Resiliency in Distributed Workflow Systems for Numerical Applications

Laurentiu Trifan

▶ To cite this version:

Laurentiu Trifan. Resiliency in Distributed Workflow Systems for Numerical Applications. Performance [cs.PF]. Université de Grenoble, 2013. English. NNT : . tel-00912491

HAL Id: tel-00912491
https://theses.hal.science/tel-00912491v1

Submitted on 2 Dec 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Resiliency in Distributed Workflow Systems for Numerical Optimization Applications: Design and Experiments

Laurentiu Trifan
OPALE Team
"Joseph Fourier" University

2013 September

1. Reviewer:

2. Reviewer:

Day of the defense:

Signature from head of PhD committee:

Abstract

Put your abstract or summary here, if your university requires it.

To ...

Acknowledgements

I would like to acknowledge the thousands of individuals who have coded for the LaTeX project for free. It is due to their efforts that we can generate professionally typeset PDFs now.

Contents

1 Introduction **1**

1.1 Context 1

1.2 Problem 1

1.3 Proposed Solutions 3

1.4 Organization of the Mansucript 6

2 State of the art **9**

2.1 Introduction 9

2.2 Distributed Computing Infrastructures and Middleware Software 9

2.2.1 Overview of Grid Computing Systems 9

2.2.1.1 History 10

2.2.1.2 Classification 10

2.2.1.3 User Interaction 12

2.2.1.4 Programming Models 13

2.2.2 Grid5000 14

2.2.2.1 General View 14

2.2.2.2 Architecture 16

2.2.2.3 Tools 16

2.2.3 Middleware Systems 19

2.2.3.1 Globus Toolkit 20

2.2.3.2 Distributed Interactive Engineering Toolbox (DIET) . . 20

2.2.3.3 OAR 21

2.2.3.4 ProActive 23

2.3 Fault Tolerance Methods in Distributed and Parallel Systems 26

2.3.1 Faults - General View 26

2.3.2 Fault Tolerance Techniques 28

2.3.2.1 Fault Detection (1) 28

2.3.2.2 Fault Recovery 29

2.3.2.3 Redundancy 29

2.3.3 Checkpoint/Restart 30

2.3.3.1 Globality 30

2.3.3.2 Independent vs Coordinated Checkpoint 31
2.3.3.3 Message Logging 32
2.3.3.4 Multilevel Checkpoint 33
2.3.3.5 Checkpoint Storage 33
2.4 Workflow Systems 34
2.4.1 General View 34
2.4.1.1 Basic Components of a Workflow System 35
2.4.1.2 Models of Abstract Workflow: Control vs Data Driven . 39
2.4.1.3 Workflow Patterns in Control Driven Models 41
2.4.1.4 Dynamicity in Workflow Systems 43
2.4.1.5 Exception Handling in Workflow Systems 44
2.4.2 Examples of Fault Tolerant Workflow Systems 46
2.4.2.1 Askalon 47
2.4.2.2 Kepler 49
2.4.3 YAWL - Yet Another Workflow Language 50
2.4.3.1 Architecture 50
2.4.3.2 YAWL Custom Service 51
2.4.3.3 Data in YAWL 52
2.4.3.4 Dynamicity 53
2.4.3.5 Exception Handling 54
2.5 Resilience for Long-running Simulation and Optimization Applications . 56
2.5.1 Exception Types 57
2.5.1.1 Resource Limitation Exceptions 57
2.5.1.2 Application Exceptions 59
2.5.2 Exception Detection 60
2.5.3 Exception Treatment and Recovery 60

3 Platform Design **63**
3.1 Numerical Optimization Applications 64
3.2 OMD2 Project 66
3.3 Numerical Application Test-Cases 67
3.4 Famosa Execution Chain 68
3.5 Large Scale Test-Case 71

3.6 Exception Types . . . 73
3.6.1 Practical Exceptions . . . 73
3.6.2 Practical Detection . . . 74
3.7 Platform Design Issues . . . 74
3.7.1 Data Management and Data Synchronization . . . 76
3.7.2 Exception Detection and Recovery Using YAWL . . . 77

4 Implementation and Results 81
4.1 Interface Between YAWL and External Computing Resources . . . 81
4.2 YAWL and Grid5000 . . . 85
4.2.1 Interface Between YAWL and Grid5000 . . . 88
4.2.2 Resource Reservation and Deployment on Grid5000 Infrastructure 89
4.2.3 Distribution of Computation on Grid5000 Infrastructure . . . 90
4.2.4 Passing to a Larger Scale . . . 92
4.3 Resilience: Scenarios and Implementation . . . 96
4.3.1 Sequential Execution on Multiple Different Machines . . . 96
4.3.2 Parallel Execution on Two Different Clusters . . . 100
4.3.3 Sequential Execution with Timeout Exception Handling . . . 101
4.3.4 Distributed Execution with Resource Balancing . . . 103
4.3.5 Speed-up Gain in Large Scale Optimization Application . . . 106

5 Conclusions and Perspectives 113

References 119

1. Introduction

1.1 Context

The design and implementation of large scientific applications, corroborated with the large demand for computational power has led to continuously growing High Performance Computing(HPC) systems under the form of GRID infrastructures or clusters of computing nodes. These applications imply processing large data sets, control flow management and execution on distributed resources. To fill the gap between non-experimented scientists playing the role of users and the complexity of these large distributed systems, the need for user environments easy to configure and deploy has increased considerably. To achieve this, the HPC research community is showing a great interest in workflow systems. These systems provide solutions for facilitating the access to large distributed infrastructures for scientists working in domains like Bio-Informatics, Forecast, Pharmacy, Aeronautics and Automobiles, etc.

1.2 Problem

The large scientific applications involve multiple disciplines in their design and implementation. This means that the execution environment has to regroup all the necessary tools belonging to each discipline and put them at scientists' disposal using user-friendly interfaces. Another characteristic of these applications is their dynamicity. Since their execution time can span over days or even weeks, it is difficult to foresee every execution scenario before execution starts and integrate it in the application design. The execution platform is supposed to be able to adapt to all these run-time situations and make sure the application succeeds to an end.

The main drawback represents the increased vulnerability to faults. The causes are multiple but the main ones distinguish as follows. First there is the computing infrastructure. With the advent of exascale systems, the number of processors is expected to reach the level of millions according to (2). If the building components are not fault tolerant, even the simple applications are prone to failures during their execution. Not only the physical platform must be reliable in order to assure safe communication between computing nodes, but also the applications have to adapt their software

components to the distributed nature of the infrastructure to minimize error effects. Insufficient resilience at both hardware and software level would render extreme scale systems unusable and would prevent scientific applications from completing or obtaining the correct result.

The fault tolerance aspect has been widely studied from the hardware and system point of view. Network failures, systems out of memory, down resources and others are mostly taken care by the middle-ware layer of a computing infrastructure and dealt with techniques like checkpoint/restart, job migration. These techniques experience inadequacies in their traditional form when used for continuously growing exascale systems. The complexity of new scientific applications that these systems are supposed to execute will result in larger amounts of data to be check-pointed and increased sources of faults: soft errors, silent soft errors, transient and permanent software and hardware errors. The application logic becomes important when evaluating error propagation and deciding the recovery strategy. Errors in the design or configuration of algorithms will have serious repercussions in the safety of the execution. If there is a way to correct these errors on the fly, a lot of time and resources will be saved. (2) states that for exascale systems faults will be continuous and across all parts the hardware and software layers, which will require new programming paradigms. Here are some important remarks that sustain the importance of the problem:

- Present applications and system software are not fault tolerant nor fault aware and are not designed to confine errors /faults, to avoid or limit their propagation and to recover from them when possible.
- There is no communication or coordination between the layers of the software stack in error/fault detection and management, nor coordination for preventive and corrective actions.
- There is almost never verification of the results from large, long running scale simulations.
- There are no standard metrics, no standardized experimental methodology nor standard experimental environment to stress resilience solutions and compare them fairly.

As mentioned in the beginning of the previous section, workflow systems have gained a lot of interest from the distributed computing community. They are seen as a viable solution for adapting complex scientific applications to large scale distributed computing resources. But as stated in (3), “current Grid Workflow Management Systems still cannot deliver the quality, robustness and reliability that are needed for widespread acceptance as tools used on a day-to-day basis for scientists from a multitude of scientific fields”. Because of the graph structure of a workflow application, recovery techniques of workflow have attracted enough attention in recent years. Still, (4) claims that several critical issues regarding the distributed recovery have not been addressed like recovery during error occurrence or synchronization. Even though the above cited paper deals with distributed transactional processing systems, the concepts discussed in it can be applied also in scientific workflow systems.

1.3 Proposed Solutions

To address the problems presented until now, or at least a part of them, we propose the development of an execution platform based on YAWL(Yet Another Workflow Language (5)) workflow system, complementary to a resiliency algorithm that is in charge of error detection, error handling and recovery. YAWL was developed based on a rigorous analysis of existing workflow management systems and languages. As the official manual states (6), YAWL is based on one hand on Petri nets, a well-established concurrency theory with a graphical representation and on the other hand on the well known workflow patterns (7). Even though at its origins YAWL is designed to deal with management work scenarios, it contains a set of services in its architecture that allows to address some of the most important issues present in current scientific workflow systems like dynamicity, exception handling or flexibility. The most important such service is the YAWL Worklet Service with its two sub-services: Worklet Selection Service (8) and Worklet Exception Service (9).

The Selection Service provides each task of a process instance with the ability to be linked to a dynamically extensible repertoire of actions. In this way the right action is chosen contextually and dynamically from this repertoire to carry out the task. In YAWL such an action is called a *worklet*, which is a small, self-contained, complete workflow process. The global description of the process is provided at design time

and only at run-time, when a specific task gets enabled by the engine, the appropriate worklet is selected, using an associated extensible set of rules. New worklets for handling a task may be added to the repertoire at any time during execution, as different approaches to complete a task are developed and derived from the context of each process instance. Notable is the fact that once chosen that worklet becomes an implicit part of the process model for all current and future instantiations, allowing a natural evolution of the initial specification. A bottom-up approach to capture contextual data are the Ripple Down Rules (10) which comprise a hierarchical set of rules with associated exceptions.

The Exception Service extends the capabilities of the Selection Service in order to provide dynamic exception handling with corrective and compensatory actions. The Exception Service uses the same repertoire and Ripple Down Rules as the Selection Service. For every unanticipated exception (an event not expected to occur in most instances, so excluded from the main logic) a set of repertoire-member exception handling processes are defined, known as exlets, which will be dynamically incorporated in the running process. An exlet can also contain a compensatory action in the form of a worklet, defined in the same manner as for the Selection Service. Each exception has also a set of rules attached that will help choosing the right exlet at run-time, according to their predicate. If an unanticipated exception occurs (an event for which a handling exlet has not been defined), either an existing exlet can be manually selected from the repertoire, or one can be adapted on the fly, or a new exlet can be defined and deployed while the parent workflow instance is still active. The method used to handle the exception and the context in which it has occurred are captured by the system and immediately become an implicit part of the parent process model. This assures the continuous evolution of the process while avoiding the need to modify the original definition.

The above mentioned services are built-in and ready to use by any user. An increase in flexibility is obtained also through the concept of Custom Service. It is a web-based service responsible for the execution of one specific task, part of a workflow specification. The engine orchestrates the delegation of tasks to different services according to user specification. This service-oriented, delegated execution framework is the extensibility cornerstone of the YAWL system. A communication is established between the engine and the service summarized in the following steps:

- The engine notifies a service that a task is scheduled to be delegated to it.
- The service informs the engine that it has taken responsibility for executing a task.
- The service performs the task's activities, as appropriate.
- The service informs the engine that the task's execution has completed, allowing the engine to continue processing the specification instance and determine the next task or set of tasks for scheduling (if the instance has not completed).

The YAWL workflow system represents the upper layer of the execution platform where the user can model its application in terms of a workflow specification. Once the execution starts, tasks are delegated to custom services. Every custom service communicates with a distributed computing platform on which executes the code associated with every task. The obtained results are then transferred to the engine. During the execution of a workflow specification there are two algorithms activated. One is for saving the state of the workflow at specific intervals according to some heuristics (built-in or user-specified) and also to transfer the saved data from different remote machines . The second algorithm is responsible for resilience. It detects any exception risen by the system using the Exception Service then it evaluates its propagation throughout the rest of the application and in the end tries to diminish its effects so that the application can continue execution. This can be achieved locally but sometimes it can require the recovery of a previous checkpoint. The resilience algorithm emphasizes the propagation and recovery part, since the goal is to contain the damages as soon as possible and perform the recovery phase as locally as possible in order to avoid re-execution of tasks not influenced by the exception that occurred.

We use the Custom Service concept also to connect the workflow upper layer, represented by YAWL, to a distributed computing platform called Grid5000. This platform is composed of several homogeneous clusters grouped by site and interconnected using a high speed network. Its main advantage is the flexibility, being a platform created especially for research purposes, thus presenting less constraints than a commercial or just public grid. The YAWL Custom Service is the link point between YAWL and Grid5000. Each service is installed on Grid5000 and customized according to the type

of tasks it is designed to execute. Since we are working with multidisciplinary applications we can create dedicated services for each discipline involved in the computing process and delegate the work load corresponding to each discipline to the appropriate service.

The tests performed to validate our execution platform are based on numerical optimization applications prepared by our colleagues in Sophia Antipolis. Their level of complexity varies from very simple ones used to test basic features of the platform to large scale applications that need tens of computing nodes located on several clusters in the computing grid.

1.4 Organization of the Mansucript

The rest of this manuscript is organized as follow:

- Chapter 2 gives the reader an overview of what already exists in the two research areas on wich this thesis is based: High Performance Computing (HPC) infrastructures and Workflow Systems. The two concepts are always presented to the reader from fault tolerance point of view, as the basic technology to render our execution platform resilient.
- Chapter 3 focuses on the theoretical aspects of this thesis without going into the implementation details. We describe the structure of a numerical optimization application. Also we present some test-cases from Sophia Antipolis that we use to test our platform and finally we present the exception and exception treatment concepts that we want to implement for such applications.
- Chapter 4 presents all the implementation issues of the platform. We explain the reason for our choice of different components, like Grid5000 and YAWL and also the mechanism of interconnection of these two. We also describe the extensions that we have added to accommodate the optimization applications presented in the previous chapter. Finally we describe our resilience algorithm using different resilience scenarios that could interest a numerical researcher.

- The last chapter concludes the previous chapters and proposes a set of perspectives to improve the work that has been done and also to extend the current platform with new features.

2. State of the art

2.1 Introduction

Fault tolerance is nowadays an indispensable characteristic for the distributed computing infrastructures which have gradually increased in capacity over the last years making their coordination very difficult. Bigger capacity allowed execution of more complex scientific applications, but not every scientist knows how to operate with big computing platforms. This encouraged development of middleware systems that are offering a variety of services from low-level functions like allocating computing nodes individually to high-level functions of application description and configuration, using workflow systems and full infrastructure isolation as it's the case with cloud computing.

In this chapter we will present in detail what type of distributed computing infrastructures are available and how they have evolved in time. Then we will show how these systems are affected by faults and what are the standard techniques to handle them. We will also present the concept of computing workflow systems that have emerged recently as a solution for executing long-running complex scientific applications on distributed infrastructures. The last part of this chapter, an introduction to the basic idea of this thesis, will explain what does resilience mean and which type of exceptions are concerned by this fault tolerance procedure.

2.2 Distributed Computing Infrastructures and Middleware Software

2.2.1 Overview of Grid Computing Systems

The continuous advances in information technology applications manifested at every level like speed, performance and cost. To keep up with these demands, the same advances had to be obtained for the underlying architecture. This explains the evolution of grid technologies regarding their development, deployment and application execution. We mentioned grid technologies, but new distributed systems paradigm have emerged in the last years, like utility computing, everything as a service or cloud computing. Since their characteristics often coincide with those of a grid system and an exact distinction

is hard to be made, we will focus our description only on the grid technologies. A quite complete taxonomy of grid systems is presented in (11) that aims, as the authors claim, to facilitate a study of grid systems under one framework and ease the understanding of their similarities and differences.

2.2.1.1 History

Grid computing systems have evolved rapidly since their first use. At the beginning (in the 90's) the implementation was reduced to only a model of meta-computing where resources were shared inside supercomputers. Then it evolved to the integration of middleware systems (year 1998) that had to glue different grid technologies. The third evolution level (year 2001) concentrated on the fast data transfer and storage request brokers for persistent data storage. Later the web technologies started to be combined with the grid systems (year 2002). The next generation of grid systems defined by experts from the European Commission emphasize the need for grids to support the *Ambient Intelligence* (AmI) vision, where humans are surrounded by computing technologies without being intrusive.

2.2.1.2 Classification

Even though the new generation grid systems are interesting for the future development of grid systems, we will emphasize only those characteristics that we used for our experiments and that are relevant for the type of applications we executed on them. Firstly, grid systems can be characterized by their spanning size. We can include here *global* grids that provide computational power everywhere in the world. To achieve this, they often use the Internet infrastructure so they are also called *Internet* grids. Another type, implemented more often, would be the *national* grids restricting the computer resources only to one country's borders. They are often developed as infrastructures for research experiments, Europe being the leader in building such grid systems. The rest of the categories addresses a much smaller number of users, like enterprises, departments or even personal users.

