer. Two of these mediating components will be of particular importance for this thesis: the *runtime environment* component – integrating the major grid system capabilities into a single, simple, uniform API[1] – and the *application manager* – dependably executing users' applications.

A grid environment is a heterogeneous, vast, dynamic and inherently unreliable parallel machine. Failures of its components can never be fully eliminated by applying traditional software engineering techniques. Thus, failures must be anticipated, they must not prevent applications from successful completion – we say that the execution must be made *fault-tolerant*. Grid middleware can intercept execution failures and, if necessary, apply recovery scenarios. Software diversity will be crucial here to achieve fault-tolerant execution and it can be provided by taking advantage of the runtime environment. Another problem is dealing with grid middleware failures by a non-expert developer. It is so complex and requiring extensive knowledge in the field of grid computing that it substantially hinders solving the real scientific problem. Instead of log files containing low-level details from all layers of execution, the real cause of the failure should be determined and presented to the user.

The main objective of this thesis will be to perform research on the reliable execution of applications in an unfavourable distributed environment such as a computational grid. Basing on the research results, a persistently running service, the *application manager*, guaranteeing application completion – in spite of occurrence of transient errors and without user involvement – should be proposed. The design's correctness should be verified by implementing the framework and evaluating it on the distributed computer DAS-2. It would also be beneficial to demonstrate that high-level grid API can be used not only in client applications but also to build a grid service.

The remainder of this thesis is structured as follows. Chapter 2 introduces the notion of a computational grid, focusing on its complexity and unreliability – main factors motivating applying fault-tolerance techniques in grid software. We review current approaches to reliable execution of applications on the grid and discuss numerous projects and work done related to this field. Chapter 3 further investigates the Mediator Components architecture, exploring the grid runtime environment concept and discussing its existing implementations. Chapter 4 states in detail the requirements that the solution framework must comply with. It presents the Reliable Execution Service (RES) design with the emphasis on the failure detection scheme and the way that applications are executed. Chapter 5 evaluates the designed framework. We perform multiple tests against correct detection and handling of user-related and grid-related failures. We demonstrate that the service can be successfully used in real-life applications and we use one such an application to examine how the service might affect performance of the execution of an application. Chapter 6 discusses the service limitations and possible future directions and concludes the thesis. Two appendices, A and B, describe what the thesis bundle contains and explain how to administrate and use the Reliable Execution Service.

[1]Application Programming Interface (API) – a set of exported system routines

Rozdział 2

Fault-tolerance in grid computing

This chapter introduces the notion of a grid (Section 2.1), outlining its complexity and unreliability which are reasons why applying fault-tolerance (in short: FT) techniques is necessary (Section 2.2). We analyse failures in the grid environment – what they are, how they occur – and give real-life examples of failures of applications (Section 2.3). Next, we discuss what can be done about failures – we review current approaches to reliable execution of applications in an unreliable environment and present numerous projects and work done related to this field (Section 2.4).

2.1. Grid environment

2.1.1. What is a grid?

A grid is a service for sharing computer power and data storage capacity over the Internet. Its ultimate objective is the creation of one vast computational resource [8]. A grid was defined by Ian Foster [22] as a computer system with three following characteristics:

1. There is no centralised control over coordinated resources.
2. Only standardised and open protocols and interfaces are used.
3. Non-trivial qualities of service are delivered.

Grids are primarily intended to be used by scientists to solve large long-computation and/or data-intensive problems, to enable collaborative engineering or computational steering. Throughout its evolution also new fancy uses of grids developed, for example teleconferencing, web servers load balancing, treating a grid as a gigantic storage for enormous amounts of data produces by quantum physics experiments such as currently built CERN's Large Hadron Collider [49].

2.1.2. Grid complexity

The grid is composed of numerous types of components, such as:

- Computational resources, e.g devoted cluster machines, workstations, volunteer processors[1].
- Storage resources, e.g. databases, network filesystems, ontologies.

[1]Volunteer of good-will computing – computing method that emerged in recent years: it exploits Internet-connected computers, volunteered by their owners, as a source of computing power and storage [52].

- Mobile devices, e.g. portable media players, mobile phones, RFID[2] devices.
- Services, e.g. replica catalogues, information, execution, monitoring, logging services.
- Scientific instruments, e.g sensors, microscopes, telescopes, visualisation hardware such as regular displays, IC-Wall [54] or CAVE [53].
- Software libraries, e.g. graphic libraries, language implementations, compilers.
- Processes, e.g. daemons, running applications.
- People, e.g. in the certificate issuing process.

The number of nodes in a grid might be stable or highly variable. For example DAS-2 [16] is a homogeneous (i.e. all machines are of the same type) grid consisting of 5 clusters, 200 processors in total, located in the Netherlands. Another example is the LHC Computing Grid (LCG) [49] which only today consists of between 32631 and 60172 processors, in 186 sites in 43 countries.

2.1.3. Unreliability in a grid environment

A grid environment is unreliable for many reasons, including:

Geographical dispersion. A grid often consists of clusters located in different geographical parts of the world. Therefore, for communication between clusters we expect high latencies and the bandwidth varying often and in an unpredictable manner.

Heavy network communication. Grid software is a set of distributed components or services interacting with each other by the means of network links. In fact, each non-trivial grid operation involves network activity, which substantially increases grid operations instability.

Heterogeneity, complexity, dynamism. Grids consist of numerous nodes of various types and the set of available resources changes dynamically. Component failure probability is high.

Development difficulty. Grid software and applications development is difficult, and therefore more prone to bugs.

Administrative and security difficulties. Grid environment involves multiple autonomous administrative domains, each with its own policies. Security and trust requirements are exacerbated. Grid environment maintenance is a laborious task.

2.2. Motivation for fault tolerance on a grid

While making life easier for scientists might be a beautiful dream, we need to face tough reality. As stated in the previous sections, a grid environment is an inherently complex, fluctuating and unreliable system what leads to frequent component failures. Hence, it is widely recognised that applying fault-tolerance techniques in grid software is necessary.

[2]Radio Frequency Identification (RFID) – an automatic object identification method: an RFID tag is identified by an RFID reader using radio waves.

How often do failures happen on a grid? In a simplified model we assume that a machine fails on average once every N day, e.g. $N = 16$. A program run on m machines for d days succeeds then with the probability:

$$p_f(N, m, d) = (1 - \frac{1}{N})^{d \cdot m}$$

For a program running for one day on one machine the chance for success is 94%. But for the same program run on 10 machines the success probability is 52% – it almost halved!

What typically happens in case an application fails running on a traditional computer? A failure of a component the application depends on, is considered rare and exceptional – operating software usually limits its actions to detecting the problem, informing the user about it and aborting the execution.

What typically happens in case an application fails running on a grid? The developer hopes to find proper log files. If they are available, it is up to the programmer to read them, find the root cause of the problem and finally distinguish between middleware and application specific failure. The developer has to know about possible failure, about all middleware layers, error codes and exceptions in multiple programming languages, cluster hardware and many more. A non-expert user would just have no idea what to do.

What can be done about it? Fault-tolerance techniques should be implemented, to allow an application to continue in spite of transient failures happening in the unfavourable environment. Grid middleware can intercept execution failures and, if necessary, apply recovery scenarios. It should shield users from the nasty business of grid middleware level failures and let them concentrate on real scientific problems they solve.

2.3. Failures on a grid

Because a grid is a complex system consisting of numerous layers, the anticipated failures vary strongly – from hardware failures, through middleware failures, up to application failures. We begin this section with a presentation of generally accepted fault-tolerance terminology. Then we analyse why and how grid components and applications fail. Finally we give real-life examples of grid application failures.

2.3.1. Terminology

Generally accepted fault terminology is that given by Avizienis and Laprie [4]. A *fault* is a violation of the system's underlying assumptions, internal or external of the system. An *error* is an internal data state that reflects the fault. The fault might, but does not have to, cause an error. A *failure of a system* is an externally visible deviation from its specifications caused by an error. An error might not cause any failure or cause only *partial failure*, e.g. slower or limited service.

For example, a random cosmic ray that passes through a storage and corrupts some data is a fault. If this data was in use, it creates an error. If the error changes the result of an executing algorithm, which uses this data, the algorithm experiences a failure.

A *service failure* is a transition from a *correct service*, i.e. delivering specified functionality, to an incorrect service, i.e. delivering functionality that deviates from expected behaviour. From the service user point of view, the service has failed when it does not respond (is down or is too busy) or misbehaves (responds with incorrect actions or data). The failure might be *permanent* or *transient*.

Fault tolerance (FT) is a mean to attain *dependability* of the service. It aims at the avoidance of failures in the presence of faults. A fault tolerant service detects errors and recovers from them without participation of any external agents, such as humans. Strategies to recover from errors include *roll-back*, i.e. bringing the system to a correct state saved before the error occurred, *roll-forward*, i.e. bringing the system to a fresh state without errors, or *compensation*, i.e. masking an error, in situations when the system contains enough redundancy to do that.

A *failure detection service* is a distributed oracle (a monitor) aiming at providing some distributed (notifiable) objects with hints about the crash of other (monitorable) objects.

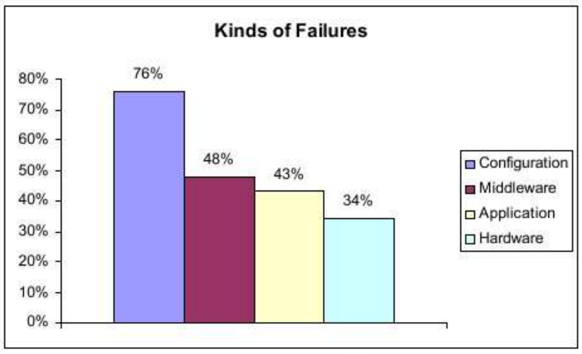
2.3.2. Why do grid components fail?

There are various reasons for failures on a grid, such as:

- Hardware crashes, e.g. permanent defects in machine's memory or disk, temporary CPU overload.
- Software crashes, e.g. memory leaks, synchronisation bugs leading to deadlocks.
- Machine downtime due to its maintenance, rebooting, local policy disconnection.
- Network congestion, outage or partition. Power outage.

2.3.3. Why do grid applications fail?

Madeiros et al. [35] conducted a survey in the grid community with the following result. The most frequent grid applications failure cause is faulty configuration – it was pointed out by 76% or respondents. Other are middleware failures – 48%, application failures – 43% and hardware failures – 34%.



Rysunek 2.1: Types of failures pointed by grid users [35].

Kola, Kosar and Livny [34] analysed logs of long runs of four real-life data intensive production applications. They determined failures that occur most frequently:

1. Intermittent wide-area network outages.
2. Data transfer failures:
   (a) Hanging indefinitely (usually due to missing acknowledgements in FTP transfers)
   (b) Data corruption (usually due to faulty hardware)
   (c) Storage server crash (due to too many concurrent write data transfers)

3. Outages caused by machine crashes and downtime for hardware/software upgrades and bug fixes.

4. Misbehaving machines caused by misconfiguration and/or buggy software.

5. Insufficient disk space for staging-in input files or for writing out output files.

2.3.4. Examples of grid application failures

Beside failures that grid middleware is responsible for, there are numerous "user-related" grid application failures to consider. In this section we give examples of such real-life applications, their failures and expected recovery schemes:

1. *Linear solver.* If the execution of a linear solver is not completed within 30 minutes, it means that the solver did not converge. The algorithm should be restarted.

2. *Successive over-relaxation*[3]. If the program executed its loop too many times, the algorithm did not converge for specified data. The application should inform the user about the fatal problem and stop the execution.

3. *Weather simulation.* The program failed to write out its temporary files because there was not enough disk space. The simulation should be restarted on a host with more disk space available.

4. *A tree search algorithm with checkpointing*. The algorithm executes in an unreliable environment, for example on a volunteer machine, which can reboot at any time. The program periodically persistently saves its current state, i.e. a path in the search tree, to a separate file (the program checkpoints). If the resource crashes, and the application is restarted, computation resumes not from scratch but from the last correct checkpoint. The application might also be migrated to another machine. The service takes care of staging checkpoint files to a new host.

5. *Genetic algorithm.* A genetic algorithm runs on a grid in such a way that each machine explores a small piece of the search space at a time. Provided that the number of pieces is large enough, if some machine fails and gives no result, the application can just ignore this fact.

6. *Segmentation fault.* A C-application fails with a "Segmentation fault" and non-success exit code, e.g. 137. The exit code and the output of the application should be reported to the user.

2.4. Fault tolerant application execution on a grid

2.4.1. Execution on a grid

How is the execution on a grid performed we explain using the DAS-2 distributed computer (see Section 5.1) as an example. The organisation of this machine is quite simple, for more generic framework the reader may consult the OGSA Execution Management Services [25].

On DAS-2 a user can execute a job on one of five clusters. One possibility to perform this operation is to submit it to one of five resource brokers (e.g. GRAM [46]) which bridge between demanded and available resources. The resource broker interacts with the local scheduler (e.g. SGE [56]) to actually

[3]Successive Over-Relaxation (SOR) – an iterative numerical method of solving a linear system of equations $\mathbf{A}x = \mathbf{b}$. Often used in physics.

orchestrate the delivery of computational power to the job. Before the job is started, required input files must be pre-staged (copied to the execution host). During the execution the resource broker can be queried or can notify the client about the job status. After the job is done output files must be post-staged to the user.

2.4.2. Execution failures and what can be done about them?

Even a simplified job submission scenario presented above is a process that goes through multiple grid layers. Failures might happen during many stages of that process:

1. *Execution services failure.* Services executing the application such as grid job manager (a component scheduling between resource brokers), a resource broker or a local scheduler might misbehave/crash or might not be able to satisfy job requirements.

2. *Local environment failure.* Application might fail due to local execution environment failure, such as machine rebooting, virtual machine crash or using all available disk space.

3. *Resource failure.* A resource that the application depends on might fail, for instance the network might be out, web-service not responding or data transfer hanging.

4. *Application specific failure.* The application might fail due to a bug in its code, for example performing illegal operation leading to segmentation fault, numerical error or causing thread deadlock.

Failure (1) of a resource broker or a scheduler might be masked by trying to start a job on a different site. A failure of grid job manager can be circumvented by submitting directly to a cluster, provided we can discover hosts belonging to a grid. Failure (2) can be masked by restarting a job on a different host and the choice of this host might be made with support of artificial intelligence techniques and heuristics. To discover failure type (3) a special failure detection and monitoring services are used. There is broad literature in this area [39, 15, 29, 21, 31]. Only failures of type (4) should not be masked, and feedback about their occurrence should be provided to the user.

2.4.3. Current approaches to reliable execution on grids

Currently several different solutions to reliable execution in an unreliable environment dominate: an application might ignore FT, implement failure detection and response behaviour completely within the application or use external failure detection services.

Application built-in FT. This type of fault tolerance is built directly into the application code. Fault detection and recovery can be performed efficiently, as the application and its algorithms specific structure and characteristics can be exploited, for example in Fault-Tolerant MPI [20], or even switched off to speed up the execution, for example for naturally fault-tolerant genetic algorithms.

Since the primary purpose of grids is to provide computational power to scientists, the intended application developer is a domain expert, typically a physicist, a biologist or a meteorologist, not necessarily a grid expert. Application built-in fault-tolerance imposes an undue burden on the programmer, who should in this case be knowledgeable about grid technologies and aware of all the different types of failures that can happen on a grid. The scientist cannot fully concentrate on the real problem to solve. Implementing FT techniques increases application complexity and time needed to develop it. Medeiros [35] reports that 57% of users complain that they are highly involved in diagnosing failures compared to only 19% users reporting that the the environment they use supports fault-tolerant scheduling.

Moreover, it is often unclear how to build fault-tolerance on the application level. The developer would have to come up with certain timeouts and thresholds which depend on end-system, network characteristics, current grid load, things not known at the job submission time.

Multi layered FT. In many multi-layered systems, each layer might handle its scope errors and pass the rest up [40]. This approach reduces higher-level layers complexity but hurts the performance of higher layers that can use this error information to adapt [34]. Building FT into each component is also not practical – no only do grids often use third party software but also (re)engineering software in all clusters (numerous administrative domains) would not be welcomed by these clusters' administrators and users. Also, due to new hardware or software installation new types of failures can emerge [40].

External specialised FT services. The environment might offer an automatic failure detection middleware-level service which basically allows building more intelligent fault-tolerant services. As developer has no impact on this service [39, 15], it is not flexible and might hurt application performance. Using such a service still requires grid-related knowledge and effort from the developer. Moreover, such a service usually needs to be installed on each machine and therefore hinders the grid's maintenance.

2.4.4. Work related to reliable execution on a grid

Fault tolerant middleware. Kola, Kosar and Livny [34] discuss Phoenix, a grid middleware layer which handles grid errors transparently to the user. The system detects failures by scanning computation scheduler and data placement scheduler log files. It uses special database to persistently store the history of transfers and computation and uses stored information to set appropriate transfer timeouts and retry thresholds in order to provide adaptive hung transfer detection. It also diagnoses data placement and data integrity failures in a sophisticated fashion. The data placement agent determines the reason of the failure by examining the URL, distinguishing between problems with: DNS, network, host, protocol, authentication, file permissions, etc. The system incorporates a simple stateless failure manager which comes up with strategies to handle failures. Users can influence the decisions by specifying policies. All encountered errors are persistently logged what enables detecting misbehaving machines, self-adapting of grid middleware components and preventing the information loss. However, scanning log files requires services to log information in a specific fashion and delays reaction to the failure. The system nicely detects data placement and movement failures but ignores all the rest.

Workflow failure detection. Hwang and Kesselman [30] propose a workflow-level failure handling framework for the Grid. An application consists of a set of tasks structured as a workflow, i.e. directed, acyclic graph with nodes representing tasks and edges denoting dependencies between nodes. Using the graph, one can express high-level fault-tolerance schemes, for example alternative task (if the task A fails, run task B) or redundant tasks techniques. The framework implements also reliable task execution by using techniques such as retrying, replication and checkpointing. However, the service bases its reactions only on resource broker feedback, which is not always accurate. Introducing workflows can help nicely in expressing fault-tolerance interdependencies between multiple tasks.

NASA. Smith and Hu [38] implemented an execution service for NASA Information Power Grid [51] which reliably executes complex jobs in a distributed environment. Each job consists of many tasks along with dependencies between these tasks. The dependencies mechanism enables the user to specify which task to execute when another task fails or is cancelled by the user. There are also predefined data movement tasks such as directory creation or file deletion. Interestingly, instead of executing the application on the grid, IPG-ES executes a script which prepares execution environment, runs the application and properly detects its exit code. However, for executing a job, the user has to specify where to run it. Also, the exit code is the only way to inform the service about application failure, what is not natural for Java applications.

GRAM and Condor-G. The Grid Resource Allocation and Management component is the execution service of Globus [46], widely used, state-of-the-art standard, grid software. The GRAM protocol lacks fault-tolerance support. It returns its own exit code specifying the type of failure encountered but does not detect application's exit code and fails to prepare environment variables for the execution [38]. If submission or execution fails, the result is an execution log containing execution status code. The only way of acquiring information about executing jobs is being notified by the service. Condor-G project uses the GRAM protocol and provides additionally higher-level fault tolerance – it detects and handles resources crash to provide "exactly once" execution semantics. However, Condor-G is "all or nothing" solution, i.e. the reliable execution part cannot be used standalone.

Reliable execution service proposals. Kielmann et al. [33] propose a design of problem solving environments (PSE) system for Science Experimental Grid Laboratory (SEGL) using Mediator Components Toolkit [13]. Bubak, Szepieniec and Radecki [7] point out the need of failure detection and recovery service in a grid and propose a concept similar to [13]. The Fault Tolerant Manager supervises the execution of jobs and in case of a failure a Decision Maker decides on a recovery scenario. The proposal counts a dramatical decrease in the application's performance as a failure.

Component failure detection services. Stelling, Foster et al. [39] propose the Heartbeat Monitor (HBM), an unreliable, low-level fault detection service for a distributed system. The service bases on unreliable fault detectors [9] which try to discover which system components (machines or processes) *might* have failed and notify the application of that fact. Whether to trust this information, how to interpret it, what to do about it, is left entirely to the application.

Défago et al. [15] improves performance and scalability of this scheme by introducing hierarchisation and gossipping between monitors. It also introduces an interesting concept of a service watchdog – a separate process tied to a failure detector: should the detector fail, its watchdog would restart it, should the watchdog die, its failure detector would restart it.

However, these services are lower-level, they only inform that they suspect that a component has crashed. The application should decide whether and how to recover from the failure what can be non-trivial, especially that the heartbeat monitors do not know the failure root cause.

Interestingly, Hwang and Kesselman in [29] showed how they equipped their flexible failure handling framework [30] with a generic failure detection service. This failure detection service bases also on heartbeats and notifications but can distinguish between task crashes and tasks failure requests. If a resource essential to the application execution fails, the failure handling framework can be notified and proper failure masking schemes can be applied. These two services coupled create a versatile fault-tolerant application execution environment.

2.5. Summary

We introduced the notion of a grid with emphasis on application and component failures in this unfavourable distributed environment. We argued that applying fault-tolerance techniques is indispensable to make the user able to easily develop and reliably execute applications on a grid. Finally, we presented broad literature and numerous projects related to these issues.

Rozdział 3

Mediator components

The Mediator Components [13] system is part of the CoreGRID's design of a generic grid platform. When building a grid system out of smaller, independent components, we naturally distinguish between the lower level system components and higher level application components. Mediator components mediate between them, the most important components are the integrated runtime environment which exposes system capabilities in a simple, uniform grid programming API and the Application Manager which controls the execution of applications. The mediator components architecture and the Application Manager in particular are the conceptual "big picture" for the framework designed in this thesis. The envisioned generic grid structure is presented in the Fig. 3.1 and explained in the following section.