A different criteria of classification for grid systems is their accessibility level. On one side we have the so called *closed* grids where the structure of the underlying infrastructure is fixed and the nodes are usually stationary. The connection between different machines is often wired and the user can access the grid through well specified

fixed entry points. On the other side we have the so called *accessible* grids in which we include *ad-hoc, wireless and mobile* grids. The *ad-hoc* grids (12) allows a spontaneous association of computing nodes that forms a computing grid, with nodes being able to join or leave at any moment. *Wireless* grids (13) integrate in their composition devices like sensors, laptops or even mobile phones. Depending on their technical specifications these devices can be used for computing purposes, as grid nodes, or just as connecting devices. *Mobile* grids (14) represent an answer to the increased market gain of PDAs or smart phones that combined can offer a considerable computing power.

The level of interaction between the user and the computing infrastructure classifies grid systems into *batch* grids and *interactive* grids. The *batch* type is closer to the traditional grids, where real time interaction is not supported. Users submit their jobs, and then the middleware system handles their execution by exchanging messages between computing nodes through message passing interface (MPI) methods. *Batch* grids use a system of waiting queues in which the submitted applications are buffered before allocation on computing resources for execution. This adds an extra time for the overall execution of an application as seen by the user. Instead, the *interactive* grids allow interactive sessions for the users through which they can control or modify their running jobs.

The traditional grid systems are managed using a centralized approach where the experienced staff has a detailed view of the underlying architecture. It is easy to deploy and control but quite exposed to faults and lacks scalability. A more flexible architecture is the peer to peer (P2P) one. In this case, every node is controlled separately and can join or leave the system at any moment. However, the novelty is represented by the *manageable* grids (15). Aiming complete autonomy, this type of grid is supposed to be able to automatically configure, adapt and control itself with very few human intervention. Some examples of this kind of grid systems are: *Autonomic* grids (IBM OptimalGrid), *Knowledge* grids (OntoGrid, InteliGrid, K-WfGrid) and *Organic* grids.

The current trend is the evolution of grid systems toward service oriented architectures (SOA). The computing resources, remotely shared by multiple users, are now offered with a set of computing services. Features like availability or accessibility are now better quantified for each user and provided more on-demand like services. This

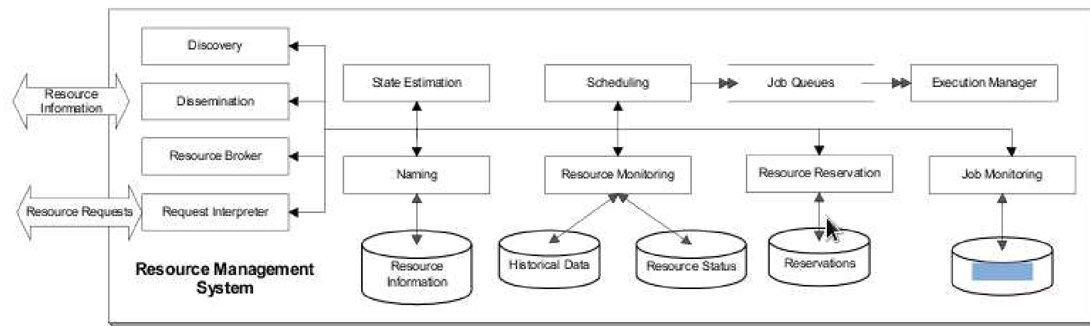


Figure 2.1: Resource Management System Abstract Structure (taken from (17))

produces an increased efficiency in resource utilization, by acquiring them only when a demand is made. To give some examples of categories of such platforms, we will mention *Everything as a Service* (EaaS), *Utility Computing* and *Cloud Computing* (16).

2.2.1.3 User Interaction

We described before the grid systems using general criteria but nothing has been said about the possibilities of an external user to interact with such systems. In order to ensure communication between user and grid system and access its computing nodes, a Resource Management System (RMS) is used. At an abstract level, a RMS provides three fundamental services: resource dissemination, resource discovery and scheduling of resources for job execution (17). In figure 2.1 are described the main requirements of a RMS without specifying the particular machines that implement and provide those requirements. The various ways to implement these characteristics will determine the architecture of RMS systems and also their classification.

A first distinction criteria represents the organization of machines inside the grid. This influences the way machines communicate to each other and can determine the architecture and scalability of the system. For example, in a flat organization every machine can directly communicate with the others in the system. Alternatives to this are the hierarchical organization (different levels with direct communication between neighbor levels) and the cell organization in which machines are disposed in distinct

structures and where machines belonging to the same cell can directly communicate between themselves.

Another characteristic that differentiates RMS types is the resource model. This represents the way a RMS describes the available resources for the user. More precisely, it offers the interface between the user and the computing nodes. The user instead of dealing directly with the machines, will handle meta-resources where data referring to a resource is whether described in a particular language that allows queries to be formulated to address resources, or as an object model where operations on the resources are defined inside the resource model.

Mapping resources to jobs can also be done in various ways. This task is devoted to the scheduler that uses a specific policy to address it. It is the choice of policy together with the scheduler organization that make the difference between RMS systems. The scheduling task can be done by a centralized resource, but this diminish considerably the scalability. The alternatives to this solution are the hierarchical model and the decentralized model. The policy through which a mapping is done usually depends on the estimation of the actual state of the resources in the system. The two main options are the predictive and non-predictive estimation. The techniques to make an estimation are based on heuristics, probability models or machine learning.

2.2.1.4 Programming Models

A grid computing system implies distributed computing resources and also parallel cores sharing the same memory. These resources need to be able to communicate between themselves so that an application is executed in a distributed but coordinated manner. Also, when executed on distributed resources, scientific applications need a different programming approach than the standard one. To adapt application programming requirements (memory and CPU) and map them on distributed resources, organizations from all over the world have established a common library standard called Message Passing Interface (MPI). This allows developers to implement programs or libraries that obey the same specifications thus ensuring a set of features necessary for both HPC applications and systems (like computing grids) (18):

- Standardization: MPI is supported on almost all HPC platforms

- Portability: same source code can be transferred to a different platform with no modifications if that platform supports the MPI standard
- Performance Opportunities: new hardware features can be better exploited for improved performance
- Functionality: hundreds of routines that address every programming paradigm
- Availability

At the beginning MPI was conceived only for distributed memory architectures. With the advances in hardware trends, shared memories started to be used in networked platforms so MPI adapted its standards to handle both types of memory architectures (see Figure 2.2).

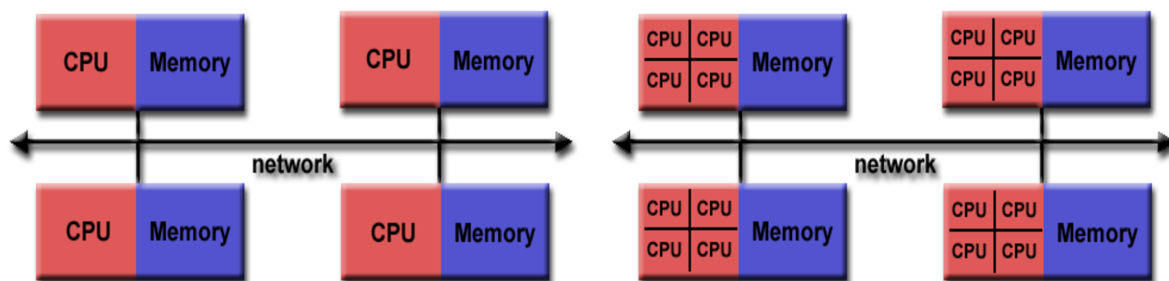


Figure 2.2: Distributed and Hybrid Memory Architecture

In the following sections we will describe distributed computing infrastructures first from a hardware point of view, using Grid5000 as an example, and then from a software point of view by presenting several middleware systems.

2.2.2 Grid5000

2.2.2.1 General View

Grid'5000 (19) is a scientific instrument for the study of large scale parallel and distributed systems. It aims at providing a highly reconfigurable, controllable and easy to monitor experimental platform to its users. 17 laboratories are involved in France with the objective of providing the community a testbed allowing experiments in all the

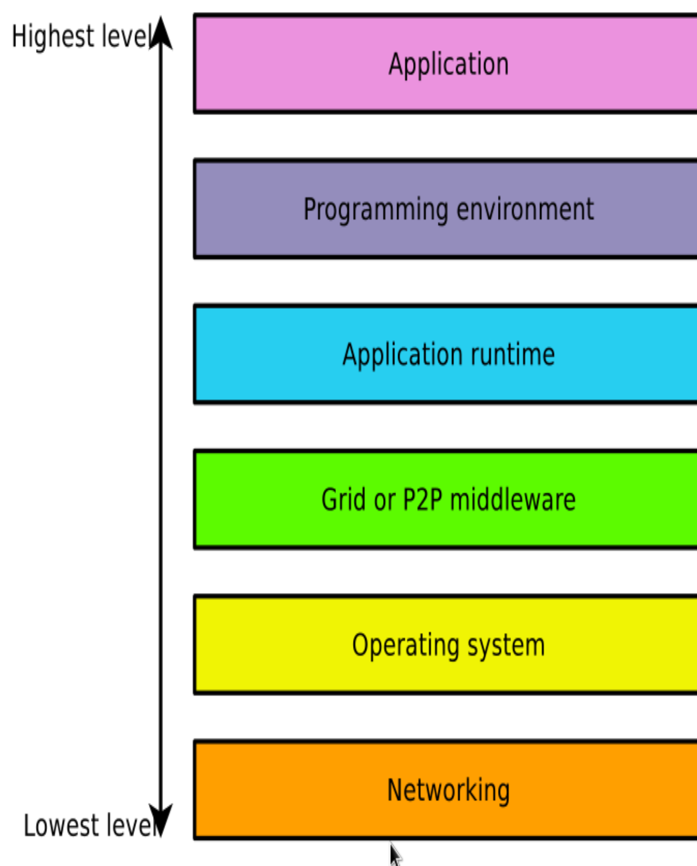


Figure 2.3: Grid'5000 Research Applicability

software layers between the network protocols up to the applications (see Figure 2.3). In addition to theory, simulators and emulators, there is a strong need for large scale testbeds where real life experimental conditions hold. The size of Grid'5000, in terms of number of sites and number of processors per site (9 sites, 7244 coeurs), was established according to the scale of experiments and the number of researchers involved in the project.

The platform is devoted to experiments from various fields that suppose an open access to the computing resources. Thus users are required to respect some basic rules concerning resource usage, like avoiding to occupy the platform for a long time

or respecting the privacy of other users without abusing the low security level of the nodes. When deploying an experiment on the platform there is a set of steps one must always follow: connecting to the platform through one of its sites only, reserving the necessary resources for his application, configuring the resources if necessary, running the experiment, retrieving the results and free the resources.

2.2.2.2 Architecture

Beside the 9 sites in France, Grid'5000 has connected two extra sites in Luxembourg and Porto Alegro(Brasil). The general topology is pictured in Figure 2.4. Every user that wants to use Grid'5000 needs an account to connect to the platform. Basic knowledge of SSH are needed to use Grid'5000 as this is the technology used for connection. On every site a user owns a home directory that is shared through NFS with all the component clusters (see Figure 2.5). This is not the case with two home directories belonging to different sites. In this situation, the user is responsible for synchronizing the data between directories. As a consequence, a user will have as many home directories as sites in the platform.

2.2.2.3 Tools

The interaction with Grid'5000 platform requires the use of different software tools. Some of them are standard tools, not specific to Grid'5000, like SSH. Others were specially developed and supported by Grid'5000 staff like OAR, taktuk, KAAPI etc. We will present two of them that support a basic usage of Grid'5000 platform.

1. OAR (21)

   Represents the resource manager for Grid'5000 that allocates resources to users for their experiments. It allows them to create jobs on the platform's sites by blocking a certain amount of resources for a desired period of time. Each Grid'5000 site has 1 OAR resource manager. We will further describe OAR when we will talk about middleware systems in the following section.

2. Kadeploy (22)

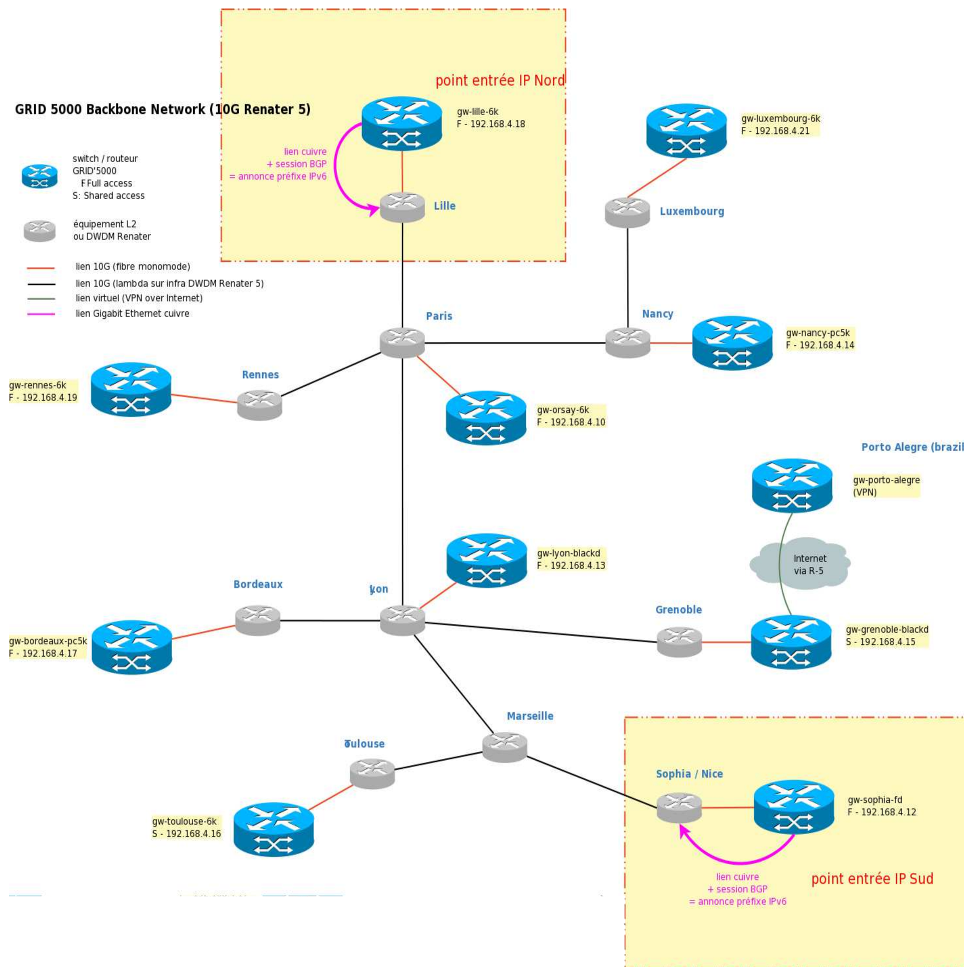


Figure 2.4: Grid'5000 Topology (taken from (20))

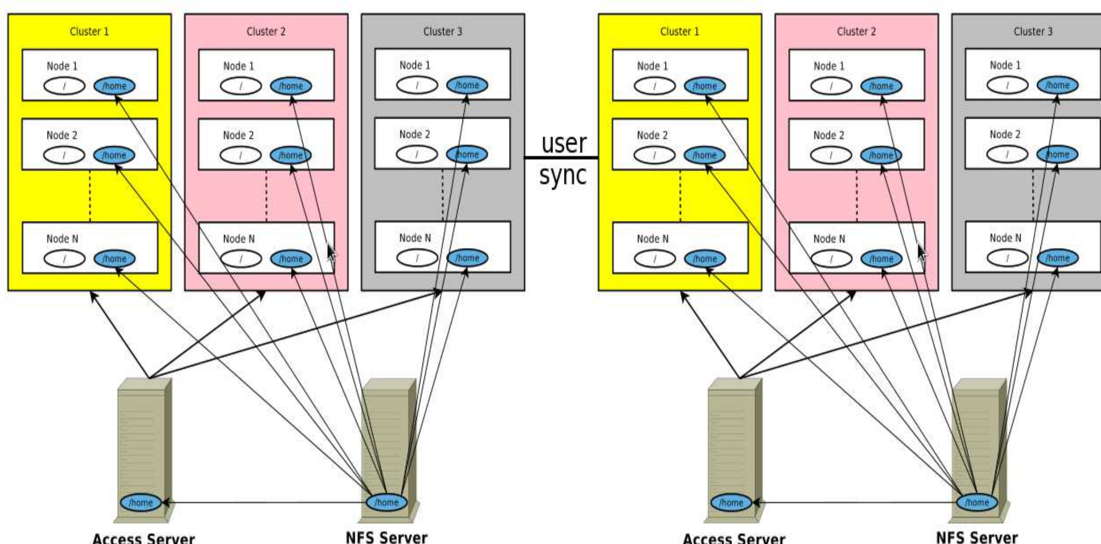


Figure 2.5: Shared Home Directory on Grid'5000

By default Grid'5000 nodes are running with a given operating system based on GNU/Linux. This environment provides basic functionality for running jobs but forbids any modification or new installation. Applications submitted on Grid'5000 vary a lot in requirements so very often the default functionality is insufficient. The purpose of Kadeploy is to give opportunity to users to change the default environment installed on the nodes with a customized one that meets their requirements.