3.1. Mediator components framework

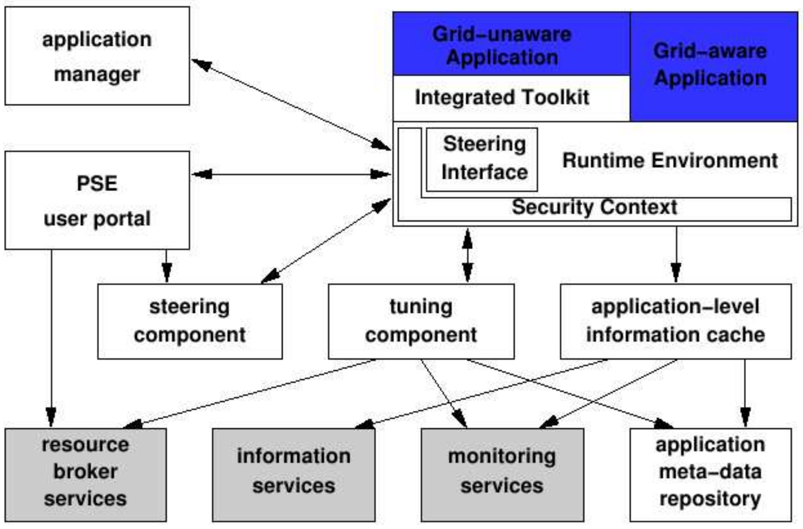
Runtime environment. The runtime environment API constitutes an uniform and integrated interface to various grid services and resources, such as job schedulers, resource brokers, data storage services, etc. The runtime component delegates its API invocations to selected services in the grid environment. Thus it delivers an abstraction layer between application components and system components. GAT [1] and SAGA [28] are examples of such grid "runtime systems".

Grid applications. Grid applications use the underlying grid system through the runtime environment API. Grid unaware applications use an additional Integrated Toolkit indirection level to leave out all grid-related aspects from their code. Each activated application is bound to its security context.

Application manager. The application manager establishes a pro-active agent on behalf of the user, in charge of tracking an application execution, from submission to successful completion. It guarantees masking application-independent temporary error conditions occurring during execution on the grid. To perform this task, the application manager needs to communicate with the application (via the steering interface), meta-data repository and cache, and steering components.

Steering and tuning. Components and executed applications implement a dedicated steering interface, which makes them manageable from other components, services, pro-active threads of control or the user. Steering is useful in contexts such as debugging, performance tuning, job activity or status control.

Data repositories. Application-level meta-data repository stores persistent data concerning an application and its runs. Application-level information cache delivers uniform interface to retrieve meta-data from all kinds of repositories, such as monitoring system, application-level meta-data, user databases, etc. Its additional goal is to cache those information, speeding up access to them.



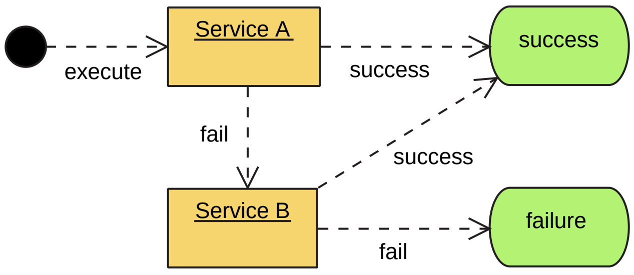
Rysunek 3.1: Generic grid mediator-components design. Blue boxes indicate application code, white boxes indicate CoreGRID deliverables, grey boxes – third party components.

3.2. Runtime environments

The concept of providing simple, integrated grid API and services diversity to applications seems to be the future of high-level grid computing. The implementations are quite recent though. In this section we discuss work related to this field, in particular the Grid Application Toolkit – an implementation of such a runtime environment that will actually be used in the solution framework.

Meta-services. Anderson et al. in [43] discuss dependability of grid services. They point out the need to use diverse, redundant resources to reduce the probability of operation failures and enable the system to continue in spite of node failures. They argument that for common operations the pool of different implementations could be large and it is likely that the implementations would differ significantly between each other. Service composition presented in the Fig. 3.2 is suggested. If service A succeeds, the operation succeeds. If A fails, B is invoked. If service B succeeds, the operation succeeds and if B fails, the whole operation fails.

Meta-containers service. Dobson, Hall and Sommerville [17] came up with a similar concept of fault-tolerant meta-containers for services. They build a service which is capable of integrating several services implementing the same functionality but possibly using diverse methods – in different programming languages, deployed on different hosts, using different hardware. Replica services are



Rysunek 3.2: Service composition.

invoked in parallel, results compared, composed and returned to the caller. The service imposes additional delays (at least doubling the time) on each invocation but the benefit is that replica services might be discovered dynamically and the service is independent of a client language binding.

Grid Application Toolkit. The Grid Application Toolkit (GAT) [1, 2, 37, 41] is a part of the GridLab [47] project. It is a grid system high-level "meta" interface. It aggregates available grid services in one uniform interface on the client-side and delegates its calls to appropriate services. A call fails if and only if all replica services fail. There are C++, Python and Java implementations of the Grid Application Toolkit. A Java implementation is called JavaGAT [48], was developed at the Vrije Universiteit and is used to implement the framework designed in this thesis. The main GAT component is the *GAT Engine* delegating calls such as submitJob or copy to *GAT Adaptors* implementing functions/operations bindings, for example resource brokers adaptors: LocalAdaptor, SSHAdaptor or GlobusAdaptor. The engine picks from adaptors implementing specified functionality at a run-time, which means that the adaptors can be loaded dynamically.

The Grid Application Toolkit evolved into the Simple API For Grid Applications (SAGA) [28] which is currently being worked on.

3.3. Summary

We presented the Mediator Components architecture, the conceptual "big picture" for the reliable execution framework. We emphasised the importance of the grid runtime environment, a simple uniform interface to the grid system, integrating many lower level services into one consistent API. Such a runtime environment – the Grid Application Toolkit – was designed in the GridLab project, implemented at the Vrije Universiteit and will be used as the basis for the solution framework implementation in this thesis.

Rozdział 4

Solution framework

This chapter describes the solution framework designed according to the issues discussed in previous chapters. First we state requirements that the framework should comply with (Section 4.1). Next we present the framework design (Section 4.2) with the emphasis on the failure management scheme (Sections 4.2.3 – 4.2.5). We describe the framework implementation (Section 4.3) and the concept of the job Executor (Sections 4.2.2 and 4.3.3). Finally we discuss several implementation issues which turned out to be problematic and involved reengineering of some grid middleware (Section 4.4).

4.1. Framework objectives

The Reliable Execution Service (RES) is a permanent service providing reliable execution of applications submitted to the grid. Permanence means that the service runs all the time. Reliability means handling grid-middleware transient failures transparently to the end user. The user should observe only the application errors, with perhaps the exception of permanent grid environment failures.

The service will be designed to meet the following requirements:

1. **Application feedback.** In an ideal case, there should not be any requirements posed on the executed application. However, having application feedback enhances greatly failure detection capabilities and enables application specific debugging and logging. Therefore, for both service-aware and unaware applications, the reliable execution functionality should be offered. Additionally, an application can talk to the service, for example requesting an application-specific failure or sending a message to the user.

2. **Failure detection.** The service should detect submission and execution problems. After the execution is done it should properly detect the exit code and intercept uncaught exceptions whenever possible. If a monitoring service is available on a grid, the service might monitor hosts executing the application, the application process or resources that the application depends on.

3. **Failure handling and recovery.** The service should offer diverse failure handling strategies and the user should be able to change or refine them (see Section (4)). If the submitted application is capable of checkpointing, the service should take advantage of this fact and, in case of a failure, restarting should be performed from the last correct checkpoint. If the failure is permanent the user should be informed about the failure root cause but shielded from all unnecessary details.

4. **User influence.** The user might influence failure detection and recovery schemes by specifying policies. Policies should be separated from the application code. They might describe speci-

fic failure detection requirements (e.g. specifying the maximum application execution time or which applications signals are fatal) and application-specific failure handling strategies.

5. **Workflow-level and task-level fault-tolerance techniques.** The service will not implement task-level fault tolerance techniques, such as alternative or redundant tasks (see Section 2.4.4). However, the service's direct API should allow a developer to implement those techniques easily.

6. **User decoupling from the application.** After submitting a job, the user can decouple and access the historical record of the job at a later time. Job history is kept during and after job completion. The user should not have to examine output files to check the application status or to see if it exited correctly.

7. **Avoidance of administrative problems.** The service should be designed such, that special privileges, accounts or certificates are not necessary. The service should use the security framework deployed on the grid, and require users to delegate their rights for the service to act on their behalf. Also, there should be no need to install anything on each machine in the grid.

8. **Preventing information loss.** Masked grid component failures should be transparent to the user but not to grid middleware or to the administrator. Records of failure occurrences should be stored persistently, to allow sophisticated middleware components to adapt their behaviour based on this information and to prevent a situation when a permanent resource failure goes unnoticed by the administrator for a long time.

9. **Dependability.** The service should be able to sustain a crash of its hosting machine. When the machine is back on, the service should recover and re-acquire jobs activated before the crash.

10. **Portability.** The service should be independent of the specific middleware installed on the grid, for example Globus, ProActive or SSH, the hosting machine platform, the language-binding of an executed application and the environment of targeted machines. The service acts as a grid meta-scheduler (is able to choose a cluster to run a job), so it has to encapsulate specific grid information at least on clusters belonging to the grid. The encapsulation should be clearly separated from the service code and it should be possible to discover this information dynamically.

4.1.1. Job model

The service is intended for computation intensive, long running applications (long here means for example more than an hour). An application is defined according to POSIX standard:

a. executable location,

b. execution arguments,

c. environmental variables,

d. names for standard streams: stdout, stderr, stdin.

The staging model acquired is very simple – application specifies additionally:

e. a list of files that should be pre-staged,

f. a list of files that should be post-staged.

Additionally, for checkpointable applications – application specification contains:

g. a list of files containing checkpoints,

h. checkpoint files repository location.

4.2. Framework design

The service architecture is presented in the Fig. 4.1. Selected important aspects of service workings are described in more detail in subsequent sections.

The most important thread of control in the service is the JobController. It handles users' requests and controls currently activated jobs. Waiting jobs are submitted via the Grid Application Toolkit, the GATEngine, to the grid. Executing jobs are checked for failures or a success. The controller also adapts and optimises its behaviour using the SelfAdaptationAgent.

Job's Executors are threads of control active on grid machines. They supervise the execution, give feedback to the service and mediate communication between the service and the application. In case of a failure, the FailureRecoveryAgent is activated to handle the problem.

4.2.1. The job lifecycle

After the user submits a job to the service, the JobController is managing its execution. Job description and part of execution information is persistently saved to the JobRepository. When the job is to be submitted to the grid, the ExecutionPlanner first plans the execution, specifying for instance on which cluster the job should run. Afterwards, the GATWrapper converts RES job description to a job description understood by the GATEngine. Eventually, the JobController submits the job to the GATEngine.