The Grid'5000 staff offers a set of reference environments with a kernel supported on any type of hardware present on the platform. A user can start from such an environment and customize it. After finishing he has to register the new environment in a database along with the default ones. Having your own environment presents some advantages like:

- installing whatever libraries you need for your applications
- connecting as root on the nodes

- reproducing the experiments without being affected by default system updates performed by Grid'5000 administrators

2.2.3 Middleware Systems

Resource management is essential to constructing and using computational grid systems. It is implemented inside middleware platforms providing the necessary level of abstraction and services to facilitate the design, development, integration and deployment of distributed applications in heterogeneous computing systems. A middleware platform involves integration of multiple systems like databases, networking, operating systems, programming languages and others. When designing a middleware system two main approaches co-exist nowadays (23):

- A first approach presents to the user a uniform view of the resources and it is the user's responsibility to handle the heterogeneity of these resources when designing his applications. This means that the system ensures communications between nodes but leaves to the user's responsibility the choice of nodes according to his application's requirements.

- The second approach is more restraining in transparency. Users submit jobs to computational servers that offer specific computational services. Also known as Application Service Provider (ASP), this model offers for free or using a charging system, computational resources to users, like Internet providers offer network access to clients.

The main difference between the two systems is the granularity level. In the first case the user is able more or less to adapt the granularity of his application to the available resources. In the second case, the granularity is quite coarse but addresses a wider user public since no advanced knowledge about distributed systems are required. We will exemplify these concepts by presenting four middleware systems: OAR and Globus as part of the first category described earlier, ProActive and Diet as part of the second category.

Our research interests have focused on ProActive and OAR (as part of Grid5000 platform), being the reason for which we give a more detailed presentation in the following. Even though ProActive was a complete middleware with a wide range of services included, we chose Grid5000 with OAR since it answered better to our need for low-level configuration of computing nodes.

2.2.3.1 Globus Toolkit

Globus Toolkit is a software project that aims to support development of distributed computing infrastructures with all the required services for easy user access. These services include security, resource access, resource management, data movement, resource discovery and others. The Globus Toolkit is destined to all kind of distributed resources like computers, storage devices, services, networks or sensors. The idea is to federate all these resources in one infrastructure and provide necessary tools to design and implement applications that will be executed on this infrastructure (24). In figure 2.6 is presented the general architecture of the Globus Toolkit which is basically a service oriented architecture (SOA). We can see that is formed of a set of service implementations that represent the core of the infrastructure, taking care of execution management, data movement, monitoring, discovery and so forth. There are also three containers used to host user-developed services written in different programming languages like Java, Python or C. At the upper layer there are the client libraries that allow client programs in Java, C and Python to invoke operations on both core services and user-developed services.

2.2.3.2 Distributed Interactive Engineering Toolbox (DIET)

(25)

A representative example of an Application Service Provider middleware system is DIET. It is a toolbox for developing ASP systems on Grid platforms based on a Client/Agent/Server scheme. The user requests for resources are shared among a hierarchy of *Local Agents* and *Master Agents*. Figure 2.7 represents the organization of this agents hierarchy. The different agents have a particular role and interact with each other to assure the proper computational services for users. Thus, a client is the application that connects the user to DIET from a web page or from a compiled program.

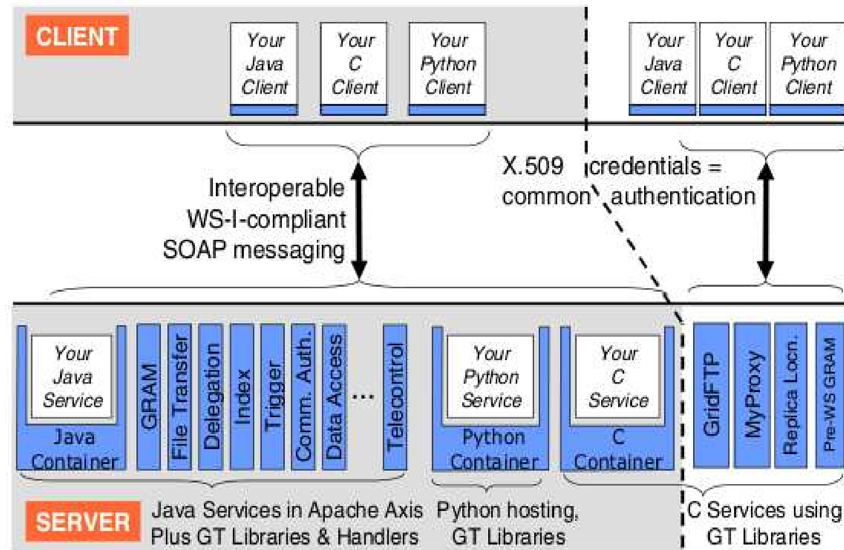


Figure 2.6: Globus Architecture (taken from (24))

The *Master Agent* (MA) accepts computation requests from the clients and contacts the servers for computational abilities and chooses the best one. It then returns a reference of the chosen server to the demanding client. A *Local Agent* (LA) transmits requests and information between MAs and servers. Finally, a *Server Daemon* (SeD) incorporates a computational server and hosts a set of information regarding problems that can be solved on the server, available memory or computational power.

2.2.3.3 OAR

OAR is batch management system used on Grid5000 platform for resource management. In this role it handles very well the scalability of the platform and its heterogeneity. For instance the latest version of OAR uses the Linux kernel feature called *cpuset* that helps identifying which resource can be used by a particular process.

OAR is also appreciated for being able to manage all types of complex resource hierarchy. Thus inside a grid infrastructure, OAR can configure resource structures like cluster, switch, host, cpu and individual cores. This means that it can isolate a process up to the core level which improves resource usage especially when dealing with multi-core machines. Another important feature is the grid resources interconnections which

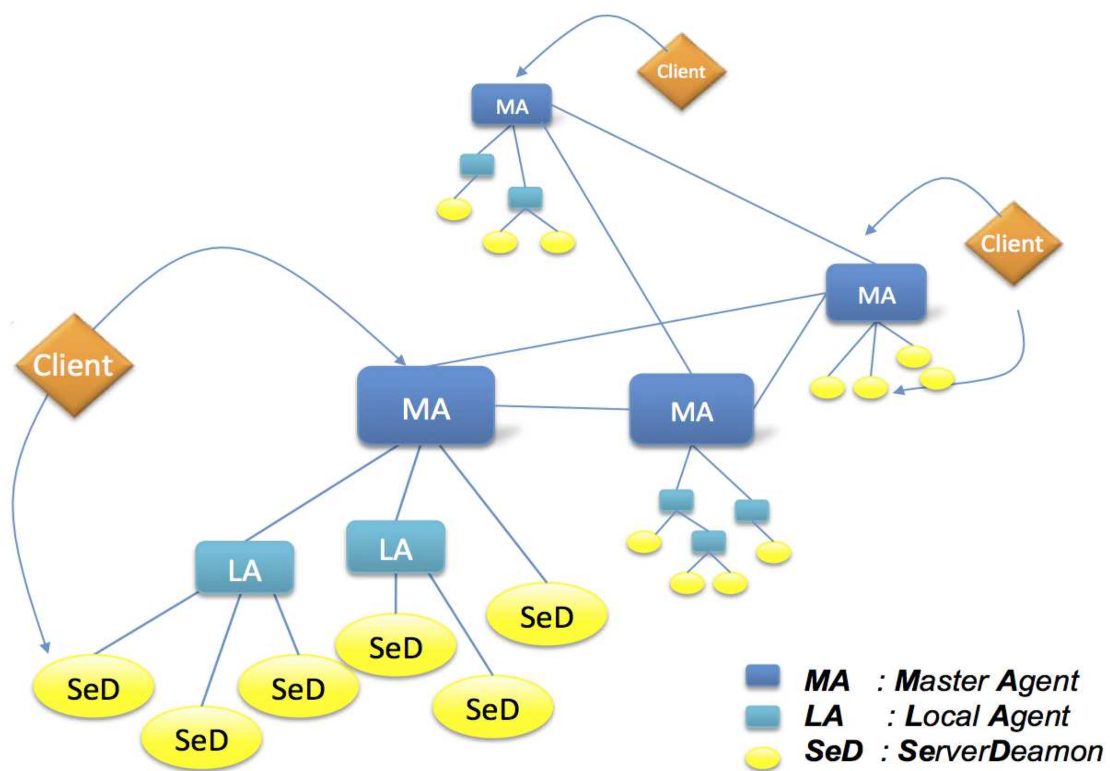


Figure 2.7: Hierarchy of DIET Agents (taken from (23))

facilitates communication between jobs, clusters or resource sites (26, 27). Among the features OAR provides to users we mention:

- *interactive jobs* - instant resource reservation for a specific amount of timeout
- *advanced reservations* - resources are reserved at a given date for a given amount of time
- *batch jobs* - a script is associated to a job that will run in background
- *best effort jobs* - resources can be released at any moment for a more efficient usage
- *deploy jobs* - a customized OS environment can be deployed on the allocated resources with full access

Beside this, OAR allows also to visualize a reservation, check its status, cancel a reservation or verify the status of the nodes.

2.2.3.4 ProActive

ProActive (PA) is an open source middle-ware software presented as a Java library, aiming to simplify the programming of multi-threaded, parallel and distributed applications for Grids, multi-core, clusters and data-centers. With a small set of primitives, ProActive provides an API allowing the development of parallel applications which can be deployed on distributed systems using the deployment framework. ProActive doesn't require any modification to Java or to the Java Virtual Machine, therefore allowing the deployment of applications using ProActive API on any operating system that provides a compatible JVM. In the rest of this subsection we will concentrate on the deployment framework with its two main components, PA Scheduler and PA Resource Manager.

1. *ProActive Scheduler* (28)

   Executing parallel tasks on distributed resources, requires a main system for managing resources and handling task executions, also known as a batch scheduler. The scheduler enables users to submit jobs, containing one or several tasks, and then to execute these tasks on available resources. The ProActive Scheduler

is connected to the Resource Manager which provides resource abstraction. The Scheduler is accessible either from a Java programming API or a command-line based job submitter.

In ProActive Scheduler a job is the entity to be submitted to the scheduler, composed of one or more tasks. A task is the smallest schedulable entity, and will be executed in accordance to a scheduling policy on the available resources. There are two types of tasks:

- Java Task its execution is defined by a Java class extending the JavaExecutable class from the ProActive Scheduler API.
- Native Task its execution can be any user program, a compiled C/C++ application, a shell or batch script; a native task can be specified by a simple command line, or by a generation script that dynamically generates the command to be executed.

By default the Scheduler will schedule the registered jobs according to a FIFO policy, but if a job needs to be executed faster one may increase its priority or contact the Scheduler manager. A job can be created using an XML descriptor or the provided ProActive Scheduler Java API. When creating a job, one can specify several parameters like: *name, priority, cancelJobOnError, restartTaskOnError, nbMaxOfExecution, logFile, variables, genericInformation, JobClasspath, inputSpace* (an URL representing an abstract (or real) link to a real data space), *outputSpace*. Similar to a job, a task can also have different parameters, some of them being identical to those for a job. Some parameters specific to a task are: *Walltime* (timeout), *parameters* (to be transferred to the executable), *numberOfNodes*, *scripts*, *selectionScript*, *pre/post script* (to be executed before and after the executable), *cleaning-script* (executed after the executable or post-script).

2. *ProActive Resource Manager* (29)

The Resource Manager is an entity in charge of nodes acquisition/release from particular underlying infrastructures. It consists of two components: *infrastructure manager* and *node source policy.*

The *infrastructure manager* is responsible for communication with an infrastructure, having three default implementations: Default Infrastructure Manager (used with the ProActive agent), GCM Infrastructure Manager (able to acquire/release nodes described in the GCM deployment descriptor), GCM Customized Infrastructure (can deploy/release a single node to/from the infrastructure).

Node source policy is a set of rules and conditions which describes when and how nodes have to be acquired or released. Policies use node source API to manage the node acquisition. There are 4 policies implemented which should cover the most common scenarios: static node source policy, time slot policy, *release when scheduler is idle* policy, *scheduler loading* policy.

New infrastructure managers or node source policies can be integrated into the Resource Manager as plug-ins, like SSH Infrastructure (a basic but effective way to acquire resources through an SSH connection), PBS Infrastructure (acquires resources on an existing PBS installation). Beside this, the Resource Manager also supports the integration with the Amazon EC2, but more importantly, the integration with a virtual infrastructure. Such an infrastructure runs a virtual software and then can be used as a resource pool for Resource Manager (RM) execution. RM nodes belonging to a virtual infrastructure are acquired in the following way:

- Contact the virtual machine manager for powering on the virtual machines that will be used to run RM nodes.
- Start the RM nodes this step requires the retrieval of the information provided in the previous step.
- Register RM nodes done either by remote or local node registration.

There are several types of virtualizing software that Resource Manager can handle, like VMWare products, XenServer or xVM VirtualBox.

ProActive has been the initial middleware choice for our platform. Unfortunately we gave up this idea when we couldn't use the virtualization system to simulate a cluster platform before passing to a real infrastructure. A second reason was the overhead created by superposing ProActive onto Grid5000 (the physical infrastructure we planned to use) that already has its own middleware, which is OAR, even if it offers less features to manage the computing resources.

2.3 Fault Tolerance Methods in Distributed and Parallel Systems

Despite their usefulness for executing long-running applications, distributed systems are inherently affected by faults. Whether we talk about hardware or software faults, when they occur the correct execution of the application is endangered. This can manifest as a complete stop of the system or just by producing wrong results.

2.3.1 Faults - General View

There are three main levels from where faults can be generated and those are the human failures, soft faults and hardware faults (30). Of course these main levels support variations and to classify them the literature uses models of failures as presented in (31):

- Byzantine Faults : They are the most difficult to deal with, since the part of the system that functions correctly is not able to detect them. Also it accepts input from nodes affected by them, thus spreading rapidly the error in the entire system.
- Fail-stop Faults : A node affected by such a fault ceases to produce output and stops any interaction with the environment. This makes it easier for the system to detect the fault and to take needed measures to overcome it.
- Fail-stutter Faults : This model is an extension of the Fail-stop model, considered too simplistic. Beside the fail-stop faults, this model includes also the so-called *performance* faults. When such a fault occurs, the affected component continues to function correctly regarding its output, but provides unexpectedly low performance.

Figure 2.8: Fault Causality Chain (taken from (33))

What makes a distributed system stop functioning correctly is not the fault itself but the effect it produces on the system. Figure 2.8 shows how a fault initiates the dis-functionality in the system, propagating until the failure occurs that will trigger other faults in turn (32).

A distributed system (34) can be abstracted to a set of processes communicating through messages using communication channels. This model of communication is prone to failures of different kind. If a message is not delivered the waiting process can stop or continue with an omitting message which can lead to wrong results. Also, giving the distributed nature of the applications, if one machine participating at the execution fails it will affect the results of the whole application. Without a global watch for synchronization, the coordination between processes is done using the following models:

- *synchronous* - the time allocated for exchanging messages is limited.
- *asynchronous* - the communication channel handles delivery of messages but does not limit the time to do that; in presence of faults, the coordination between processes is affected.
- *asynchronous with fault detection* - this model assures consensus between processes in presence of faults.

Before a system reaches a failure state because of a fault there are different methods to avoid the fault occurrence. For example one can use fault prevention, especially when we deal with development faults, by respecting development rules like modularization, string typing, etc. A system developer can also try to reduce the number of faults both during development and maintenance of the system. A very popular method nowadays is fault prediction that aims at estimating the occurrence and consequence of faults using system modeling and evaluation. A very interesting study has been done in (35) where the authors have analyzed failures contained in five years of event logs from a

production high performance computing cluster. Based on this they have determined the distribution of failure inter-arrivals of specific components of the computing system (CPU, disk storage, etc.). This allowed them to build holistic failure models based on the component-usage of applications which they applied to derive the optimal time to checkpoint.

These methods do not aim at eliminating faults completely (36, 37) and in a large distributed system, like computing clusters, grids or high performance computing systems (HPC), the presence of a fault tolerant mechanism has become indispensable.

2.3.2 Fault Tolerance Techniques

The use of large scale distributed systems is justified by the increased demand in resources from application domains like numerical optimization and simulation. As we have seen before it is impossible to fully eliminate faults from such systems, so the only way to ensure correct results when executing applications is to use fault tolerant mechanisms.

2.3.2.1 Fault Detection (1)

Before dealing with faults/failures, the system must be able to detect them first. Unfortunately there are also undetected faults called *silent* for which there are only a few ways to deal with (38):

- ignore them assuming that their impact is not crucial for the final results
- develop resistant algorithms that, by construction, can deal with a high level of undetected faults (39)
- use redundancy and on-line checking
- store, transfer and compute on an encoded version of data

The detection of a fault can be done while executing the application (concomitant) or by interrupting the execution service (preemptive). In the first case there is always the need of some kind of redundancy that can manifest under various forms: error code correction, duplication and comparison, watchdog, type checking, etc.

2.3.2.2 Fault Recovery

As shown in Figure 2.8 a fault usually triggers a failure of a part or the entire system. After detection, the goal is to bring the system in a coherent state that allows it to restart execution. Of course this requires a mechanism for fault treatment. Nowadays there are three main mechanisms for treating faults:

1. Failure correction - The system tries to repair the error and reach a coherent state from which it can continue execution.
2. Compensation - This method is used both for fault detection and recovery of the system and it requires an important amount of redundancy.
3. Checkpoint/Restart - The coherent state of the system is saved periodically in a secure space. When a failure occurs the execution will restart from a state prior to the failure.