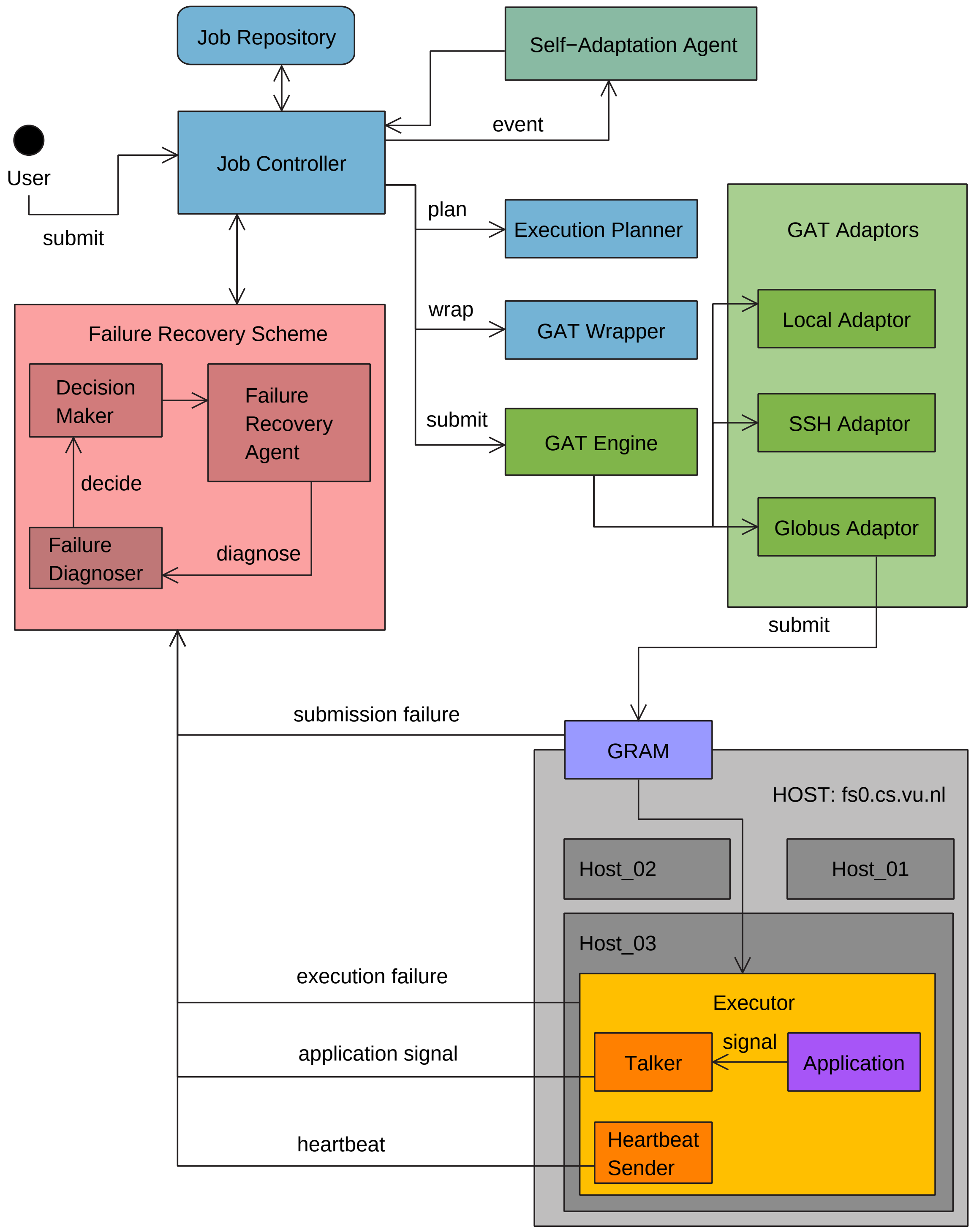
The GATEngine tries to submit the job using installed resource brokerage adaptors. Example invocation could look as follows: First, the GATEngine tries the LocalAdaptor and fails, because the job is to be run on a remote machine. Next, it tries the SSHAdaptor and fails because the SSH daemon is not running on the remote machine. At last it tries the GlobusAdaptor and succeeds. The adaptor contacts the GRAM service at the target cluster and submits the job. The job waits in the local scheduler queue. When computational resources are available, the Executor is started.

The Executor initialises a signal sending mechanism, runs the HeartbeatSender and the Talker and sends the service a JobStart signal. The application might use a language-dependent API to contact the Talker. Through the Talker the application can send the service/user signals, interpreted according to the policy specified by the user, for example as debugging signals.

After the job is done, the Executor detects exit code and uncaught exceptions, sends a JobDone signal to the service and exits. The service waits until the GATEngine reports the job to be done and verifies the execution. The job is done now.

4.2.2. Wrapping

As explained above, applications are not executed directly but they are wrapped in the Executor which locally controls the execution. A wrapper solution was proposed earlier, for instance in [40] to circumvent problems with the incorrect exit values in the JVM implementation and in [38] to circumvent problems with the execution environment incorrectly prepared by the resource broker. In RES the Executor is responsible for more tasks:



Rysunek 4.1: The Reliable Execution Service architecture. The black circle denotes the service user, grey boxes represent hardware installed and middleware deployed on the grid and remaining boxes represent the Reliable Execution Service, singling out its functional modules. Red boxes are components responsible for failure handling, blue boxes – for job submission and control, green – the GAT library, yellow – the job executor, pink – the application.

1. *Gaining control over the application execution.* The wrapper signals to the service when the execution begins, runs the application and after it is done, reports this fact to the service.

2. *Local monitoring of the application execution.* During the execution the wrapper sends heartbeats to the service. When the job is done, the Executor detects its exit code and intercepts uncaught exceptions if possible.

3. *Talking to the application.* The Executor mediates the communication between the application and the service. The application might send signals, which are interpreted by the JobController according to the user's policy. The signal might be fatal, i.e. describing a failure which occurred during the execution, or informative, for example debugging and logging signals.

The information received from the Executor might be more accurate and more timely than information from the resource broker which does not control the application directly. Also this is the only way to correctly detect and propagate the exit code of the application.

4.2.3. Failure handling scheme

The FailureRecoveryAgent is responsible for handling failures after they are detected – see Fig. 4.1. The scheme consists of following steps:

Failure detection. The service keeps track of the application by "external" and "internal" observations of the job. External – when the service queries the GAT Job object state, i.e. indirectly querying the resource broker, and catching and examining GAT exceptions and error codes. Internal – when the observation is done using the Executor mechanism on the machine executing the application, for instance intercepting application exceptions, detecting its exit code and talking to the application (see Section 4.2.2). In case of a failure, the FailureRecoveryAgent is activated.

Failure Diagnosis. The FailureRecoveryAgent activates the FailureDiagnoser to diagnose the failure root cause. The FailureDiagnoser bases on information gathered by the JobController but it can also acquire additional information, for example simply by pinging hosts using ICMP protocol or by invoking sophisticated monitoring services.

Failure classification and recovery. The DecisionMaker classifies failures into permanent and transient (lasting only for a short time). Permanent failures are reported to the user in the way that shields the user from middleware and hardware details. Transient failures are handled with accordance to the user's policy. The DecisionMaker comes up with a suitable RecoveryScenario for the job. The scenario is executed by the JobController.

4.2.4. Failures detected by the service

The most important detected failures, related to the fact that the application executes in the grid environment, are as follows:

Failures prior to the execution. Before the execution on the target machine takes place at all, numerous types of failures must be anticipated, including:

- Pre-staging failure, e.g when files to pre-stage do not exist, the user has no permissions to read them or they are locked by another process.
- Incorrect job description, e.g. an incorrect submission cluster specified by the user.
- Security problem, e.g. user credentials expired or are not available.
- Files pre-staging could not be realized, e.g. due to the incorrect destination file URIs or a data transfer failure.
- Numerous types of submission failures, such as: inability to locate the service, detecting service misconfiguration, service internal problem, not enough resources to execute the job, failure of the network connection with the service, user authorisation failure, user proxy lookup/open failure, local scheduler failure detected by the service, client problems, data transfer to the broker failure, job communication initialisation failure, and invalid request problems, such as incorrectly specified standard streams' names, arguments or maximum execution time.

Failures during the execution. Job execution consists of three phases performed by the Executor: initialisation, the actual job run and cleaning. Failures might occur during each stage. They might be caused by:

- The inability of the Executor to locate the executable file or the lack of executable permissions to it. For applications developed in Java – the inability to locate a class or a method to execute.
- Security problem when executing the command.
- Talker problem, such as impossibility to open a talking channel (when it is already used or the execution environment misbehaves), or reception of an application signal which could not be parsed (when application misbehaves),
- Stream forwarding problem, such as inability to create file containing output streams of the application.
- Interception of an application uncaught error by the Executor.
- Executor internal problem which generally should not – but might – happen and in this case should not mislead the user as to the reason for the application failure.

Except failures detected by the executor, the application itself might request a failure by sending an application-specific fatal signal. The service collects job's heartbeats send by HeartbeatSender and in case they are missing for a long time, the job crash, host crash or network outage are suspected.

Post execution failures. After the job execution, the GATEngine controls the post-staging phase, which might fail, for instance due to the incorrect destination URIs or the inability to perform appropriate transfers. When the GAT Job reports the job to be done, the service checks the following conditions to determine whether the execution was successful:

- The submission or execution failures did not occur.
- Heartbeats were received during the job execution.
- The executor reported that the job is done by sending a JobDone signal.
- The detected exit value is 0.
- The expected output files are present.

The user might override parts of this scheme, for instance by requesting in their policy to ignore the exit code of the application. In case a network outage during application execution was suspected or the service crashed and resumed its work during the job execution, conditions are weakened and additional checks are performed.

4.2.5. Recovery techniques

When the FailureRecoveryAgent was activated and the problem was detected and diagnosed, the DecisionMaker must come up with a suitable RecoveryScenario to amend the problem.

As we cannot prevent grid resources from failing, what we can basically do at the task level is restarting the application in the way that diminishes the probability of the failure recurring. Therefore we mask grid transient failures using various types of RETRY techniques: retrying on a different cluster (what provides hardware and software diversity), retrying on a machine with certain characteristics, such as a faster processor, more disk space, more memory or with more network bandwidth available or restarting after a certain time – it is for example possible to equip the service with information about the maintenance windows.

Another technique used is CHECKPOINTING. It can be used only if the application is developed so that it can write checkpoints or the grid provides system-level generic checkpointing. The stored state information should be enough to restart the process, even on a different resource. This technique is useful for processes which involve significant data manipulation and are unstable or cannot finish data manipulation in a single run.

If the application fails a certain number of times, the failure must be considered permanent. The application FAILURE is REPORTED back to the user. Also in certain cases Executor failures might be worthy to IGNORE, as we may still hope that the application executes successfully.

To select a proper recovery scenario, artificial intelligence techniques might be used, to base the decision also on job history and overall failure occurrence rates.

4.3. Framework implementation

4.3.1. Programming tools

The service was implemented using open-source programming tools. The programming language used was Java^{TM} [60], to provide platform independence according to Java's principle "write-once, run anywhere", for both the service and the executor. It was also selected, because Java implementation of the Grid Application Toolkit, the JavaGAT library (see Section 3.2), is available. The service was tested using mainly the Globus adaptor, implemented using the Java Commodity Grid (CoG) [55] interfacing the GRAM [46] protocol.

4.3.2. Service implementation

The JobController thread executes the server part of the service. The controller wakes up periodically and repeats the algorithm of the jobs control, which typically is as follows:

- If the job is ready to be submitted, the ExecutionPlanner prepares and verifies the plan of the execution, for example it decides on which host the job should be run and what are job requirements. It bases its decisions on job description, job history, information about the grid and information from the SelfAdaptationAgent (see below). Afterwards, the GAT job description is prepared and submitted using the GATEngine to the grid. Submission failures are detected.

- If the job is executed on the grid, the controller refreshes cached job information provided by the GAT. When the GAT reports that the application was done, its execution is verified according to criteria described in the Section 4.2.4.

However, the algorithm gets complicated in two cases:

Missing job heartbeat. If heartbeats from a job are missing, possible reasons include: deaths of the job process or the Executor, a crash of the machine executing the application, a transient or permanent network outage or a problem with connecting to the service (for example when the RMI registry of web service crashed). The service tries to differentiate between these problems. It uses the SelfChecker agent to eliminate the problem with connecting to the service. It checks aliveness of the machines executing the applications and aliveness of the connection to the cluster. In case they are alive, an unexpected job death is assumed, otherwise a network outage is suspected. We try to wait until a transient network outage is over but a permanent network outage must be reported so that the job could be restarted somewhere else.

Service crash and recovery. The service can stop working when the hosting machine crashes, or reboots or the process fails due to a hardware crash or for other reasons. When resuming its work, the service must not discard previously activated jobs. The obstacle is that connections with the resource broker created when job was submitted, are lost now and might not be possible to re-acquire. Now the service must rely on the heartbeats sent by Executors. When the job properly sends a signal JobDone it is assumed to be finished. The service performs manual post-stage phase (normally it is done by the GATEngine). The conditions for job execution success are weakened (exit value availability, JobDone reception).

The job control algorithm is influenced by two factors:

Self adaptation. The SelfAdaptationAgent collects information about events occurring on the grid and computes per-cluster statistics. This enables the service to detect misbehaving clusters, which for example accept jobs but do not execute them or are unreliable and fail very often. The service can then better choose execution hosts for future jobs.

User policy. The user influences many aspects of job execution and failure handling processes, by submitting a policy along with the job. The policy specifies conditions to verify the application successful execution, for example whether the exit value of the process should be ignored or what is the maximum execution time for the process. It also defines interpretations for application-specific signals, for execution problems, for intercepted uncaught exceptions (for applications implemented in Java) and for exit values. Interpretations might be applying a certain recovery scenario or sending a debug information to the user.

4.3.3. Executor implementation

Each executor runs several threads of control, each performing a specific task. First of all, the HeartbeatSender sends heartbeats to the service. Secondly, two InputStreamForwarders forward application output as the Executor output.

Two kinds of Executors were implemented – a GenericExecutor which can execute any application, and a JavaExecutor which can execute only applications implemented in Java but provides more functionalities for them. The mechanism of communicating with the service is the same in both executors: when sending a signal, they lookup the service in the RMI registry and invoke remotely a signal(Signal, RESJobId) method. If the service cannot be found, because the network is down, important signals are queued and await for the outage to be over.

However, the executors differ in the way they run the application and talk to it. If an application is implemented in Java, it is expected to extend the `res.javabind.Signaller` class. When the JavaExecutor is to execute the application it locates pre-staged jar file and the class and appropriate static method, for example the `main` method. Next, using Java reflection mechanisms, it initialises signalling mechanism in the application (because the application does not a priori know its executor). The application sends signals by "normal" local method invocation on the Executor object.

If the application is *not* implemented in Java, things are not that simple. We need a generic way to talk to the application (see Section 4.4.2) – we used sockets, inter-process communication abstraction, available on all popular platforms and in all popular programming languages. If the application is willing to talk to the Executor, it opens a socket and sends "signals" through it. On the other side of the socket, the Talker thread listens to what the application has to say, parses and forwards messages via the Executor to the service.

As low-level socket programming is not appealing too all developers, we implemented small language-binding libraries that a user can link to their programs. Two non-Java language-bindings libraries were prepared: a Python library `pyres` exporting routines `res.initialize(port)`, `res.signal(signal)` and `res.cleanup()` and an analogous C library.

4.3.4. Security

The service takes advantage of the Globus MyProxy Credential Management System [27] deployed on the grid. Each user entitled to run jobs on the distributed machine owns an X.509 Public Key Infrastructure certificate approved by the grid's Certificate Authority and the administrator. Before executing a job, the user runs `myproxy-logon` to authenticate, store the certificate temporarily in the MyProxy credentials repository and secure it with a password. The password is then given to the service which can authenticate the user before running a job on their behalf. Other security contexts, supported by the JavaGAT library, could be also used in the service.

However, delegating POSIX (UNIX) file permissions could not be done easily. Therefore, the user must make sure that the service can access files which are to be pre-staged: the service might be run by a privileged user or the files might be made readable to the appropriate user group or to everybody. This might only be a problem when staged files are to remain secret. An appropriate solution would be accessing files through a grid file system abstraction where permissions would not be file system permissions but "grid permissions".

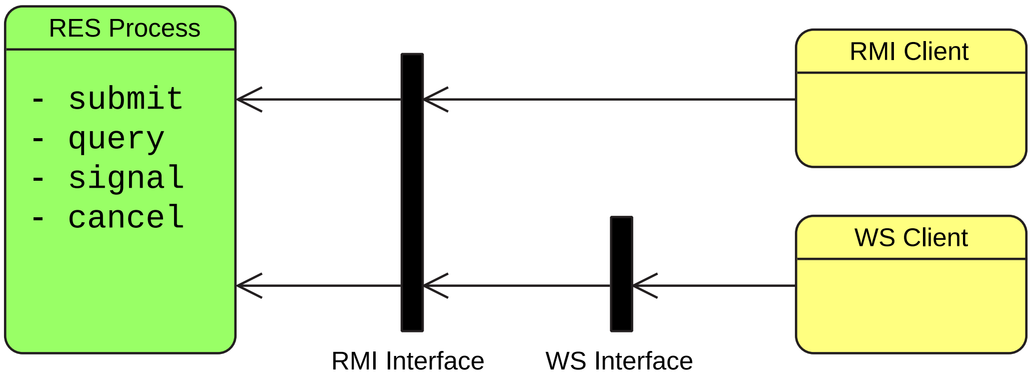
4.3.5. Service interfaces and clients

The service exports capabilities to submit and manage jobs. The main functionality is the

submit(RESJobDescription, RESUserPolicy)

operation which submits a job described by the user to the service and returns a number uniquely identifying the job, a RESJobId. Afterwards, the job's status, history, execution information and logging information can be retrieved using a query(RESJobId) operation. During the execution, the Executor sends signals to the service – heartbeats, information signals, application signals and signals indicating failures. It is done by invoking a signal(RESJobId, Signal) operation. The job can be cancelled during or after the execution with a cancel(RESJobId) method.

As depicted in the Fig. 4.2, two RES interfaces were implemented. A Java Remote Method Invocation [61] interface (`res.RES`) and a web-service interface on top of it (`res.ws.RES`). Since the former uses Java-specific object serialisation techniques, only applications implemented in Java can use it. The latter mediates the calls to the RMI-Interface, and is thus slower but independent of the client language binding as it uses an XML dialect (SOAP) to send and receive parameters.



Rysunek 4.2: Reliable Execution Service interfaces.

4.4. Further implementation issues

4.4.1. Service platform-independence

The acquisition of the Java programming language to provide platform independence has also its drawbacks – there are UNIX-specific operations not available in Java, for example accessing POSIX execution permissions, discovering that the storage device is full during a write operation or ICMP protocol support. To implement missing capabilities Java Native Interface (JNI) should be used to interface native operating system functionalities. In the early implementation less elegant but simpler solutions were used.

4.4.2. Application language-/platform-independence

Why is it difficult for a generic execution environment, such as a grid, to discover the reason why an application execution failed? First of all, the application communicates its errors to the environment in various ways depending on a particular programming language used. The most popular ways are as follows:

- **Error codes.** Used typically in procedural programming languages, such as C or Perl – the program returns a number/string description of its exit status.
- **Exceptions.** Used typically in object-oriented languages, e.g. Java, Python, OCaml – methods raise exceptions carrying rich information about the error and a change of control flow [40].
- Other mechanisms such as **POSIX signals** or **logging** are also used.

Thus, it is difficult to detect them, as it has to be done basically in a different way for all programming languages. And, even if we detect it, the errors are known at the execution *lowest* level – for the user to know about them, they need to be propagated up, what does not happen or happens incorrectly [38, 40] on the grid, because of multiple in-between middleware layers.

4.4.3. Reengineering the JavaGAT library

A number of JavaGAT shortcomings were discovered during the implementation of the Reliable Execution Service.

First of all, it turned out that even though failures are detected and exceptions thrown, diagnosing the failure cause based on the exception was impossible. The JavaGAT library only used two very generic exceptions with adaptor-specific exceptions nested inside. For example an inability to locate the service, a problem common to all adaptors, would manifest differently in each adaptor. This solution violates a number of exception handling rules given in [40]. Therefore, parts of the JavaGAT engine and adaptors had to be reengineered. A broad spectrum of possible failures was incorporated in the library and if an adaptor discovered an adaptor-specific failure is would be converted into an exception understood by the engine. The exception would then clearly determine what the encountered problem was about.

Secondly, the set of possible states the job is in, should also include phases such as `PRE_STAGING` and `POST_STAGING`. The lack of the latter one was especially painful, as it is difficult to predict the time between stopping the application (JobDone signal) and really application being done. If the post staging phase takes too long, it can be taken as a hint to a network outage.

Moreover, in the early implementation of the JavaGAT library a number of features were not implemented. There was no maximum execution time support and no checkpointing support.

4.5. Summary

We described the architecture and the implementation of the Reliable Execution Service – a framework designed to solve typical problems related to developing and executing applications in the unfavourable distributed grid environment. An application is not executed directly but it is wrapped in an Executor which controls the execution as "near" the application as possible, on the target machine. Thanks to the Executor, feedback from the resource brokers and monitoring services, we detect, diagnose, classify and handle application failures. The framework was implemented using a high-level grid system API – the Grid Application Toolkit. Although this piece of software turned out be faulty with respect to the error information propagation, it was partially reengineered and could be successfully taken advantage of in the implementation.

Rozdział 5

Framework Evaluation

The Reliable Execution Service was tested on the DAS-2 distributed computer. The service was tested against correct detection and handling of most frequent failure causes, during all stages of the application execution and related to both the user and the grid environment. This chapter describes the testbed used and the tests performed to evaluate the solution framework.

5.1. The DAS-2 Testbed

The Distributed ASCI Supercomputer 2 (DAS-2) is a wide-area distributed machine situated in the Netherlands. It consists of 200 nodes, grouped in five clusters located at five Dutch universities – the Vrije Universiteit's cluster, containing 72 nodes, is the largest and other clusters consist of 32 nodes each. Each node has two 1 GHz Pentium-III processors, 1 GB Random Access Memory, 20 GB of local disk and access to shared disks, for example with `home` directory. Nodes within each cluster are connected with 2 Gigabits per second Myrinet-2000 and Fast Ethernet networks and clusters are connected with up to 10 Gbps Surfnet 6 backbone.

The platform of the grid front-end machine the service was run on was Linux, kernel version 2.4.21-37.0.1.ELsmp. The local scheduler used on DAS-2 is Sun Grid Engine [56].

The DAS-2 computer is not a production site but is dedicated entirely to parallel and distributed computing research.

5.2. Tests

The service was tested to verify that it correctly detects, diagnoses and handles failures when executing applications on a grid and that it does not significantly decreases performance of the application. Here we first give a list of typical "failure use cases", caused by an incorrect submission or application misbehaviour. Then we test for detection and handling certain grid-related failures, such as host or process crashes. Finally, the reliable execution broader context is presented by implementing and testing two high-level distributed applications.