In (40) the fault tolerance methods presented above are classified according to the way they relate to the failure from a temporal point of view. The first class is represented by the *failure avoidance* methods that take preventive actions before failures occur. The second class is called *failure effect avoidance* where the application execution is continued until its termination, even if failures have occurred. A sort of compensation mechanism, either at run time or at algorithmic level, ensures that the execution terminates and the correct results are delivered. In this class we can include techniques like replication or algorithm based fault tolerance methods (ABFT). The third class is represented by *failure effects repair* techniques that consists of repairing the effects produced by the failures. Despite their usefulness, the preventive fault tolerance techniques are not developed enough to be used independently. More research is required to prove that their benefits outcome their disadvantages. This is why the most studied and implemented methods are those treating the failures generated by faults when a prevention has not been possible.

2.3.2.3 Redundancy

All the recovery methods previously presented demand a certain level of redundancy. This can be expressed in a spatial dimension when components of the system (usually

processes) are replicated (compensation) and then their results compared for validation. Redundancy can also be expressed in a temporal dimension when a critical part of a process is treated multiple times to ensure again the obtaining of correct results. Last, we have the informational redundancy when data, code or other type of information is stored on stable memory devices (checkpoint/restart). In the following we will focus on describing different mechanisms to implement the checkpoint/restart procedure and all the coordination protocols that come with it.

2.3.3 Checkpoint/Restart

Checkpoint/Restart is a traditional and efficient fault tolerance mechanism for distributed systems and applications. It consists of periodically saving the state of a running system on a stable storage so that when affected by a failure the system can restart execution from a previous consistent point using a rollback procedure. The checkpoint/restart procedure is quite complex and has some weak points whose effects every implementation tries to diminish (40):

- Saving the state of a distributed system is often a difficult task.
- The time to save the execution state can become very large.
- The time interval between two consecutive checkpoints must be well chosen to avoid spending more time for saving the state of the execution than for the actual execution.

2.3.3.1 Globality

Saving the state of a running system translates into saving the state of each component process independently along with the messages found in the communication channels. The state of the system is considered consistent if no orphan message is present in the collection of processes. Such a message is described by the existence of a receiving process but the absence of the sending one. When recovering after a failure, a system should rollback to the most recent consistent state, also called a recovery line (see Figure 2.9). The condition of consistency is difficult to achieve since it is not possible to implement a global clock for coordination. In the following section we will present the advantages and disadvantages of two important checkpoint protocols.

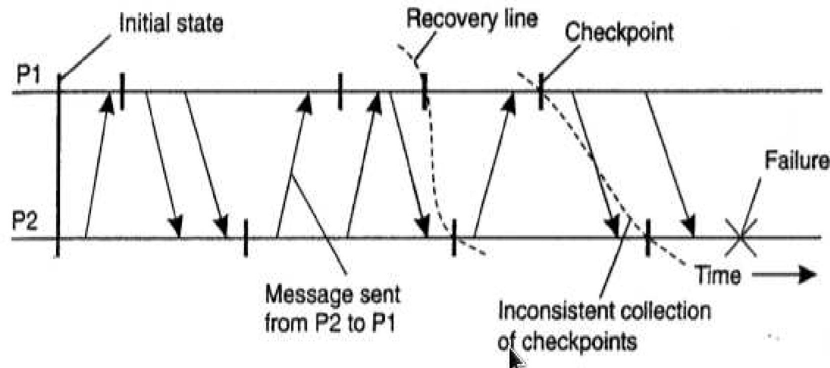


Figure 2.9: Recovery Line (taken from (34))

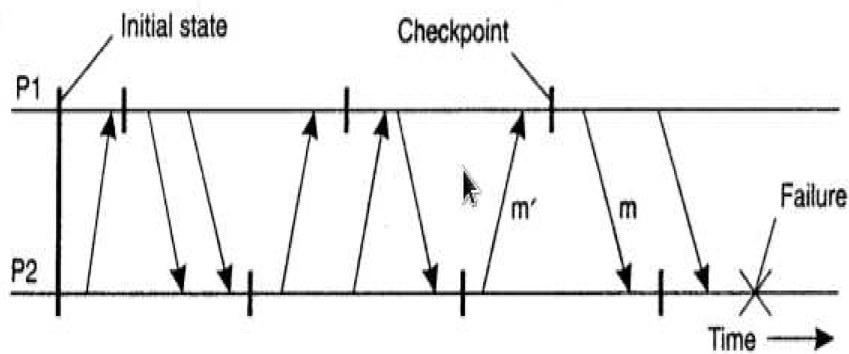


Figure 2.10: Domino Effect (taken from (34))

2.3.3.2 Independent vs Coordinated Checkpoint

One possibility of saving a system's state is by independently saving the local state of each process in the system, from time to time, with no coordination between them (41). This approach is preferred especially for large distributed systems where the time effort of coordination would be too high. The downside of this protocol is the difficulty of reaching a recovery line when roll-backing the application. Because each process took an individual checkpoint regardless of the other processes in the system, its saved state might be inconsistent with the others' states which forces the rollback procedure to choose a different saved state further back in time. This can generate the so called *domino effect* in which no intermediary consistent state is found so the application is roll-backed at the beginning (see Figure 2.10).

The second checkpoint protocol is the coordinated protocol (42). In this case all processes have to synchronize before saving their state on the stable storage. The advantage of this method is that the consistency of the state is already assured when the checkpoint is performed so when a failure occurs the application will rollback to the most recent checkpoint, thus avoiding the domino effect. The downside is the extra time needed for synchronization, that for a large system can affect considerably the execution time.

The research community keeps an active interest on improving these two protocols. The most common alternative to a global checkpoint protocol is the incremental checkpoint. In (43) the authors propose a model that alternates global checkpoints with incremental ones. The first checkpoint is a global one saving the entire data along with the stack of the application. What follows is a set of incremental checkpoints which only save the address spaces that have changed since the previous checkpoint. Another full checkpoint will be performed if its cost is cheaper than the recovery cost for an incremental checkpoint.

Another approach for optimizing checkpoint performance is proposed in (44). The authors want to automatically adapt a fault-tolerance protocol to the minimal requirements of an application. The specialization of the protocol is done at the level of an abstract representation of the execution which permits important optimization ar run time. Thanks to this it is possible to compute the strictly required set of computation to resend messages to the failed processors.

2.3.3.3 Message Logging

A special type of a system model called *piecewise deterministic model* allows reducing the frequency of checkpoints. In such a model the execution is assumed to take place as a series of intervals and controlled by events. Inside an interval every event is deterministic but the beginning of each interval is triggered by a non deterministic event, such as the receipt of a message. Given these characteristics, a system can register only the non deterministic events and just replay the deterministic ones inside an interval to reach a desired consistent state. This method of recovery is called message logging (34, 45). An important detail is that every process in the system will store its own non deterministic events and log the deterministic ones, so when a failure occurs,

only the faulty processes will rollback and replay the logged messages to reach again the consistent state before the failure (40).

2.3.3.4 Multilevel Checkpoint

For some type of HPC systems it is preferred a multi-level checkpoint strategy, combining local and global checkpointing for enhanced reliability. This means that each computing node is equipped with a checkpoint system that saves the node's state on the local storage but at the same time there is a global checkpoint system that usually saves the entire application state in a parallel file system. The global checkpoint is used only when local recovery of a node's failure is not possible due to loss of checkpoint data. In (46) is presented an extended version of a multi-level checkpoint system. Beside the local and global checkpoints there is an intermediary level based on topology-aware Reed-Solomon encoding scheme that is used to encode the checkpoint files. This can be later used to recover a checkpoint when all its files have been lost due to some hard failures. This way the global parallel file system checkpoint is less stressed during the execution which improves the global execution time.

2.3.3.5 Checkpoint Storage

The performance of saving the state of an application is very much influenced by the type of storage used to backup the data. The most important is that the information should be safely stored. In this regard the most effective system is the stable storage which is designed to survive anything except calamities. The main idea behind stable storage is to have multiple disk copies of the same data and periodically update it on all the copies (34).

In (47) is contested the habit of saving the state of an application into a single shared file because of the incapacity of the file system to cope with a lot of small writes. This results in bad performance for the file system which in general is optimized for large, aligned writes to non-shared files. To solve this issue the authors propose a virtual parallel log structured file system called *PLFS*. Its role is to remap an application's preferred data layout into one which is optimized for the underlying file system.

An interesting approach aiming to eliminate the overhead of checkpointing on a stable storage is proposed in (48). The goal is to remove stable storage when saving the state of a parallel and distributed system by replacing it with memory and processor

redundancy. The diskless checkpoint is performed as a coordinated checkpoint where all the processors synchronize before saving their state into the memory and not on disk. After doing this, the in-memory checkpoints are encoded and stored in *checkpointing processors*. The main disadvantages of this method is that it reduces the failure coverage compared to a disk-based checkpoint and introduces memory and processor overhead.

The utility of a checkpoint strategy for Infrastructure-as-a-Service(IaaS) cloud computing is studied in (49). They propose first to build a dedicated checkpoint repository using the local disks of computing nodes. This will enhance performance under concurrency for read and write operations. At the same time this scheme has a great potential for scalability, since the storage repository will grow when more computing nodes are used. Along with the repository solution they propose to perform a sort of incremental checkpoint to save the state of the infrastructure, that in their case translates into virtual disk images from every deployed node. Finally they tackle also the problem of restart delay that every system encounters after performing a checkpoint recovery. Their solution is to optimize this delay by using a lazy transfer and adaptive prefetching. This is possible because a virtual machine instance typically access only a small fraction of the virtual machine image throughout their run-time.

2.4 Workflow Systems

2.4.1 General View

First used as business process management tools or in the field of bio-informatics for DNA sequencing (50, 51), workflow systems have gained a significant importance in the field of high performance computing. A main reason for which they have been taken in consideration is the increased complexity of the scientific applications. This complexity translates both in the application structure and in a growth in the demand of computational resources. Also the control and management of these applications become harder to achieve, this being in a direct relation with an increased volume of data. Workflow systems are considered to be the appropriate tools to accommodate these requirements and simplify the scientist's job by allowing him to focus more on the scientific aspect of the application. They allow the user to describe the scientific process, organize it in tasks and execute it to create an output (52, 53, 54, 55, 56, 57, 58, 59).

2.4.1.1 Basic Components of a Workflow System

The *Workflow Management Coalition* (60) gives a set of definitions for concepts related to workflows. The most important are presented below:

- Workflow - "Is concerned with the automation of procedures where tasks are passed between participants according to a defined set of rules in order to achieve or contribute to an overall goal."

- Workflow Management System - "A system that defines, creates and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with process participants and, where required, invoke the use of IT tools and applications."

- Process Definition - "A representation of a process in a form which supports automated manipulation, such as modeling or enactment by a workflow management system. The process definition consists of a network of activities and their relationships, criteria to indicate the start and termination of the process and information about the individual activities, such as participants, associated IT applications and data."

- Activity - "A description of a piece of work that forms one logical step within a process. An activity may be a manual or automated. A workflow activity requires human and/or machine resources to support process execution."

- Instance - "The representation of a single enactment of a process, or activity within a process, including its associated data. Each instance represents a separate thread of execution of the process or activity, which may be controlled independently and will have its own internal state and externally visible identity, which may be used as a handle, for example, to record or retrieve audit data relating to the individual enactment."

- Transition - "A point during the execution of a process instance where one activity completes and the thread of control passes to another, which starts."

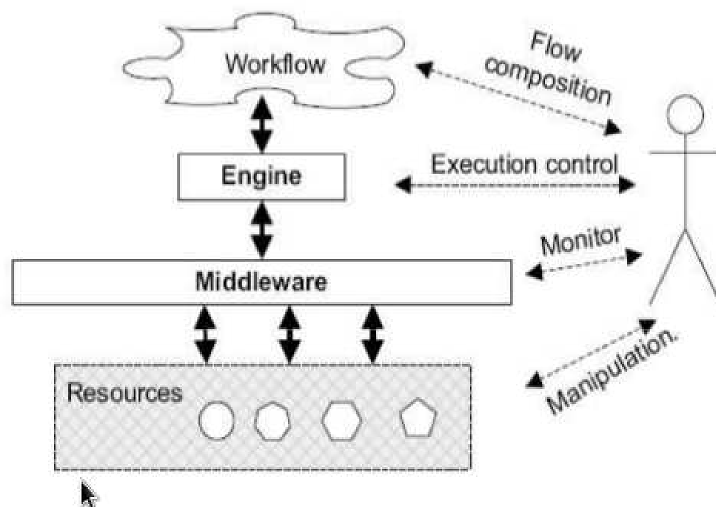


Figure 2.11: Main Components Of a Grid Workflow System (taken from (61))

The definitions given above characterize workflow systems in general. However, no matter how a system is adapted for a specific domain, these concepts will be present under various names but with the same significance. An intuitive domain of application is the scientific one. Here a *workflow* can be seen as a set of nodes, or *tasks* (a.k.a. "activity") interconnected by directed links (a.k.a. "transition"), representing data or control *flows/dependencies* (52). Figure 2.11 describes the basic architecture of a grid workflow system. It emphasizes the main idea of a grid workflow approach, which is to separate the process description of a scientific experiment from the system responsible with the execution.

By analyzing the basic components of a workflow system we can deduce some of the ancestor systems from which it was inspired. Thus, a lot of workflow systems base their internal mechanism on Petri nets. These were defined in 1962 by Carl Adam Petri as a tool for modeling and analyzing processes. Among the attributes of this tool we distinguish three important ones: graphical description of a process, strong mathematical basis, complete formalism (62). Also in (62) is explained that such a formal concept allows a precise definition of a process. Thus, unlike other schematic techniques, Petri net formalism avoids ambiguities, uncertainties and contradictions.

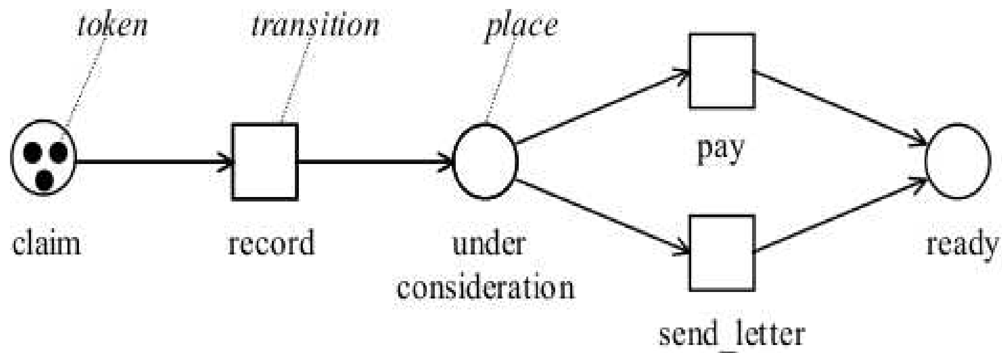


Figure 2.12: A Classic Petri Net (taken from (62))

A classic Petri net is represented by *places* and *transitions.* These two entities can be linked together by a *directed arc.* Arcs can be directed from a *place* to a *transition* and vice-versa. Also a *place* can have tokens represented by black dots. The distribution of these tokens in the Petri net can change according to the state of the process. *Transitions* are able to change the state of a process by moving tokens from one *place* to another when enabled. Figure 2.12 shows a classic Petri net modeling the process of dealing with an insurance claim.

Other systems that present similarities with a workflow system are the *Finite State Automata* and *Grafcet.*

- A *Finite State Automata* (63) is a device that can be in one of a finite number of states. An important subset of states are the final state. If the automaton is in a final state we say that the input, a sequence of symbols, was accepted. The interpretation of the symbols depends on the application, most of the time representing events. The symbols belong to a finite set of symbols called an *alphabet.* If a particular symbol in a particular state triggers a transition from that state to another one, that transition is labeled with that symbol. The labels of transitions can contain one particular symbol that is not in the alphabet. A transition is labeled with (not present in the alphabet) if it can be traversed with no input symbol 2.13.

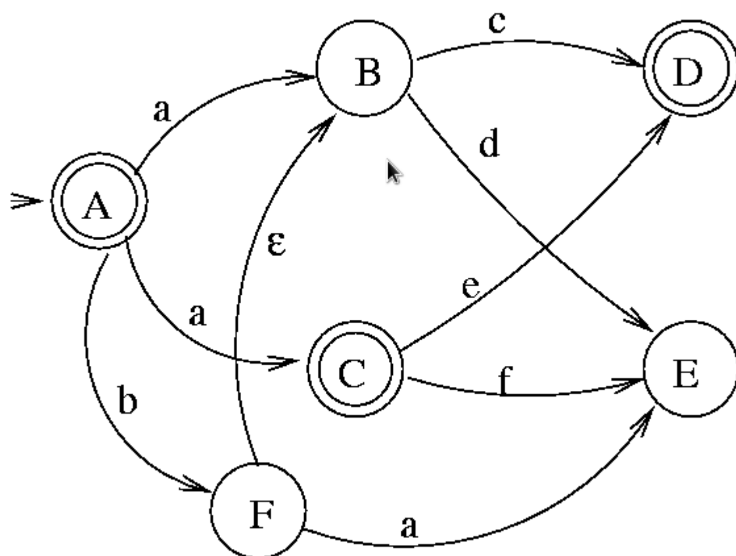


Figure 2.13: Simple Finite State Automata (taken from (63))

- A *Grafcet* (64) is a way of representing the analysis of an automat, well suited for sequential evolutionary systems, i.e. decomposable in stages. It derives from the Petri net mathematical model. Thus it is a graphical language representing the functioning of an automat by a set of:
  - stages to which actions are associated
  - transitions between stages to which conditions of transition are associated
  - directed links between stages and transitions

Returning to the scientific domain, the execution of a scientific application goes through different stages depending on the level of description of the workflow associated with the application. These stages form the life-cycle of a workflow. According to (52) most of the authors identify three different levels in the description of a workflow:

- Abstract workflow - At this level the user only defines the structure of the application, specifying all the composing tasks and how they are interconnected. However, there is no detail about how the tasks are implemented or how the input data is delivered.