5.2.1. User-related failures

User-related failures were tested by preparing logically incorrect submission scripts or executing purposely misbehaving applications. The use cases and the description of resulting service behaviour is presented below.

Authorisation failure. The user did not initialise their credentials – in case of the DAS-2 grid – the user did not perform a `myproxy-logon` operation prior to the execution, or the credentials expired. The service detects the problem and reports the failure and its cause to the user.

Executable file problem. Redefining failure handing scenarios. The location of the executable file is incorrect, the file does not exist or the user has no permissions to execute it. For applications developed in Java: the class or the method to execute were not found or the Java security context prevented the method from running. In all cases, if the users did not specify otherwise, the Executor detects the problem and forwards it to the service which reports the failure and its root cause to the user. If the user's policy requested applying a custom interpretation to the executor problem, for example CannotExecuteCommandProblem → EXECUTABLE_NOT_FOUND → SCENARIO_RESTART, then the provided scenario is executed.

Non-success exit code. An application implemented in C executes an incorrect memory operation which typically results in a "Segmentation fault" written to the standard diagnostic stream and the process being aborted with an error exit code. Files containing standard streams are staged back to the user, the service recognises application failure and reports it along with the exit code to the user.

Pre-staging failure. An application requests pre-staging of the file which does not exist, the user is denied read access to or the file cannot be read, for example because it is being locked by another application. The failure is detected and reported to the user.

Interception and handling of application's uncaught exceptions. The application fails to write out its temporary files, because there was not enough disk space. It throws a `MyDiskFullException` (Java does not handle disk full with an appropriate exception, see [40]). The exception is intercepted and interpreted according to the policy submitted by the user. The policy specifies that on the interception of this exception, the service should restart the application on a different host, possibly with more disk space available. Information about the exception is propagated to the user and the application is restarted.

Service-aware application. A scientific application implemented in C / Python performs a long computation algorithm 10 times. The user would like to know immediately when each step is done. The application takes advantage of the predefined library `cres` / `pyres` and on completion of each step sends a signal to the Executor. The Executor forwards signals to the service where they are interpreted, according to the user's policy, as debugging signals. The service puts debugging information in the job history.

Custom interpretation of program exit codes. Probabilistic search algorithm application might exit with two different exit codes, one indicating success (solution found) and one indicating failure (solution was not found). While in the former case, the application just executed successfully, in the latter case it should be restarted, to try to find the solution again. The user specifies an exit value interpretation in their policy: Exit-value 5 → SCENARIO_RESTART. The service properly restarts the application.

Maximum execution time. If the execution of a linear solver takes more than 10 minutes, the algorithm did not converge, and the application must be restarted. The user specifies the maximum execution time in the user policy. Application is stopped and restarted if the execution time exceeds 10 minutes.

A checkpointable application. The user runs a tree search algorithm which takes a long time to execute. Therefore, to make sure that in case of a failure, partial computation is not lost, the application saves periodically its current search path to a file. On startup, the application reads the file in and resumes computation from the point where it has stopped. The user's policy specifies that the application is checkpointable and gives the name of a checkpoint file. The service takes care about staging it in case of a failure and migration to another resource.

5.2.2. Grid-related failures

Grid-related failures were difficult to test, they must have been simulated. Selected tests and resulting service behaviour are given below.

Application-related process crash. The application abortion might be the result of executing an illegal operation such as accessing non-allocated memory or numerical exception. For C/C++ or Python applications such problems manifest themselves by returning a non-zero exit code whereas for Java applications an exception is thrown. Both situations are detected and handled, as described in the previous section.

Environment-related process crash. Processes executing on a grid node might fail, for instance due to incorrect workings of the operating system or due to a hardware crash. The following tests were performed:

1. In case of the execution of an application implemented in Java, there is one system process running – the Executor. We can simulate process crash, by connecting directly to the computational resource and killing the process. The resource broker informs the service that the application is done. The failure is recognised in the service due to the lack of a JobDone signal from the Executor. The service restarts the application.
2. When a generic application is executed, the Executor must start an additional process for it. The crash of the Executor process is detected and handled similarly to the case of a Java application. A crash of the actual application process is more difficult to detect though. In theory, in a UNIX-like operating system, if the application is killed by a signal, the exit code should indicate the type of the signal (typically the application returns the maximum exit value plus the signal number). Not only is this behaviour platform-dependent but also the Java Virtual Machine does not propagate these exit codes properly. Therefore, in case of an application process crash, the non-success return value is detected and a failure is reported to the user.

Transient network outage. The goal here is to survive short-time network outages without restarting the application and resuming control over it after the network is back on. Two types of tests were performed: when the application still runs and sends heartbeat and when the application was already done after the network outage had been over. Network outages were simulated by temporarily ceasing to send signals to the service. When the network was working again applications finished successfully.

Host crash / Permanent network outage. The detection of network failures tends to be difficult because it is hard to discriminate between a host failure and a network failure without the existence of a second, independent path [39]. In both cases the signs of a problem are the same – the host or the cluster does not respond – and the goal is to detect it, classify as permanent problem and restart the application on another cluster. The service collects data on when the contact with the Executor was made. When the time of missing heartbeats exceeds a certain threshold the network outage is assumed to be permanent and the application is restarted on another resource.

Cluster misbehaviour. Cluster misbehaviour, such as accepting jobs and not executing them or causing all of them to fail, is detected by the SelfAdaptation component. It computes on the fly per-cluster statistics concerning failure occurrence rates and most frequent failure causes. These statistics can be used as hints to cluster misbehaviour. For example, if a cluster has a 100% failure rate and sufficient number of submissions were tried, we can assume that this cluster is misconfigured or overloaded and cease to use it for some time. This feature was tested "accidentally" in real life – during test runs of applications described in Section 5.2.3 one of the DAS-2 clusters was several times temporarily overloaded and would accept jobs to the queue and then never actually execute them or execute them after an unacceptably long time. The SelfAdaptationAgent detected the problem and its reason – scheduling timeout. The information was used by the ExecutionPlanner to omit the cluster in future job plans.

Service recovery. Service recovery was tested by submitting jobs to the service, killing the service process and restarting it after some time. For jobs which were still running when the service had recovered, the heartbeats reception was resumed. Because after the recovery the GAT connections with activated jobs were lost, determining that the job is done bases on received heartbeats. When heartbeats were detected missing, the job was assumed to be done, the output files post-staging was performed and the execution result was verified against weakened successful execution conditions (see Section 4.3.2). Jobs can execute in spite of the service failing.

5.2.3. A broader context

We implemented two real-life applications which substantially take advantage of the Reliable Execution Service. The first application shows that the use of service API is quite straightforward. The seconds one was used to estimate how using the service affects application performance.

Workflow-level fault tolerance. An *alternative task* technique is a workflow-level fault-tolerance technique used in distributed applications. A typical example might be as follows: a user wants to have some computation done. He/she has two algorithms at their disposal: first one – faster and using a lot of disk space – and another one – slower but with less disk space requirements. The user would like to run the first algorithm, and only if it fails because of insufficient disk space, the second algorithm should be tried. How to implement such a feature is presented in the Fig. 5.1. We note the simplicity of this code which suggests that building of a client-side workflow engine is straightforward. The disadvantage of this solution is that the user-decoupling is not so strong anymore.

Task farming application. *Task farming* is a work distribution model often used in parallel applications. These applications typically consist of a single master process which hands out a large number of independent tasks to workers (slaves). Workers then process tasks simultaneously and return results to the master. Because workers might fail, the master must usually implement a fault-tolerance layer to detect failures and appropriately restart workers. However, when

```
boolean submitAndWait(RESSoftwareDescription sd, RESUserPolicy policy) {
    // Locate the service
    RemoteRES res = RESServiceLocator.find();
    // Submit the job
    RESSubmissionResult sr = res.submit(sd, policy);
    // Retrieve the resulting job id
    RESJobId jobId = sr.getJobId();
    // Wait until job is done
    RESQueryResult qr = null;
    while (!(qr = res.query(jobId, true)).isDone());
    // Check if job is done successfully
    return qr.isDoneSuccessfully();
}

// Prepare software descriptions and user policies for both algorithms
// Try the first algorithm
if (!submitAndWait(resSD_fast, userPolicy_fast)) {
    // The first algorithm failed, try the second one
    submitAndWait(resSD_slow, userPolicy_slow);
}
```

Rysunek 5.1: The code of an *alternative task* technique implementation (only important parts).

submitting workers to the grid through the Reliable Execution Service, the master process is assured that each worker will execute successfully.

A generic task farming framework was implemented and two applications were tested:

(a) A simulation application. In the simulation application, each worker sleeps for a specified time and crashes with a specified probability. A task crash is achieved by not generating expected output files and additionally by executing a call to `System.exit()` which kills the Executor before it reports to the service that the job is done. The policy submitted by the master for the worker requests the maxRetries parameter to be unlimited. The service retries the worker as many times as necessary, until the successful completion.

(b) A prime factorisation application. In the factorisation application, each worker gets a large natural number, factorises it and prints the result to a file which is then staged back to the master. This application was used to estimate the execution service overhead on application performance. In this section we present how the timings were gathered and processed, we give results of the experiment and conclusions concerning the service overhead on the application.

Test procedure. The sequential timings were measured first. As factorising workers are implemented so that their execution times are almost the same provided that arguments do not differ much, it was enough to measure the average time of *one* worker executing on a random node. The worker was run 10 times using a cluster prun tool which interfaces directly the local scheduler of the university cluster and uses a remote shell rsh to execute the program. The application was run when the scheduling queue was empty.

The parallel timings were measured next. The application was run using the task farming framework for $1, 2, 4, 8$ and 16 workers, each time executing 32 factorisation tasks. For each workers count, scheduling and execution timings of each worker and the overall master execution time were collected. As execution of 32 tasks takes a long time and uses all clusters in the grid, it was not possible to run it with all of the queues empty. However, the tested factorisation application was the only one executed by the service at a time.

Processing collected data. To understand the overhead of the service on the parallel application, the execution time must have been "normalised" – reduced by the time that tasks spent waiting in scheduling queues. The normalised overall master execution time is the maximum of normalised times for workers.

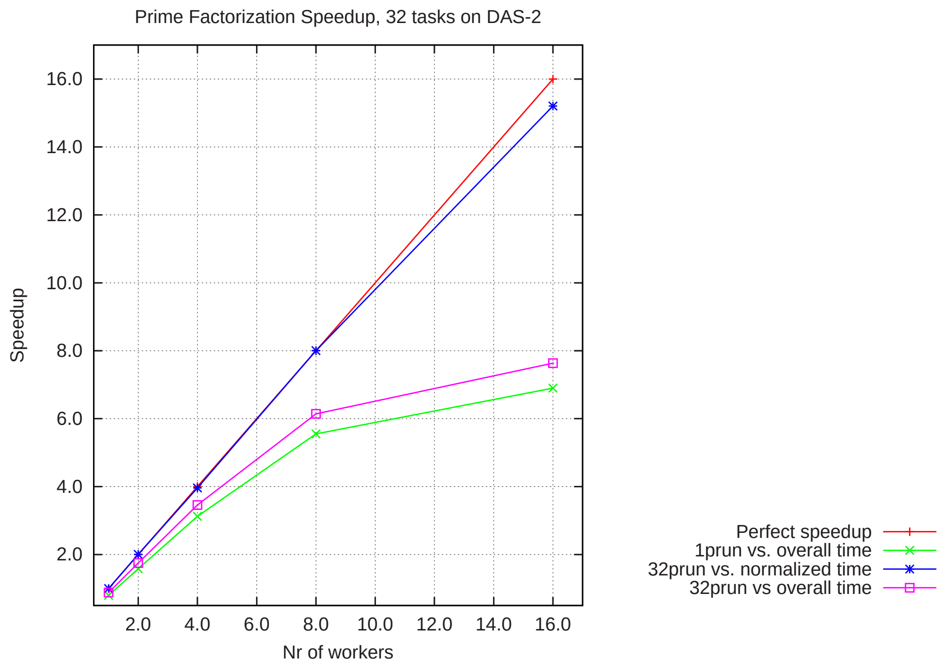
In parallel computing, the *speedup* refers to how much a parallel algorithm is faster than the corresponding sequential algorithm. It is defined by a formula: $S_p = \frac{T_1}{T_p}$, where p is the number of workers (processors), T_1 is the execution time of the sequential algorithm and T_p is the execution time of the parallel algorithm with p processors. The perfect speedup is $S_p = p$ when doubling the number of processors doubles the speed – the algorithm is perfectly scalable.

For the prime factorisation application speedups were computed against two different sequential times:

(a) the time of running all 32 applications "at once" – a theoretical sequential time, appropriate to use to determine the real speedup of the task farming approach but not helpful in our case,

(b) the time of submitting each 32 task separately one after another – a "realistic" sequential time for us.

The results are presented in the Fig. 5.2.



Rysunek 5.2: Prime factorisation application speedups. 1prun is the time of running 32 tasks sequentially, submitting the application only once. The 32prun time is how long it takes to submit each task separately one after another. The overall time is the execution time reported by the master process. It is normalised by reducing it by the workers scheduling time.

Observations and conclusions. The overall execution time reported by the master process depends on the current grid usage and the network load and therefore gives little information on how using the execution service affects application performance. It can be understood though by comparing the perfect speedup and the speedup 32prun vs. normalised time – as it compares 32 submissions using a low-level cluster tool and 32 submissions to the grid using the service.

The graph shows that the service scales very well. For small numbers of workers, the time is almost exactly the same. For 16 workers there is a small decrease in the performance, caused mainly by the fact that the service must handle more applications at the same time. We conclude that the service does not significantly affect application performance.

5.3. Summary

We tested the Reliable Execution Service on the DAS-2 Dutch distributed computer. We showed that the service correctly detects and handles most frequent failures, during all stages of the application execution, caused by both the user or by the unfavourable grid environment. Finally, we demonstrated that the service can be used to help to easily develop grid applications. Using the generic task farming framework, we examined the overhead of the service on application performance and concluded that it is not significant.

Rozdział 6

Conclusion

A Reliable Execution Service was successfully designed, implemented and tested. We conclude with the discussion of limitations of the service, possible future directions and a summary of work done in the thesis.

6.1. Limitations of the service

The service acquires a simple stage-in/compute/stage-out job model, mainly because it was used in the Grid Application Toolkit. There are two important consequences of this fact. First of all, the simple staging model does not support workflowing. If we need to build an application with more complex staging behaviour, it has to be done on the client side. However, we showed that developing workflowing features using the reliable execution service is straightforward (see Section 5.2.3).

Secondly, the application must be a computational application. Support of the grid resources failure detection is beyond the scope of the service. For example, if the application contacts an external database or a grid service, the developer must write code to detect crashes of used resources while their handling can be done by the service. However, in case such a resource failure detection service was installed on the grid, it could be neatly integrated with the Reliable Execution Service, to create a versatile reliable execution framework.

6.2. Future work

The executor mechanism. The Executor controlling the application on the target machine – used in the service, turns out to be a powerful tool and could be used to implement the following interesting features:

1. In case of jobs executing on multiple machines, executors could communicate with each other and cooperate. The tasks could be restarted in a very timely fashion, without consulting the service. Implementation of collective behaviours within the task group is possible, for example a "collective fault-tolerance" – a concept explored in detail in the Fault-Tolerant MPI [20].

2. A sudden decrease in the application's performance should be considered as a failure but this type of a failure is extremely difficult to detect [7]. The Executor could make it possible though. Many scientific application are "loop-based", if they regularly inform the Executor about finishing each step, the executor might detect a decrease in the application's performance and alarm the service.

3. Using the Executor the service can communicate with the application, to request execution of a method exposed by it, for example performing a checkpoint.

UDP Heartbeats. Service performance could be improved by changing the way heartbeats are send. If they are send too frequently and many job execute at the same time, the network and service load could increase substantially. In the early implementation, signalling is done through RMI calls, which are synchronous and TCP/IP based. A simple solution is to replace them with User Datagram Protocol (UDP) packets. The UDP protocol is, contrary to the TCP protocol, *not* reliable, and therefore much faster. As the service must anticipate heartbeats delays due to network congestions anyway, loosing a UDP heartbeat from time to time would go unnoticed.

Monitoring. If monitoring services are installed on a grid, they could enhance the Reliable Execution Service's capabilities of failure diagnosis and even prevention. In case of an application crash, the host could be monitored for additional information to better diagnose the failure root cause. It could be also used to refine variety of application restarting scenarios.

6.3. Conclusion

We have argued that distributed environments, especially grids, are inherently unfavourable and unreliable. Frequent failures of their components and applications make development difficult, especially to scientists who are not necessarily grid experts. Thus, failure-tolerance techniques should be exploited and the end user should be shielded from their details.

We have successfully designed and prototyped the Reliable Execution Service (RES) which executes applications on the grid and incorporates fault-tolerance masking techniques. Our additional goal was to make the service independent of middleware installed on a particular grid. It was achieved by using the Grid Application Toolkit library, a high-level, uniform, integrated grid API. Although this piece of software turned out be faulty with respect to the error information propagation, it was partially reengineered and could be successfully used in the implementation.

An important insight was the Executor concept – instead of executing the application directly on the target machine, we use a wrapper which controls the execution as near the application as possible. The Executor turns out to be a powerful tool in providing interesting fault-tolerance features. Thanks to the Executor, feedback from the resource brokers and possibly monitoring services, we detect and diagnose failures, classify them into transient and permanent and apply appropriate refined restart scenarios.

We tested the Reliable Execution Service on the DAS-2 Dutch distributed computer. We demonstrated that the service can be used to help to easily develop grid applications. Finally, using a real-life task farming application, we examined how the service affects performance of applications and concluded that the overhead is not significant.

Basing on this thesis and with input from Barcelona Computing Centre's researchers Raül Sirvent and Rosa M. Badia, Thilo Kielmann and I wrote an article *A Service for Reliable Execution of Grid Applications* and submitted it to the CoreGRID Workshop on Grid Middleware in Dresden, Germany, in August 2006.

Bibliografia

[1] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schuett, E. Seidel, and B. Ullmer. The grid application toolkit: toward generic and easy application programming interfaces for the grid. *Proceedings of the IEEE*, 93:534–550, 2005.

[2] G. Allen, T. Goodale, H. Kaiser, T. Kielmann, A. Kulshrestha, A. Merzky, and R. van Nieuwpoort. A day in the life of a grid-enabled application: Counting on the Grid. *Workshop on Grid Application Programming Interfaces*, 2004.

[3] A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability, 2001. Research Report N01145, LAAS-CNRS, April 2001.

[4] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 01(1):11–33, 2004.

[5] M. Baker, R. Buyya, and D. Laforenza. The Grid: International efforts in grid computing. *International Conference on Advances in Infrastructure for Electronic Business, Science and Education on the Internet*, 2000.

[6] A. Beguelin, E. Seligman, and P. Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997.

[7] M. Bubak, T. Szepieniec, and M. Radecki. A proposal of application failure detection and recovery in the Grid, 2003. Cracow, Grid Workshop.

[8] CERN. Grid-Café. http://gridcafe.web.cern.ch.

[9] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[10] CoreGRID homepage. http://www.coregrid.net.

[11] CoreGRID, Annex I - Description of Work, 2004. Grid-based systems for Complex Problems Solving, Sixth Framework Programme.

[12] CoreGRID Roadmap version 1 on Problem Solving Environments, Tools and Grid Systems. CoreGRID deliverable D.ETS.01, 2005.

[13] CoreGRID Virtual Institute on Problem Soving Environments Tools and Grid Systems. Proposal for Mediator Component Toolkit. CoreGRID deliverable D.ETS.02, 2005.

[14] CoreGRID Virtual Institute on Programming Models. Proposals for Grid Component Model. CoreGRID deliverable D.PM.02, 2005.

[15] X. Défago, N. Hayashibara, and T. Katayama. On the design of a failure detection service for large-scale distributed systems. *Proceedings International Symposium Towards Peta-Bit Ultra-Networks*, pages 88–95, 2003.

[16] The Distributed ASCI Supercomputer DAS-2. http://www.cs.vu.nl/das2.

[17] G. Dobson, S. Hall, and I. Sommerville. A container-based approach to fault tolerance in service-oriented architectures. *International Conference of Software Engeneering*, 2005.

[18] European Union 6th Framework Programme (FP6). http://ec.europa.eu/research/fp6.

[19] European Union 6th Framework Programme (FP6) – Network of Excellence (NoE). http://cordis.europa.eu/fp6/instr_noe.htm.

[20] G. E. Fagg and J. J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. *Lecture Notes in Computer Science*, 1908:346–354, 2000.

[21] P. Felber, X. Défago, R. Guerraoui, and P. Oser. Failure detectors as first class objects. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99)*, pages 132–141, Edinburgh, Scotland, 1999.

[22] I. Foster. What is the Grid? A three point checklist. *GRID Today*, 2002.

[23] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the Grid: An open grid services architecture for distributed systems integration. *Systems Integration. Globus Project*, 2002. www.globus.org/research/papers/ogsa.pdf.

[24] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, vol. 2150, 2001.

[25] I. Foster, H. Kishimoto, et al. Open Grid Services Architecture. Technical report, Global Grid Forum, 2005-2006. Versions 1.0-1.5.

[26] The Global Grid Forum (GGF). http://www.ggf.org.

[27] Globus Toolkit. The MyProxy Credential Management Service. http://grid.ncsa.uiuc.edu/myproxy.

[28] 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*, 2005. http://wiki.cct.lsu.edu/saga.

[29] S. Hwang and C. Kesselman. A generic failure detection service for the Grid. Information Sciences Institute, University of Southern California. Technical Report ISI-TR-568, 2003.

[30] S. Hwang and C. Kesselman. Grid workflow: A flexible failure handling framework for the Grid. *High Performance Distributed Computing*, 00:126, 2003.

[31] A. Jain and R. K. Shyamasundar. Failure detection and membership management in grid environments. *Fifth IEEE/ACM International Workshop on Grid Computing*, 00:44–52, 2004.

[32] T. Kielmann, A. Merzky, H. E. Bal, et al. Grid Applications Programming Environments, 2005. CoreGRID Technical Report, TR-0003.