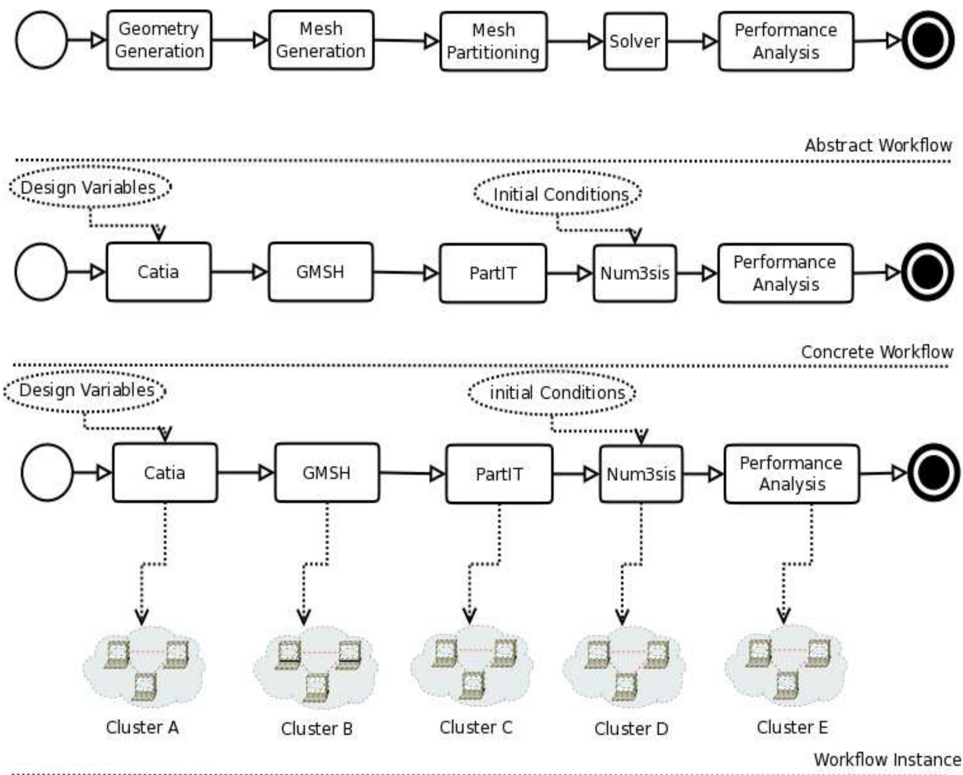


Figure 2.14: Workflow Model

- Concrete workflow - At this level the user defines the input/output parameters and also the software tools used for every task's execution.
- Workflow instances - At this stage the user specifies the actual values of input parameters. He defines also the mapping of tasks on computing resources.

These levels can be visualized in figure 2.14. In practice these levels are not always respected and sometimes they are even considered restrictive so they are partly ignored at design time. Still, it is important to define them at a theoretical level so that designers consider them as starting points from which they adapt their real systems.

2.4.1.2 Models of Abstract Workflow: Control vs Data Driven

In the case of abstract workflow we distinguish two main models: control-driven and data-driven workflows. The difference between these models is made by the signification given to the links between tasks. In the control-driven approach these links are used

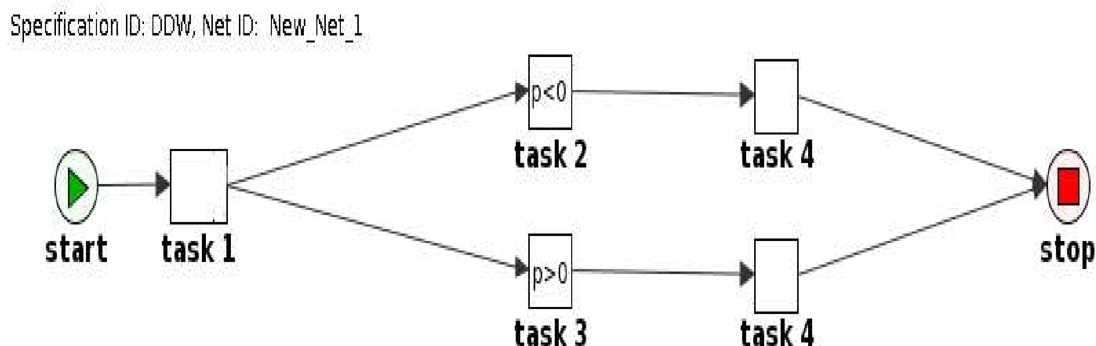


Figure 2.15: Data Driven Workflow

to control the sequence of execution of each task with respect to that of previous tasks. With the expansion of application domain, the control structures evolved in time according to the applications' specificity. We can find simple control structures like pure *sequences* or more complex types like *splits, joins, loops, etc.*. To describe every control behavior that can appear in a workflow application the workflow patterns (65) have been proposed and we will describe them in more detail in the next section.

In the data-driven approach the links between tasks are actually data dependencies. In this model a task is considered an activity that consumes input data and produces output data. This means that a task is enabled as soon as its input data is available. All the tasks can run concurrently and those that have not the input data available will just block. (52, 66)

In figures 2.15 and 2.16 are represented the two workflow models. In the data driven model there are no control structures. The *XOR* join used at *task 4* in the control driven model can be emulated in the data driven model only by replicating *task 4*. However, a test and its contrary, concerning for example the value of a parameter, is performed at the level of tasks 2 and 3. The result of this evaluation conditions the execution of *task 4*. This way only one instance of *task 4* will be executed.

As in the case of the different levels of definition of a workflow application, this classification (control vs data driven) is rather theoretical. In practice no workflow

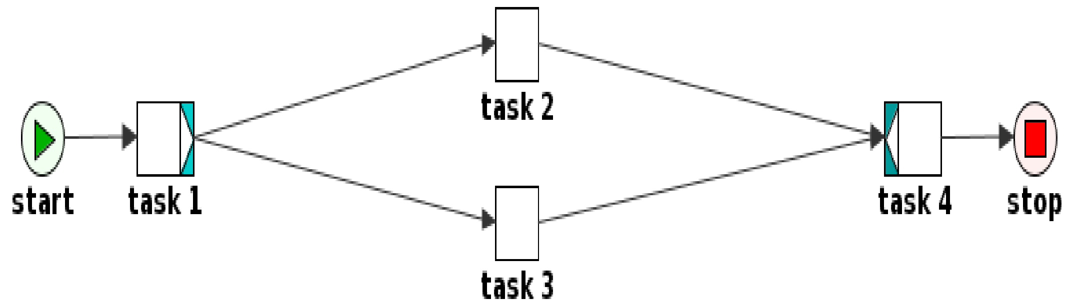


Figure 2.16: Control Driven Workflow

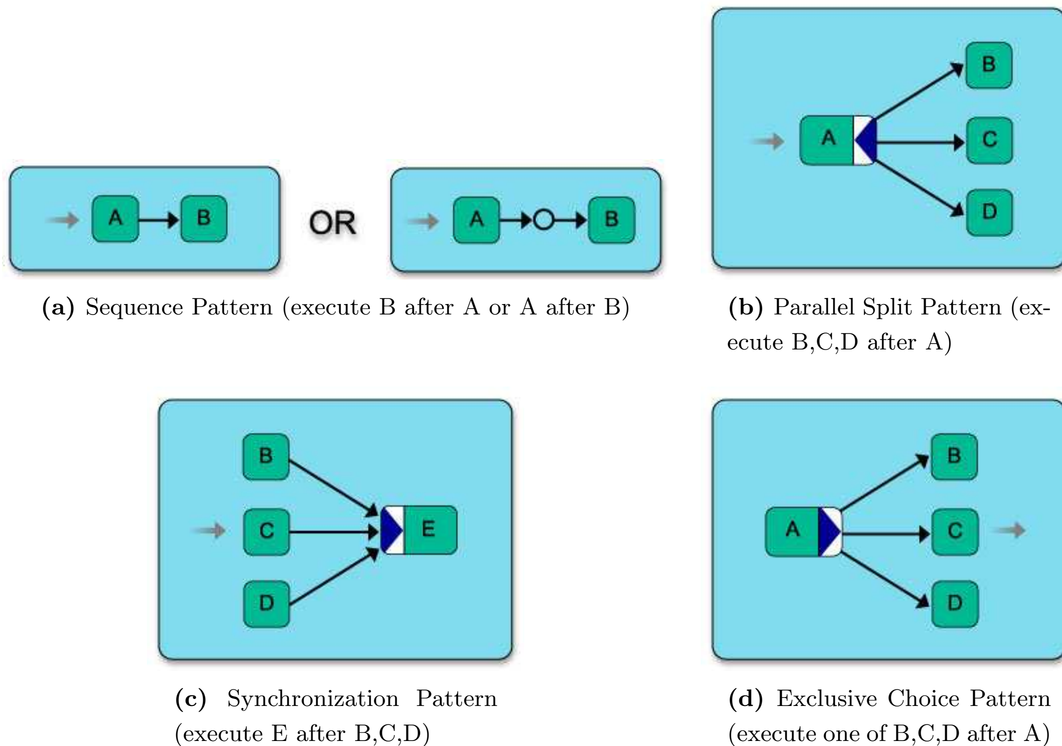
system is based entirely on one model or the other. Most of them adopt a hybrid approach between the two models in order to assure the requirements of the application domain they were designed for. For both models we can find arguments in favor or against. For example the data-driven approach ensures default parallel execution of independent tasks. Also it is better suited to model exception detection at parameter level since every task first has to verify the integrity of its input data before execution but also of the produced output data after execution. The downside of this model is the limited means of representation of more complex applications that we usually find in the scientific domain. This is why control-driven approach offers more control over the actual execution of the tasks.

The consequence of so many possibilities of designing a workflow systems is the absence of a common workflow language. This would allow scientists to execute a workflow application within a Grid environment independent of the tool that he used to create that workflow. The existence of so many workflow architectures makes it impossible to share workflows across working groups using different tools or execute on Grids where those tools are not installed (67).

2.4.1.3 Workflow Patterns in Control Driven Models

According to the official website (68) “the Workflow Patterns initiative” is a joint effort of Eindhoven University of Technology (led by Professor Wil van der Aalst) and

Queensland University of Technology (led by Professor Arthur ter Hofstede) which started in 1999. The aim of this initiative is to provide a conceptual basis for process technology. In particular, the research provides a thorough examination of the various perspectives (control flow, data, resource, and exception handling) that need to be supported by a workflow language or a business process modeling language. The results can be used for examining the suitability of a particular process language or workflow system for a particular project, assessing relative strengths and weaknesses of various approaches to process specification, implementing certain business requirements in a particular process-aware information system, and as a basis for language and tool development." Process modeling systems like Petri nets often lack support for more complex control structures (multiple instance, cancellation or the generalized OR-join). Workflow patterns represent a good tool to compensate this by extending these systems in order to deal with these control structures (6). We present in figure 2.17 some of the basic control flow patterns.



(a) Sequence Pattern (execute B after A or A after B)

(b) Parallel Split Pattern (execute B,C,D after A)

(c) Synchronization Pattern (execute E after B,C,D)

(d) Exclusive Choice Pattern (execute one of B,C,D after A)

Figure 2.17: Basic Workflow Patterns

2.4.1.4 Dynamicity in Workflow Systems

Very often a scientist can not predict from the design phase all the scenarios that his experiments will follow during execution. Instead when formulating experiments as scientific workflows, the scientist can design an initial workflow and subsequently test it with different combination of parameters and process adaptation until a suitable solution is found. But in a static implementation of the workflow system this would mean restarting the whole experiment from the beginning each time a design modification or a modified parameter value is tested (69). To avoid this overhead, the system must be flexible and accept change during run-time, thus continuing the initial thread of execution. Also in some cases, the tasks to be performed or parameters to be used at a certain point may be so dependent on the results provided by the previous tasks that it does not make any sense to try to predict them. The most desirable solution would be to enable the specification of the next task and their inter-dependencies as soon as the current task has finished. Nowadays, the application of computational technologies to new problems and the need to get more accurate outcomes demand the use of updated (and in some cases real-time) data inputs. The *Dynamic Data Driven Application Systems* (DDDAS) (52) concept entails capabilities where application simulation can dynamically accept and respond to field data and measurements. Nevertheless, final computations and data management are going to be performed on real resources. This can involve issues such as availability, capacity, performance limitations that could prevent the normal executions of the experiment. Having a system that can address dynamically all these faulty scenarios, significantly improves the performance. Also going further, dynamicity can be a great tool to address faulty design errors (by modifying the thread of execution at run-time, or changing parameters' values), programming errors, or other type of application errors, thus contributing to the resiliency of the system.

Since a workflow description spans over multiple levels (as described in 2.4.1.1), dynamicity requirements can be formulated almost at each of these levels (52):

- *Dynamicity at abstract level* - When an experiment is too big the scientist may have difficulties to design it completely. When this is the case, it would be helpful if the abstract representation of a workflow application can be modified during

execution according to the current context. This way the user can add new tasks, delete old tasks that are not needed anymore, or change the execution flow (70).

- *Dynamicity at concrete level* - Even if the abstract representation remains stable all along the execution, a user might need to adapt parameter values or even add new parameters if needed. This requirement can be a consequence of the fact that data is not always available at the beginning of the execution but obtained at run-time. Also the user can change values of parameters in order to improve the quality of the results and perform comparisons and other necessary tests. Certain values of parameters can cause erroneous behavior (exceptions) of the application and have to be changed in order to re-establish the normal path and save the work done until the exception detection.

- *Dynamicity at instance level* - Changing resources during execution is justified by several reasons. For example the user can decide between sparing resources of an infrastructure and choose smaller resources for non-critical tasks or on the contrary, allocate powerful resources for critical tasks. Another reason are the multiple ways in which a computing resource can fail causing the application to stop. Sometimes is useful to choose the computing resource of a task at run-time when all the technical requirements are defined (capacity, software tools, etc.). (66)

2.4.1.5 Exception Handling in Workflow Systems

When executing a scientific application on a Grid infrastructure through a workflow system, failures can occur for various reasons: hardware/system failures (network failures, resource non-availability, system out of memory) but also application failures(faulty algorithm, infinite loops, inadequate parameter values, stack overflow, run-time exception, programming bug). The first type of failures are mostly treated by the middleware layer residing between the workflow environment and the Grid infrastructure. Grid workflow systems should be able to identify and handle errors and support reliable execution no matter the type of failure (71). The different error handling procedures can be divided into two main categories: task-level and workflow-level techniques (72). Task-level techniques mask the effects of the execution failure of tasks in the workflow, while workflow-level techniques manipulate the workflow structure such as

execution flow, to deal with erroneous conditions. The main task-level techniques are the following:

- *Retry* - Is the simplest recovery technique, as it simply tries to execute the same task on the same resource after failure occurrence.
- *Migration (or alternate resource)* - Submits failed task to another resource (73).
- *Checkpoint/Restart* - An application/task is progressively restarted from the last good checkpoint, if available, on different resources in case of failures. The migration algorithm determines the best migration path (74, 75, 76).
- *Replication (or over-provisioning)* - Is a fault tolerance mechanism where multiple copies of an application (with the same input data set) are executed in parallel (73, 77).

The above mentioned fault tolerant techniques are mostly dedicated to hardware and system failures. They can also be used for application failures but their efficiency can prove to be very weak. Here is a set of reasons for which more advanced recovery techniques grouped in the concept of resiliency have to be developed that could address this type of failures (2):

1. If a task fails, not because of a resource failure, but because of a failure in the task itself (e.g. run-time exception or programming bug), using the retry or migration recovery techniques will only waste valuable resources without having a chance to successfully complete.
2. The amount of data needed to be check-pointed and the expected rate of faults for large systems are already exposing the limits of traditional checkpoint/restart techniques.
3. The most common parallel programming model, MPI, does not offer a paradigm for resilient programming. A failure of a single task often leads to the killing of the entire application. An exception is the MPI implementation for volatile resources (MPICH V) (45, 78).

4. Most applications (and system) software are not fault tolerant nor fault aware and are not designed to confine error/faults, to avoid or limit their propagation, and to recover from them when possible (except in limited cases (39, 78)).

5. There is little communication or coordination between the layers of the software stack in error/fault detection and management, or coordination for preventive or corrective actions. An example of such an infrastructure is presented in (1). However the number of such infrastructures is still low.

6. Errors, fault root causes, and propagation are not always well understood.

7. There are no standard metrics, no standardized experimental methodology, nor standard experimental environment to stress resilience and compare them fairly.

The workflow system should be able to monitor the application's execution and detect any software error, treat it (if possible) and continue the execution in a safe manner. Since this type of failures didn't benefit of wide studies and experiments like the hardware/system failures, the research agenda should follow mainly two important tracks as stated in (2):

- Extend the applicability of rollback toward more local recovery, reducing checkpoint size, error and fault confinement, dynamic error handling by applications.
- Fault avoidance and fault oblivious software to limit the recovery from rollback, situation awareness, system level fault prediction for time optimal check-pointing and migration.

Next we will present a set of representative workflow systems for scientific computation, insisting on YAWL system which is the one we have chosen to integrate in our computing platform.

2.4.2 Examples of Fault Tolerant Workflow Systems

The number of available grid workflow platforms is growing rapidly every year. Making an exhaustive survey is almost impossible and out of our scope. We will only present some systems that we consider relevant for the scientific field and that promote similar features with the one that we want to integrate in our platform.

2.4.2.1 Askalon

Askalon (79) is one of the most complete workflow systems that we found in the literature and specially designed to simplify the design and execution of scientific workflow applications on the Grid. It contains all the necessary floors for such a goal, as depicted in figure 2.18. Thus Askalon can build an abstract workflow model using the AGWL XML based language that shields the application developer from the grid. The resulting XML file is then transferred to the set of middleware services that support the execution on the Grid. The *Resource Manager* service is responsible for allocation of resources and deployment of required services for the execution. The *Enactment Engine* service is in charge of a reliable and fault tolerant execution of workflows (80). The paper proposes a master-slave distributed architecture for the execution engine. The master part parses the workflow representation and after some format translations it is sent to the workflow scheduler. After the workflow is mapped onto a grid infrastructure, the master engine partitions it and allocate the resulting partitions to the slave engines. An eventual unrecoverable crash of a slave engine is monitored by the master engine that will mark the entire subworkflow associated to the slave as failed. It will then ask for a rescheduling procedure, will re-partition the workflow and migrate the execution to another site. Also the master engine chooses a slave engine as a back-up before initiating the workflow execution. This way if the master engine crashes too there will always be a replacement available.

The DEE engine of Askalon can perform also application-level checkpoints for a fast execution restore in case of failures. Such a checkpoint will contain the state of the workflow activities and the state of the data dependencies. The different checkpoints performed by DEE are classified according to their level in the application: *activity-level, light-weight workflow* and *workflow-level* (see figure 2.19). More details can be found in the above cited paper.

The *Performance Analysis* performs automatic instrumentation and bottleneck detection and feeds the *Performance Prediction* service with statistics that help to estimate execution time of activities through training phase. The mapping of activities onto Grid resources is achieved by the *Scheduler* service using graph-based heuristics and optimization algorithms.

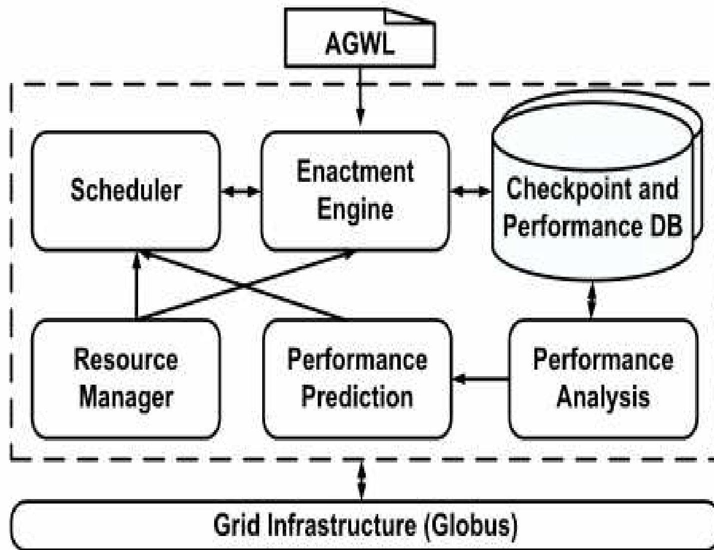


Figure 2.18: Askalon Architecture (taken from (79))

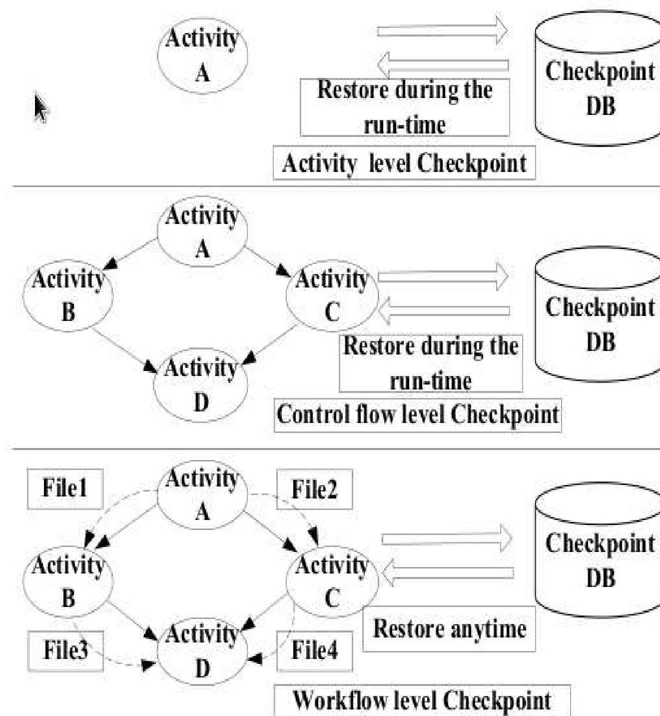


Figure 2.19: Askalon Checkpoint Classification (taken from (79))

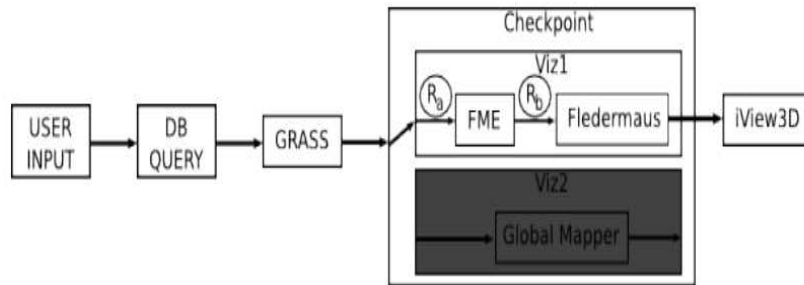


Figure 2.20: Checkpoint Model in Kepler System (taken from (83))

2.4.2.2 Kepler

Kepler(81) is based on the collaboration of several large scale projects with the aim of developing an open source scientific workflow system. This system allows scientists to combine data integration with analysis or visualization steps. Kepler is based on Ptolemy II system (82) developed at UC Berkley, a system for heterogeneous hierarchical modeling. Kepler can model a workflow system as a composition of independent components (actors) that communicate through well-defined interfaces. An actor represents an operation with parameters that operates on input data to produce some output data. Also, according to (72) Kepler has been extended to support seamless access to remote resources and services. Finally in (83) it is presented a method for capturing data values and control dependencies for provenance information in the Kepler system. It also describes how Kepler is using a *Checkpoint* composite actor to provide fault tolerance. The error detection is realized through *port conditions* that evaluate the input or output data. An error is signaled if the evaluation results to false. The *Checkpoint* composite actor contains a primary subworkflow and optionally several alternate subworkflows. When an error occurs in the primary subworkflow the *Checkpoint* actor can choose either to re-execute it or to call the execution of an alternate subworkflow. The number of times such a procedure can be applied is configurable in the design phase. If the upper limit of re-executions is exceeded the error is sent higher in the workflow hierarchy. In figure 2.20 is presented an example of a workflow *Checkpoint* model in the Kepler system.

2.4.3 YAWL - Yet Another Workflow Language

YAWL is a workflow language designed and developed by Wil van der Aalst (Eindhoven University of Technology, the Netherlands) and Arthur ter Hofstede (Queensland University of Technology, Australia) in 2002. After analyzing a wide variety of workflow management systems and languages they decided to base their language on the Petri nets. The reason for choosing this mathematical model is its enhanced expressiveness. It offers a good formal semantics despite the graphical nature, behaves much better when dealing with state based workflow applications and offers a multitude of analysis techniques (84). However it proved out that using only Petri nets was not enough to model more complex control structures. A good example are the advanced synchronization patterns like *AND* join and *XOR* join (65) or the multiple instance patterns. In consequence, they extended the initial language with additional features to accommodate these control structures (84). These features are all grouped under the concept of workflow patterns (68).

Beside the workflow language concept, YAWL extends to a wider concept representing an entire workflow system. All the components described in section 2.4.1.1 are present in YAWL. Their detailed description is given in (6), we will only enumerate the basic ones present in any workflow system:

- *Workflow Process* - A set of interdependent activities that need to be performed.
- *Workflow Specification* - Detailed description of a process ready to be deployed in a workflow engines.
- *Case* - A specific instantiation of a workflow model.
- *Task* - A description of a unit of work, part of a workflow application.

2.4.3.1 Architecture

Having a workflow language formally based on Petri nets and implementing the well-known workflow patterns (7), YAWL extends these concepts with dedicated constructs to deal with patterns like cancellation, synchronization of active branches only and multiple concurrently executing instances of the same task. Moreover, YAWL is based on a service-oriented architecture (see Figure **??**) that greatly contributes to its extensibility.

From a software engineering point of view YAWL has three main components: YAWL Editor, YAWL Engine and YAWL Services. YAWL is conceived on a server-client model and consists mainly of a set of servlets that act as server or client classes for the different services implemented. As a consequence these services need to be hosted by a servlet container, the most popular being Apache Tomcat (6). The YAWL Engine is responsible for the scheduling of task's execution and also for managing the data flow inside a workflow application. This coordination is accomplished using the so-called YAWL Custom Services (85). A Custom Service is usually a web-based service that is able to communicate with the YAWL Engine facilitating its interaction with external software in charge of executing tasks. This communication is done through special endpoints called interfaces (see Figure **??**). Every interface has a different role: loading and unloading workflow cases in the engine, exception handling, logging or interaction with a Custom Service.

2.4.3.2 YAWL Custom Service

Normally a Custom Service can be developed in any programming language and can be deployed (locally or remotely) on any kind of platform. The only condition is to be able to send and receive HTTP messages. The communication between the YAWL Engine and a Custom Service is done using a specific procedure depicted in figure 2.21. Being in charge of tasks' execution, a Custom Service is able to receive notifications from the engine when new tasks are ready to be executed, inform the engine that it has accepted a task, execute the activities contained in a task and inform back the engine that a task execution has finished, allowing the engine to continue the rest of the workflow's execution.

Custom Services are registered to the Engine by specifying basic authentication credentials, and a location in the form of a *base URL*. Once registered, a Custom Service may receive HTTP messages from the Engine at endpoints identified by URLs derived from the base URL provided at registration. On the other hand, the Custom Service can send HTTP messages to the Engine at endpoints identified by URLs that the Custom Service is assumed to know. A collection of Java classes included in the YAWL distribution provide several APIs that can be used to implement the required endpoints on top of the HTTP protocol (85).

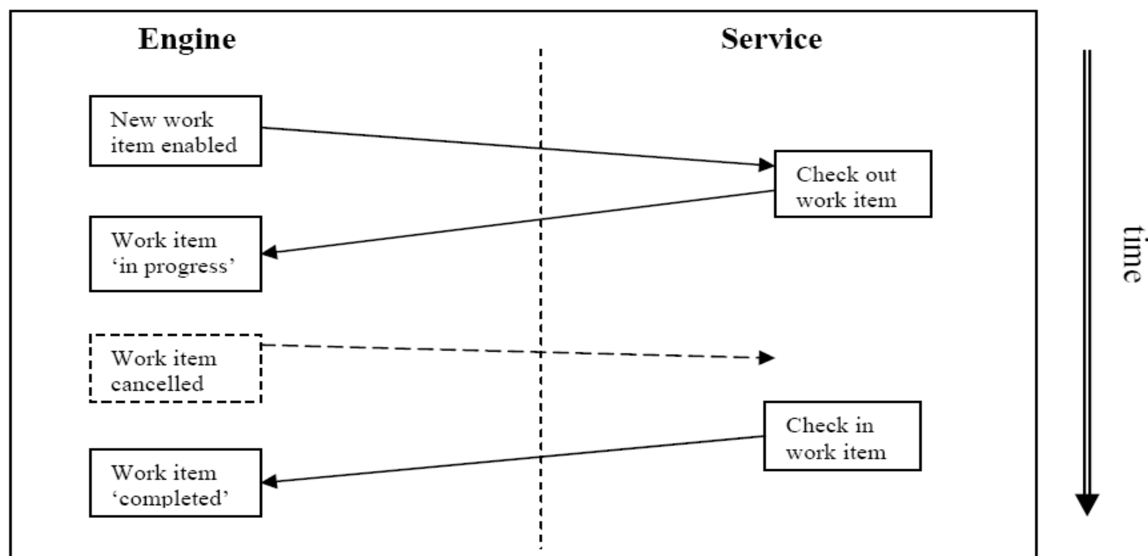


Figure 2.21: YAWL Custom Service Protocol (taken from (85))

2.4.3.3 Data in YAWL

The data transfer between tasks in a workflow application or between the engine and the external environment is done through XML documents. There are two levels available for data definition: net and task level. The net data is defined as global data that every task can access during execution. The task data is accessed and modified only within an individual instance of a task. Data types are defined using XML Schema and apart from the default set existing in the YAWL distribution a users can define their own custom data types. Concerning data usage, there are input and output variables, input/output or local variables. The general rule is that data is written to input variables and read from output variables. Local variables are only defined at net level and are used to pass initial data to the application.

Data transfer is possible only between net variables to task variables. No direct transfer is allowed between variables of distinct tasks. It is considered that task variables are local to the tasks they belong to and no direct access should be given to outside world. The definition of data transfer is done using *parameters*. They describe how data should be extracted from variables of a specific level and handled to variables of a different level (net to task $\rightarrow$ input parameter or task to net $\rightarrow$ output parameter).

When transferring data, some basic rules have to be respected. One of them is that all input variables, except those associated with the top-level net, must have data provided to them from the corresponding net variables via an input parameter definition. An input variable of the net level has data supplied from the external environment (e.g. user input) once the execution of a net specification has started. Data can also be assigned at design time, but only to local net variables. Also each output variable requests data from the environment once the corresponding net or task is executed (6).

2.4.3.4 Dynamicity

YAWL language supports flexibility through a number of constructs at design time. Like many other languages, YAWL supports parallel branching, choice, and iteration natively, which allow for certain paths to be chosen, executed, and repeated based on conditions and data values of the executing instance. In addition (and unlike most other languages), YAWL also supports advanced constructs such as multiple atomic and multiple composite tasks, where several instances of a task or sub-net can be executed concurrently and dynamically created. Another interesting feature are the cancellation sets, which allow for arbitrary tasks (or set of tasks) to be canceled or removed from a process instance. YAWL also supports flexibility through its service oriented architecture that we already described in 2.4.3.2. In the YAWL distribution there are already a set of built-in services designed to serve standard functions needed in a process execution (YAWL Resource Service, YAWL Worklet Selection Exception Service, etc.). One of the most important built-in service, providing dynamic flexibility support for YAWL processes is the *Worklet Service* (69, 86).

YAWL provides each task of a process instance with the ability to be linked to a dynamically extensible repertoire of actions. In this way the right action is chosen contextually and dynamically from this repertoire to carry out the task. In YAWL such an action is called a worklet, which is a small, self-contained, complete workflow process. The global description of the process is provided at design time. At run-time, when a specific task gets enabled by the engine, the appropriate worklet is contextually selected, using an associated set of rules. The context of a process is defined by the contextual data that can be categorized as follows:

- *Generic data* : Data that are considered likely to occur within any process. For instance, in a numerical optimization simulation, the input geometry data is considered as generic.
- *Case dependent with a priori knowledge* : Data that are known to be pertinent to a particular case when the workflow is instantiated. For instance, some process specific parameters that are used only under certain circumstances.
- *Case dependent with no a priori knowledge* : Data that only becomes known when the case is active and deviations from the known process occur. A typical example is an error code that will change the execution flow.