[33] T. Kielmann, G. Wrzesinska, N. Currle-Linde, and M. Resch. Redesigning the SEGL problem solving environment: A case study of using mediator components. *GRIDs@work*, 2005.

[34] G. Kola, T. Kosar, and M. Livny. Phoenix: Making data-intensive grid applications fault-tolerant. *In Proceedings of 5th IEEE/ACM International Workshop on Grid Computing*, 2004.

[35] R. Medeiros, W. Cirne, F. Brasileiro, and J. Sauve. Faults in grids: Why are they so bad and what can be done about it? *Fourth International Workshop on Grid Computing*, 00:18, 2003.

[36] A. Merzky and S. Jha. SAGA Use Cases. Technical report, Global Grid Forum, 2005. https://forge.gridforum.org/projects/saga-rg.

[37] A. Merzky and H. Kaiser. *Final GAT-API Specification*, 2005. GridLab document: IST-2001-32133.

[38] W. Smith and C. Hu. An execution service for grid computing. NAS Technical Report NAS-04-004, 2004.

[39] P. Stelling, C. DeMatteis, I. T. Foster, C. Kesselman, C. A. Lee, and G. von Laszewski. A fault detection service for wide area distributed computations. *Cluster Computing*, 2(2):117–128, 1999.

[40] D. Thain and M. Livny. Error scope on a computational grid: Theory and practice. *11th IEEE International Symposium on High Performance Distributed Computing*, 00:199, 2002.

[41] R. van Nieuwpoort. Getting started with the Grid Application Toolkit. Vrije Universiteit, Amsterdam, the Netherlands. http://www.gridlab.org/WorkPackages/wp-1/Doc/JavaGAT-tutorial.pdf.

[42] Vrije Universiteit, Computer Science Department. http://www.cs.vu.nl/en.

[43] S. A. Yin, Y. Chen, G. Dobson, S. Hall, C. Hughes, Y. Li, S. Qu, E. Smith, I. Sommerville, and M. Tjen. Dependable Grid services, 2003. Dependable, service centric grid computing project.

[44] Condor and Condor-G projects. http://www.cs.wisc.edu/condor.

[45] The DataGrid project. http://eu-datagrid.web.cern.ch/eu-datagrid.

[46] The Globus project. http://www.globus.org.

[47] The GridLab project. http://www.gridlab.org.

[48] The GridLab project. JavaGAT – the implementation of Grid Application Toolkit in Java. http://www.gridlab.org/WorkPackages/wp-1.

[49] LHC Computing Grid (LCG) project. http://lcg.web.cern.ch/LCG.

[50] LHC Computing Grid (LCG) monitoring site. http://goc.grid.sinica.edu.tw/gstat.

[51] NASA Information Power Grid. http://www.ipg.nasa.gov.

[52] Berkeley open infrastructure for network computing. http://boinc.berkeley.edu/.

[53] CAVE-Study project. http://www.cs.vu.nl/ renambot/vr.

[54] IC-Wall project. http://www.icwall.nl.

[55] The Globus Community Grid (CoG). http://wiki.cogkit.org.

[56] The Sun Grid Engine (SGE). http://gridengine.sunsource.net.

[57] Apache Ant Java build tool. http://ant.apache.org.

[58] Apache Axis web-service engine. http://ws.apache.org/axis.

[59] Apache Tomcat application server. http://tomcat.apache.org.

[60] Java^{TM}programming language. http://java.sun.com.

[61] Java Remote Method Invocation technology. http://java.sun.com/products/jdk/rmi.

Dodatek A

Software bundle

This chapter describes what the thesis bundle consists of. Notes on service installation, running and usage can be found in the Appendix B.

The thesis bundle

The thesis is packed as an archive containing the following folders:

Location	Description
`thesis`	Contains this document.
`impl/res`	Implementation of the Reliable Execution Service.
`impl/engine`	The (reengineered) JavaGAT engine implementation.
`impl/adaptors`	The (reengineered) JavaGAT basic adaptors implementation.
`impl/gridLabAdaptors`	More JavaGAT adaptors, from the GridLab project.

The service implementation bundle

The implementation of the service, in the `impl/res` folder, consists of several important parts:

Location	Description
`src`	The sources of the Reliable Execution Service.
`dist`	The service libraries. Consult B.1.
`c-binding`	The C language binding. Consult B.6.
`python-binding`	The Python language binding. Consult B.6.

Scripts

Number of useful scripts were created to manage the service, some for administration, some for the service users:

Script	**Type**	**Description**
`build.xml`	admin	Builds the service libraries. Consult B.1.
`service`	admin	Runs the service. Consult B.2.
`deployer`	admin	Deploys and undeploys the service. Consult B.4.
`submit`	user	Submits a job. Consult B.5.
`cancel`	user	Cancels execution of a job. Consult B.5.
`monitor[_ws]`	user	Monitors execution of a job. Consult B.5.

Examples and testing

Numerous examples of grid applications and appropriate submission scripts were prepared as well. For more information consult B.5 and B.6.

Location	**Description**
`c-tests`	Examples of applications developed in C.
`python-tests`	Examples of applications developed in Python.
`example-clients/*script*`	Examples of bash-scripts to execute on a grid.
`example-clients/client-*`	Examples of submission scripts.

Dodatek B

Service installation and usage

B.1. Installation

The Reliable Execution Service can be built using the provided Apache Ant [57] script:

```
$ ant dist
```

This call results in the service source compilation, the creation of RMI stub files and the creation of libraries in the `dist` directory. The libraries for the service are:

`res-service.jar`	The server part service library.
`res-executors.jar`	Executors library to be run on target hosts.
`res-ws.jar`	The web-service interface. Ready to be deployed.

and the libraries for the user are:

`res-client.jar`	Clients library (RMI/WS) (see Section B.5).
`res-javabind.jar`	Java-language binding library, to develop RES-aware applications.
`res-tests.jar`	Tests library.

B.2. Running

The `RES_HOME` variable must point to the directory with the service installation. The service can be run using the script:

```
$ ./service
```

This script can be edited to adapt the variables used by the service. It sets an appropriate `CLASSPATH` variable and the path to GAT adaptors and runs the service (`res.RES`) using the `res-service.jar` library. When the service is started, it auto-configures, creates GAT interface, reloads the stored data, starts a RMI-registry and exports itself to the registry.

The service was tested on a machine running the Linux operating system. The service is able to run under any platform for which a Java 1.4 implementation is available, for example Windows^{TM} operating system. However, the utilities and helper scripts would have to rewritten.

B.3. Built-in repository

The built-in repository implementation method was chosen for its simplicity. It bases on Java object serialisation feature. The list of jobs maintained by the service is periodically saved to a file, by default to a `res-db` file. The disadvantages of this solution include: the lack of transactions, weak performance as each time the entire file is rewritten and, in case of a change in the implementation of serialised objects, compatibility is lost – stored objects cannot be deserialised. Thus, for a production use an actual database should be used.

B.4. Service interfaces

Basic interface to the service is the RMI interface. It requires a port open on the firewall (1099 by default). The RMI-registry is started by the service.

A web-service interface is a secondary interface to the service. It has the advantage of the client's language-binding independence, contrary to the RMI interface which requires the clients to be implemented in Java. The web-service library `res-ws.jar` is to be deployed on a web server. Using Apache Tomcat [59] application server and the Apache Axis web-service engine [58] this can be done as follows:

1. Make sure that the Apache Tomcat application server is installed and running on the port `port` (8080 by default) and the variable `TOMCAT_HOME` is set appropriately. Make sure that the Apache Axis in installed and correctly deployed, using the context `axis`. Build the Reliable Execution Service and run it (see Sections B.1-B.2).

2. Deploy the web-service using a deployment script:

   ```
   $ ./deployer deploy <port>
   ```

   The script installs the `res-ws.jar` library in the web-service engine, restarts the engine, and talks to the Axis Admin servlet to deploy the service. It uses the deployment descriptor `deploy.wsdd`. To deploy the service you'll need the tomcat manager user and password.

3. Verify that the interface works correctly:

   ```
   $ ./deployer wscheck <port>
   ```

4. The service can be undeployed as follows:

   ```
   $./deployer undeploy <port>
   ```

   The script talks to the Axis Admin using the undeployment descriptor `undeploy.wsdd`.

B.5. Using provided clients

The service distribution contains prepared command-line service clients, both RMI and web-service (WS) based, in the `res-clients.jar` library. Run without parameters or with incorrect parameters, they print the full usage information.

To submit the application `res.client.(RMI|WS)Client` classes or the `submit` script can be used. The client is a command line client accepting multiple parameters:

- Describing the Java or non-Java application. Parameters examples are:

  ```
  --stdout, --stderr, --param,
  --pre-stage, --post-stage
  --executable, --java, --class, --method
  ```

- Describing the user policy which gives hints to the service about the execution and handling application's problems, exit values, exceptions, etc. Parameters examples are:

  ```
  --max-time, --cluster, --max-restarts
  --checkpointable, --checkpoint
  --ignore-exit-value, --ignore-missing-output,
  ```

  including parameters to express interpretations specifications:

  ```
  --problem, --exit-value, --intercept, --app-signal
  ```

When the submission client is run without parameters it prints detailed information about possible submission options. A number of examples of applications submissions were prepared as well – `example-clients/client-*`.

To monitor the submitted job, console clients `(RMI|WS)Monitor` or scripts `monitor[_ws]` can be used. The monitor queries the job once a second and if it receives new job information, it prints it on the screen. For example, to monitor the job with id 5 one can execute command:

```
java res.client.RMIMonitor 5 verbose
```

To cancel the job, before, during or after the execution, the cancel client `Cancel` or a script `cancel` can be used:

```
java res.client.Cancel 5
```

To monitor the whole service we can also use the `(RMI|WS)Monitor` clients, for example:

```
java res.client.RMIMonitor verbose
```

B.6. Using language bindings

The following applications language bindings can be used:

Java

The library `res-javabind.jar` contains the Java language-binding. To use it, the application must extend the class `res.javabind.Signaller` and indicate in the job description that the application is a Java application.

Many examples are included in the tests library, such as the `res.test.RunThrowSignal` class or the `res.test.workflow.*` package. Their submission scripts are respectively `example-clients/client-java-signal` and `client-alternative-task`.

C/C++

The C/C++-binding is implemented in `c-binding/cres.*`. To use it the application must link against `cres.c` and use calls from it, mainly `res_initialize(port)` and `res_signal(sig)`.

Several examples are included in the `c-tests` directory, a typical signalling application `application.c`, a checkpointing application `checkpoint.c` and a typical failing application `segfault.c`. Their submission scripts are respectively `example-clients/client-application`, `client-checkpoint` and `client-segfault`.

Python

The Python-binding is implemented in `python-binding/pyres.py`. To use it an application must "see" the file `pyres.py`, import the `res` module and use its calls, mainly `res.initialize(port)` and `res.signal(sig)`.

An example is included in the `python-tests` directory, `pytest.py` – a typical signalling application. Its submission script is `example-clients/client-python`.

B.7. Documentation

The service concept, design, implementation and evaluation is explained in this thesis. The code JavaDoc documentation can be generated by running the JavaDoc Ant task:

```
$ ant javadoc
```

and viewed in the distribution directory `dist`.