The YAWL approach to capture contextual data are Ripple Down Rules (RDR) (86, ch.4), which comprise a hierarchical set of rules with associated actions. A RDR knowledge base is a collection of simple rules of the form *if condition then conclusion*, conceptually arranged in a binary tree structure (see Figure 2.22). Such a *decision tree* like structure allows to define the most specific case fitting with given contextual data and therefore to decide the most appropriate worklet to handle that data. A typical example is the case where several methods are available to solve a linear system according to the kind of input data (for instance if the input matrix is diagonal or sparse)

2.4.3.5 Exception Handling

The *Worklet Exception Service* extends the capabilities of the Worklet Service to provide dynamic exception handling with corrective and compensatory actions (86, ch. 5). The Exception Service uses the same repertoire and Ripple-Down-Rules as the Worklet Selection Service. For every unanticipated exception (an event not expected to occur in most instances, so excluded from the main logic) a set of exception handling processes are defined, known as exlets, which will be dynamically incorporated in the running process. An exlet can also contain a compensatory action in the form of a worklet, defined in the same manner as for the Selection Service. Each exception has also a set of rules attached that will help choosing the right exlet at run-time, according to the predicate evaluated to true. If an unanticipated exception occurs (an event for which a handling exlet has not been defined), either an existing exlet can be manually

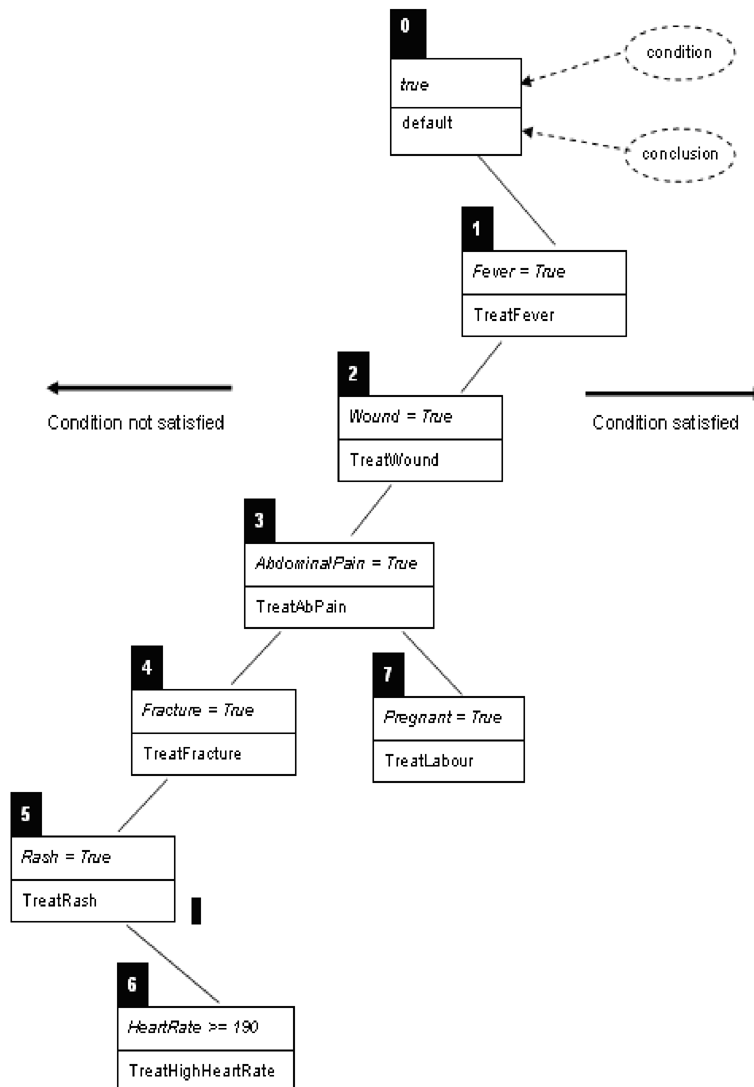


Figure 2.22: Conceptual Structure of a Ripple-Down-Rule Tree (taken from (6))

selected from the repertoire, or one can be adapted on the fly, or a new exlet can be defined and deployed while the parent workflow instance is still active. The method used to handle the exception and the context in which it has occurred are captured by the system and immediately become an implicit part of the parent process model. This ensures the continuous evolution of the process while avoiding the need to modify the original definition. There are three types of exceptions that are defined by the service for handling, as detailed below:

- *Pre/Post Constraints* exceptions - Rules applied to a work item or case immediately before and its after execution. An exception is raised whenever input or output data do not meet the criteria.
- *Time Out* - Occurs when a work item has an enabled timer and the deadline for that timer is reached.
- *Resource Unavailable* - Triggered by the Resource Service when an attempt has been made to allocate a work item to a resource but that allocation is not possible for various reasons.

When one of the above mentioned exceptions occurs, an appropriate exlet, if defined, will be invoked. Each exlet may contain any number of steps, or primitives. The available pre-defined primitives are the following: *Remove Work Item, Remove Case, Remove All Cases, Suspend Work Item, Suspend Case, Suspend All Cases, Continue Work Item, Continue Case, Continue All Cases, Restart Work Item, Force Complete Work Item, Force Fail Work Item, Compensate.* A number of compensatory worklets may be executed consecutively by adding a sequence of compensation primitives to an exlet.

2.5 Resilience for Long-running Simulation and Optimization Applications

Numerical simulation plays an important role in most scientific research fields and usually give rise to very large scale experiments. This reflects into significant volumes of data to be transferred and a substantial demand for computing resources.

The application areas that our colleagues from Sophia Antipolis focus on concern optimization of complex systems arising from physics or engineering. From a *physical* point of view, they study Fluid and Structural Mechanics and Electromagnetics. Major applications include multidisciplinary optimization of aerodynamic configurations or geometrical optimization.

The multidisciplinary aspect of applications requires a certain heterogeneity both at software level and computing resources level. The integration of such various disciplines involves powerful computing infrastructures and particular software coupling techniques. Simultaneously, advances in computer technology militate in favor of the use of massively parallel PC-clusters including thousands of processors connected by high speed gigabits/sec wide area networks. The main difficulty still remains however in the deployment and control of complex distributed applications on grids by the end-users. Indeed, the deployment of the computing grid infrastructures and of the applications in such environments still requires specific expertise by computer science specialists (87).

From the above introduction we can observe that long-running simulation and optimization applications are affected by exceptions at multiple levels. In Figure 2.23 are presented the main stages of a typical numerical optimization application of the kind that we address in this work. The tasks in the computing chain represent int this order the optimization part, the meshing of the geometry model, the partitioning of the mesh model, the solver of the optimization ecuations and finally the performance analyser. They represent at the same time the logical pieces of the application but also the blocks where some exceptions can appear. This can happen either before the execution of a block, during the execution or at the exit when the results are produced. Since the hardware errors are out of our scope, we will describe only exceptions concerning resource limitations and application level exceptions.

2.5.1 Exception Types

2.5.1.1 Resource Limitation Exceptions

Even though grid middleware systems are more and more performant and they should cope with most of applications' requirements, the heterogeneity of simulation and optimization applications can determine unexpected errors regarding resource performance.

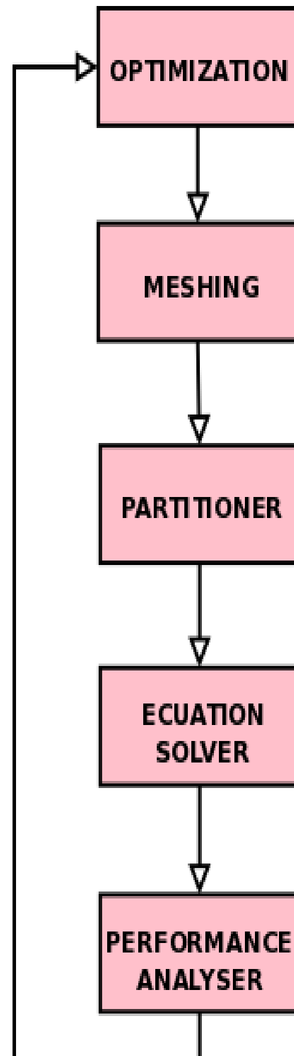


Figure 2.23: Standard Optimization Application

Thus, the most common exceptions that we encounter are:

- Out of CPU time - This can happen if the CPU time is limited on the computing platform used (e.g. grid or cloud computing).
- Out of memory or disk storage - If a job launches too many processes on a processor the memory or disk space available for a job is exceeded. In our experiment the meshing task (figure 2.23) turns out to be likely to this kind of error.
- Wrong OS - A numerical optimization application is multidisciplinary by nature so every logical task can have different requirements in terms of software to use, operating systems type or version and may sometimes run on different OS or architecture. If the job/jobs in charge of executing a task are allocated resources with the wrong requirements, it can result in a failure, blocking the execution of the rest of the application.
- Non availability of web-services - If a web-service in charge of execution of an application's task becomes unavailable (e.g. if network is down), the whole application's execution suffers.

2.5.1.2 Application Exceptions

The application exceptions represent the focus of this thesis and most of the research efforts were made to develop mechanisms to detect and treat these exceptions. Based on Figure 2.23 we can identify the nature of these exceptions:

- Input/Output parameters - Every task receives a set of input parameters and produces a set of output parameters to other tasks. A wrong value of those parameters may produce wrong final results or even crash some task.
- Meshing errors - The software in charge of this task can generate a wrong mesh or the output file can get broken which will affect the dependent tasks.
- Convergence problems - The solver task can end up in an infinite loop, overloading the computing resources and blocking the application.
- Algorithm design - If the end-user inserted design errors in his algorithm, these can whether go through execution silently but affecting the final results, or stop the application in the middle of the execution.

2.5.2 Exception Detection

Detecting application and resource limitation exceptions is a more abstract task than the general detection used for errors in distributed systems. Most of these exceptions can go unidentified, making visible only the effects produced in the system. The different exception types presented above can be detected at different levels of the execution:

- Operating system level - At this level we can detect errors like CPU time or memory shortage. The method to detect such exceptions is to invoke special OS commands that shows the level of resource utilization by a process or set of processes (e.g. *free*, *vmstat*, *top*). Also at this level we can detect if the requirements of the job match with the configuration of the operating system, triggering an exception if this is not the case.

- Service level - This is the level where we detect if web- services in charge with execution of tasks from applications are functioning correctly. The easiest way to test if a service is responsive is to put a timeout on the execution of that service for a specific task and trigger an exception if the timeout value is exceeded by the execution time.

- Application level - The rest of the exceptions are treated at the application level. When dealing with values for input/output parameters, the most common way is to place value constraints before and after the execution of specific tasks. If the values of the parameters violate those constraints, an exception is triggered. The convergency problems are usually detected by placing timeout constraints and trigger exceptions if the solver task that should converge exceeds the timeout with its execution time.

2.5.3 Exception Treatment and Recovery

For this phase the most important aspect is to determine the origin of the exception. Otherwise there is a risk to repeat the occurrence of the same exception. When we know the origin of an exception we can determine easier the recovery point for an application. When working with simulation and optimization applications the two main options are whether to put exception detectors all over in the application space thus assuring a very fine detection of the origin of the error, or to adopt a user-defined strategy in which

the user places detectors only in critical points of the application based on his a priori knowledge.

The usual approach to facilitate recovery for application exceptions is similar to the one presented in 2.3.2.2 for general exception handling in distributed systems, meaning saving the state of the application represented by all the available information about the processes and data (88). By doing this we have sufficient data to restore an application state after a recovery procedure but also we can modify specific parameter values as a fault treatment procedure in order to avoid future occurrences. If the application context permits, we can isolate critical tasks determined by the user and save only their application context, thus performing a local exception treatment and recovery that increases the recovery speed which is essential in long-running simulation and optimization applications. When the exception is related to resource limitation, the recovery procedure consists in transferring a task's state on a different computing resource that meets the requirements of that task and re-execute it.

Considering the similarities between workflow systems for numerical applications and those for business processing we mention also an original recovery solution presented in (89). The authors proposed a practical solution for on-line attack recovery of workflows. The recovery system discovers all damages caused by the malicious tasks and automatically repairs the damages based on data and control dependencies between workflow tasks.

3. Platform Design

An important challenge in computer science nowadays lies in the integration of various expertise in complex application areas such as simulation and optimization in aeronautics, automotive and nuclear simulation. For example, the design of a space shuttle calls for aero-thermal, aero-structure and aerodynamics disciplines which all interact in hypersonic regime, together with electro-magnetics.

The integration of such various disciplines requires powerful computing infrastructures and particular software coupling techniques. Simultaneously, advances in computer technology encourages the use of massively parallel PC-clusters including thousands of processors connected by high-speed networks. This conjunction makes it possible to combine computational methods and computer science for better performance. New approaches including evolutionary algorithms, parametrization, multi-hierarchical decomposition lend themselves seamlessly to parallel implementations in such computing infrastructures.

However, even if today there would be available petaflop computers, numerical simulation teams are not fully ready to use them. Despite the fact that each discipline has made significant progress to develop tools that cope with the computational power, coupling them for designing large multidisciplinary systems is still in an incipient state. The main reasons for this situation are:

- Simulation and optimization algorithms that scale well with large computing infrastructures are quite seldom.
- In a multidisciplinary context, coupling disciplinary analysis with optimization while benefiting from several parallelization levels remains a technical and scientific issue. The amount of exchanged data needs to be reduced to achieve a speed-up. Parameter definitions may be widely different between disciplines, therefore creating incompatible interfaces.
- Interfaces are long and difficult to code, accessibility in different operating systems, conflicting requirements of the simulators, software licenses attached to particular nodes, heterogeneous computing nodes and connecting network.

3.1 Numerical Optimization Applications

Optimization problems involving systems governed by Partial Differential Equations (PDEs), such as optimum shape design in aerodynamics or electromagnetism, are more and more complex. In certain situations, the major difficulty resides in the costly evaluation of a function by means of a simulation, and the numerical method to be used must exploit at best the problem characteristics (regularity or smoothness, local convexity). In many other cases, several criteria are to be optimized and some are non differentiable and/or non convex. A large set of parameters, sometimes of different types (boolean, integer, real or functional), are to be taken into account, as well as constraints of various types (physical and geometrical, in particular). Additionally, todays most interesting optimization pre-industrial projects are multidisciplinary, and this complicates the mathematical, physical and numerical settings. Developing robust optimizers is therefore an essential objective to make progress in this area of scientific computing.

In the area of numerical optimization algorithms, our team aims at adapting classical optimization methods (simplex, gradient, quasi-Newton) when applicable to relevant engineering applications, as well as developing and testing less conventional approaches such as Evolutionary Strategies (ES), including Genetic or Particle-Swarm Algorithms, or hybrid schemes, in contexts where robustness is a very severe constraint.

The application domains of the optimization methods mentioned above cover a large spectrum. The most important for the team is the Aeronautics and Space. The demand of the aeronautical industry remains very strong in aerodynamics, as much for conventional aircraft, whose performance must be enhanced to meet new societal requirements in terms of economy, noise, vortex production near runways, etc. Our implication concerns shape optimization of wings or simplified configurations. Our current involvement with Space applications relates to software platforms for code coupling. Also the team's expertise in theoretical and numerical modeling, in particular in relation to approximation schemes, and multilevel, multi-scale computational algorithms, allows us to envisage to contribute to integrated projects focused on disciplines other than, or coupled with fluid dynamics, such as structural mechanics, electromagnetism, biology and virtual reality, image processing, etc in collaboration with specialists of these fields. The main objectives of these applications is to reduce the mock-up process and improve

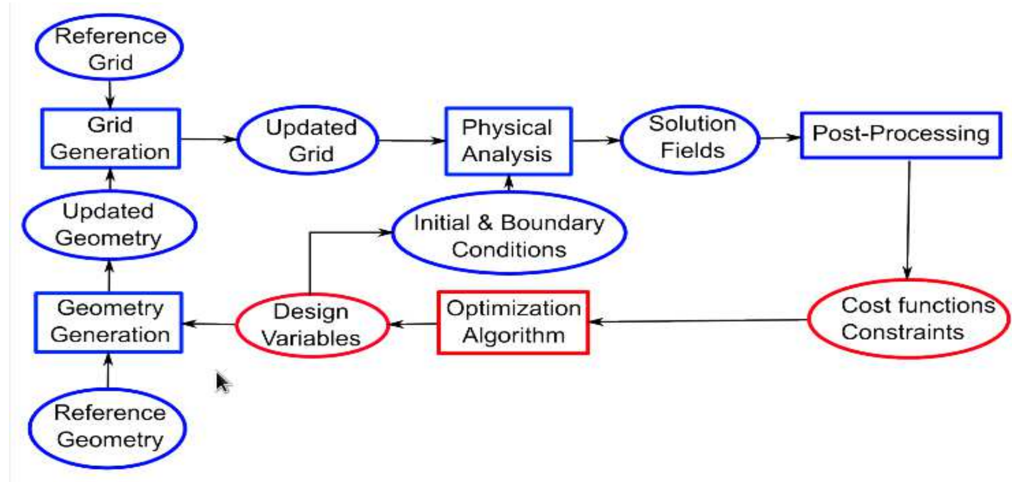


Figure 3.1: Optimization Loop

size and precision of modeling, innovate and improve customer requirements or reduce simulation time.

This explains the interest in integrating various expertise in complex application areas to design and develop high-performance distributed scientific workflows for multidisciplinary optimization applications combining powerful computing infrastructures with specific software coupling techniques.

In an optimization process there are multiple tools involved organized in a well defined structure in order to obtain the right outcome (figure 3.1). The design loop starts with a set of *Design Variables*. These design variables are used for *Geometry Generation* starting from a *Reference Geometry* and resulting an *Updated Geometry*. This is the base for the *Grid Generation* phase that like the geometry starts from a *Reference Grid*. It follows the *Physical Analysis* stage where all the *Initial Boundary and Conditions* are specified. A set of *Solution Fields* are generated followed by a *Post-Processing* phase where all the *Cost Function Constraints* are verified. If these constraints are respected it means that the required level of optimization has been reached so the application stops. Otherwise an *Optimization Algorithm* is executed to modify the design variables in a proper way and the whole process starts again until the constraints are met.

All the application stages identified above are prone to errors. We will see in the next sections what type of errors can occur in such an optimization chain, where can they occur and what fault tolerance solutions can be applied to ensure a safe execution.

3.2 OMD2 Project

The OMD2 project (90) is an industry research project gathering small and medium enterprises (SMEs) like *CD-adapco, SIREHNA, ACTIVEEON*, university research institutions like *INRIA, ENSM-SE, UTC, ECP, IRCCyN, ENS CACHAN*, *DIGITEO consortium* and *RENAULT* car manufacturer as the coordinator. The project started on the 2nd of July 2009 for a duration of 3 years. It benefited of a financial support of 2.8M€ from the *National Research Agency* (according to program *Conception et Simulation 2008*) and had a total budget of 7.3M€. It aimed to connect multidisciplinary teams (fluids and solid mechanics, applied mathematicians and computer scientists) for solving difficult industrial problems. The strategy evolved around three directions:

- the up-scaling of existing design algorithms for (task and data) distributed computing
- their integration in a distributed, collaborative, open software platform
- their application to real automotive design problems with environmental objectives

Beside linking existing software together, OMD2 provided new control algorithms for multidisciplinary simulations, uncertainty propagation and optimization that work in a HPC environment. The final objective was to create a collaborative design platform that worked in general HPC distributed environments. Users were supposed to interact with the platform in the SCILAB environment. The developments were validated on important car industry test cases related to car environmental impacts. The project split in several work packages and sub-packages. Our team, had a more significant contribution in packages related to functional specification of the platform, its conception and development, culminating with the development of interfaces between the computing platform and the optimization algorithms provided by our partner teams.

As a final objective, OMD2 project was destined to prepare the French design community to the coming of HPC age by simulating, testing and optimizing on large parallel computer infrastructures. Eventually it created links between the engineering design community and the more advanced HPC communities like bio-informatics and climatology by sharing middleware and computing environments (91).

3.3 Numerical Application Test-Cases

OPALE team obtained a set of test-case applications as a partner in the OMD2 project. I will insist only on those that we actually used for our tests and just briefly present the others to help the reader better understand the type of applications we are working with.

The first test-case represents a 2D air-conditioning pipe for a car (figure 3.2). The objective of the test-case was to do some preliminary tests of methods and algorithms proposed by different partners and also to prepare procedures for the more complex 3D cases. As for the objective of the problem itself, it resumed to finding the optimal geometry to:

- minimize the pressure loss between inlet and outlet
- minimize the standard deviation of the output speed profile

The geometry of the test-case is described by 13 parameters marked in figure 3.2 (left) and its computation time is estimated at around 3 minutes on a desktop computer.

The second test case (92) (figure 3.3), which will also have its Famosa architecture described, treats the same problem as the previous one but in 3D. Regarding the objectives of the test-case, beside testing algorithms of optimization there is also the test of distant computation. The reason for this extra objective is the large execution time that requires using high performance computing infrastructure. The problem's objective is also to find an optimal geometry modeled this time by 8 parameters. One solution will be composed of 300000 cells for the mesh design and would take about 25 minutes of computation time which impose the use of HPC for optimization.

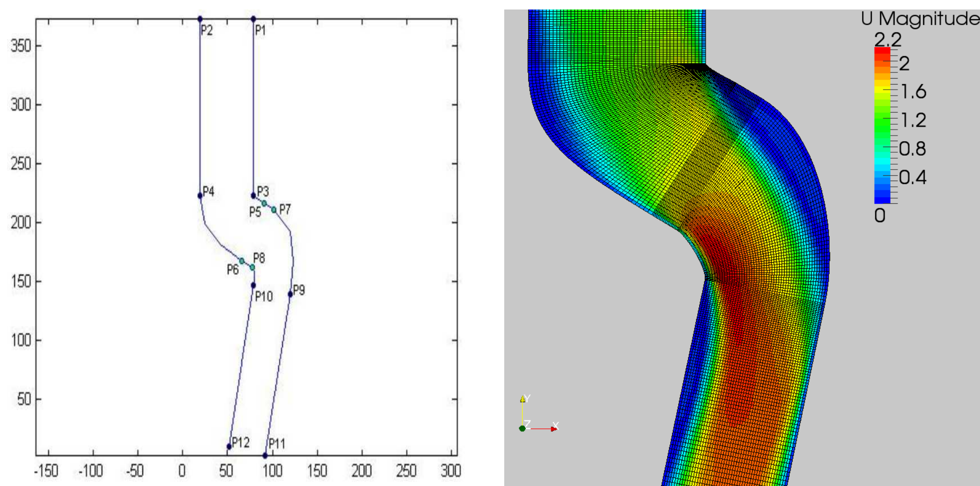


Figure 3.2: Test Case 1 - 2D Air Conditioning Pipe

3.4 Famosa Execution Chain

We first ran our experiments on the Famosa execution platform developed by our colleagues from Sophia Antipolis. Famosa has been developed in C++ and devoted to multidisciplinary design optimization in engineering. It is composed actually of several libraries with different functionality. For example we can find inside the platform a library that implements various optimization algorithms like steepest descent, multi-directional search algorithm or the efficient global optimization method. It contains also an evaluation library managing the performance estimation process (communication with external simulation tools). Other libraries are related to database creation, meta-model creation, etc.

The OPALE team uses this platform to test its methodological developments in multidisciplinary design optimization (see figure 3.4). From a software architecture point of view it is composed of two main stages: the optimization stage and the evaluation stage. The optimization stage takes as an input a function to optimize and an initial geometry. After each step it will produce a design variable vector that will represent the input for the evaluation action. With this design vector the platform will execute first the meshing procedure. The meshing can be partitioned so that parallelism can

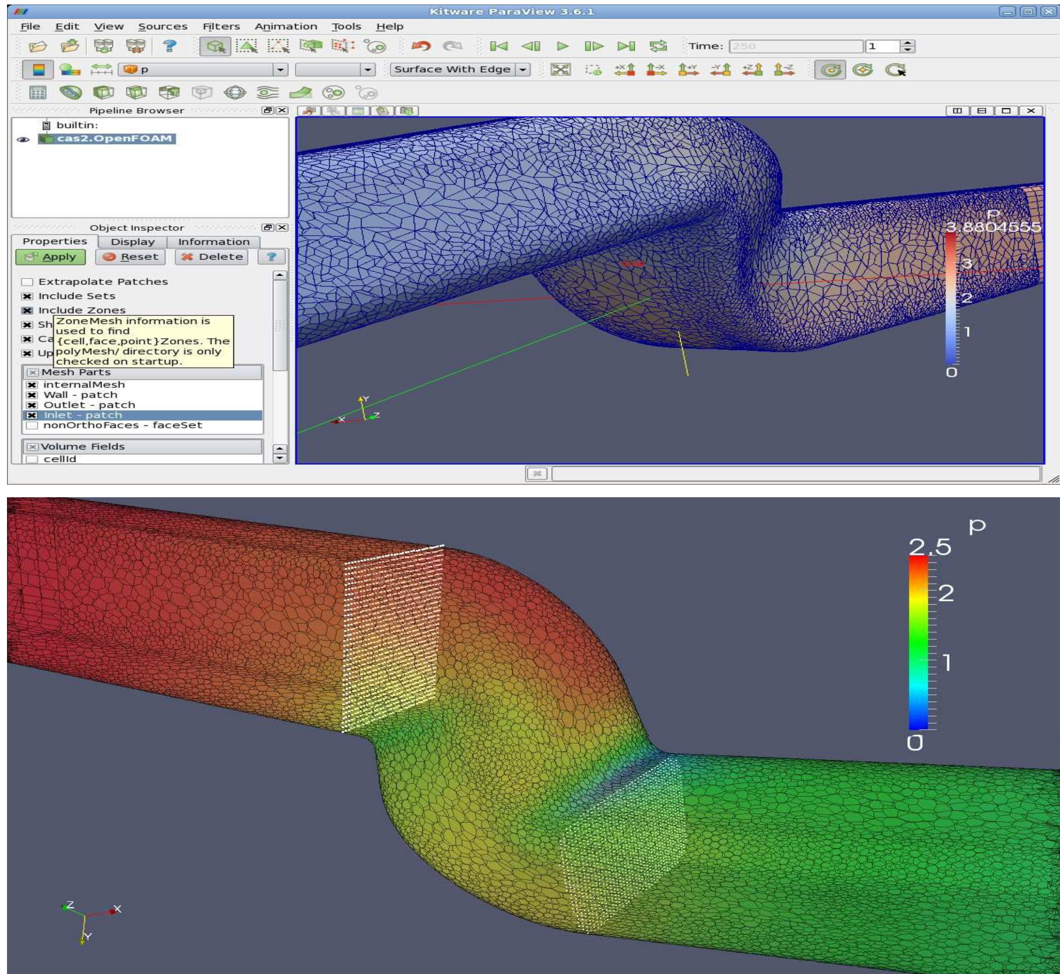


Figure 3.3: Test Case 1 - 3D Air Conditioning Pipe

be used through MPI. Based on the meshing results, the solver will compute the function to optimize. If the values of specific parameters respect certain constraints the procedure is stopped. Otherwise a new iteration is started from the optimization stage.

Comparing figure 3.1 with figure 3.4 we notice that the application stages represented by the geometry generation, mesh generation and solver are all included in the *runsolver.sh*. This script will execute on a single machine excluding the possibility of using a distributed computing platform for more demanding applications. Inside this script there is a pre-treatment phase in which are created all the symbolic links

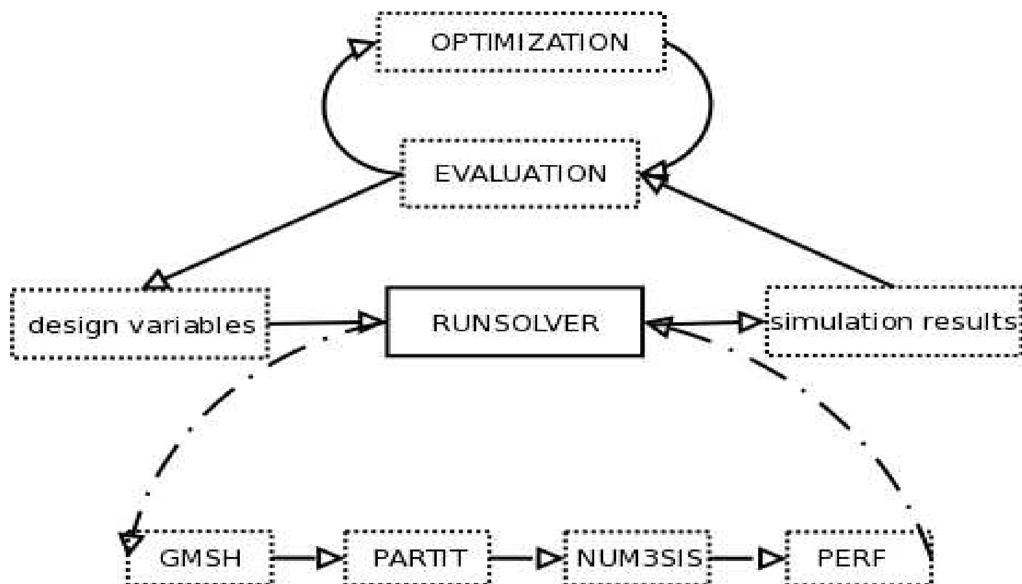


Figure 3.4: Famosa Computing Chain

to the executable software for geometry generation, meshing, solver, etc. Then these executables are called sequentially according to the application structure using a set of configuration files already existing or created during execution from one stage to another. The call to the *runsolver.sh* script is done inside *Famosa* tool.

The main advantage of this procedure is the fast implementation and execution launching. However, any error that occurs at any level will affect the entire script execution and a complete restart is needed. The monolithic structure doesn't allow any intermediary configuration, nor intermediary fault tolerance procedures implemented to prevent complete re-execution. The solution to this problem was to adapt the execution structure from figure 3.4 to the general optimization loop described in figure 3.1. By identifying in the Famosa execution chain every optimization phase described in 3.1 we were able to restructure the monolithic script into several smaller execution units, each associated with one optimization phase (see Figure 3.5).

This new structure allows a better control of the execution, facilitating an exception handling procedure that targets exceptions specific to each execution stage. In the following sections we will present what type of exceptions can affect an optimization process, which one we are interested in treating and how the new structure helped us developing a good algorithm for exception treatment.

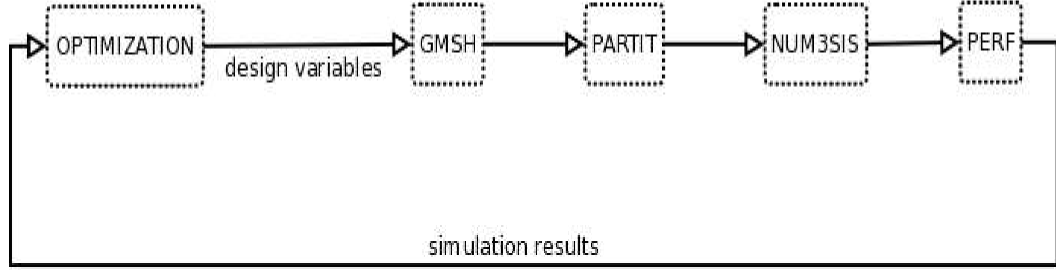


Figure 3.5: Standard Optimization Application

3.5 Large Scale Test-Case

The last test-case takes the optimization process to a larger scale to emphasize the distributed characteristics of our platform. It is considered as a realistic design of experiments exercise aimed at exploring a parameter space. The idea is to have an optimizer code at the beginning of the execution that will generate N different files containing geometrical parameters that have to be simulated. These files are being distributed on M ¡= N different nodes possibly running on different clusters of a distributed execution platform like Grid5000. For each file one must deploy on each node all the simulation tools already used in previous test-cases and defined in the Famosa computing chain (Figure 3.5): mesher, partitioner, parallel simulator, etc., and execute each of the individual workflow on K different cores, depending on each cluster. Each simulator has to return a locally produced result file that will provide the input data for the final code which produces a *response surface.* In figure 3.6 you can see an example of such a parameter surface. The meaning of the horizontal axes (geometrical parameters *X1* and *X2*) is explicated in figure 3.7 while the vertical ax measures the speed variation at the pipe's exit. Based on this, the experiment designer can identify the parameter areas that have a special meaning for his research. In our tests we can also introduce application faults by modifying some parameters and use the exception treatments that we will present in the following sections to ensure a safe execution.

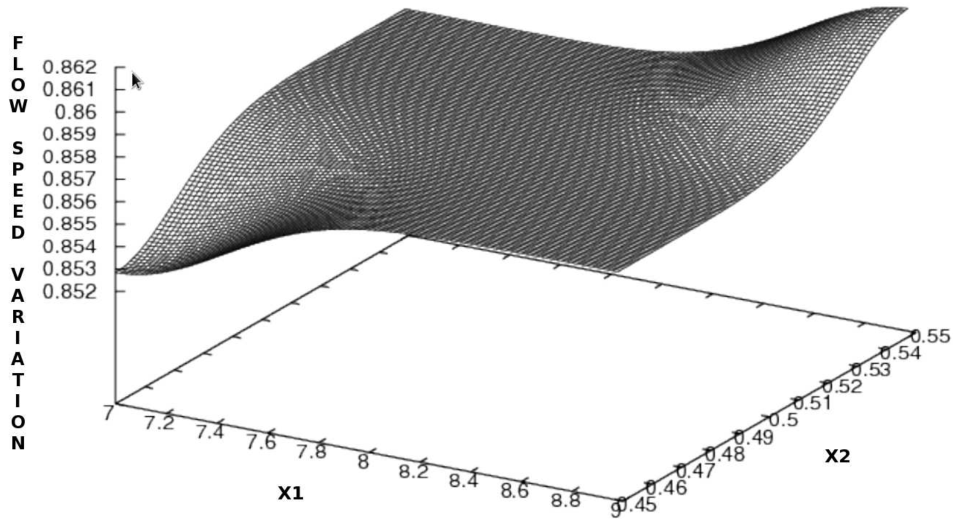


Figure 3.6: Parameter Surface (taken from (93))

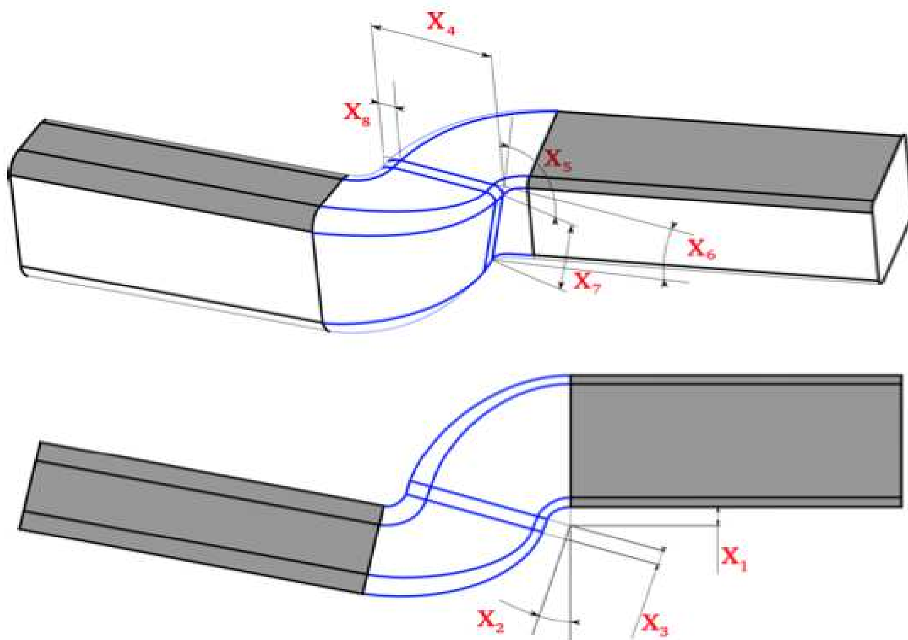


Figure 3.7: Air Conditioning Pipe Parameter Set (taken from (93))

3.6 Exception Types

3.6.1 Practical Exceptions

All computation stages are usually controlled by parameters, so an error in their configuration can determine an unpredicted behavior that the execution platform should detect. In some cases, a wrong configuration at one level can determine exceptions at the following stages, making it difficult to detect the origin of the exception in order to correct it. Most often the effect of these exceptions translates into a very long execution, maybe infinite loops. Based on the description of the Famosa execution platform presented at 3.4 we give some practical examples of errors that can occur at the levels mentioned above. With the help of our colleagues from Sophia Antipolis we have imagined several scenarios that they have translated in the configuration files of a specific application. Here we present the description of each scenario by mentioning the execution stage that has been affected:

- Invalid geometry - The set of design parameters, provided for instance by the optimizer, does not generate a suitable geometry in the CAD module of GMSH. Typically, some surfaces exhibit self-intersections, yielding the failure of the geometry construction process. In that case, GMSH generates no output mesh file.

- Invalid mesh - The mesh generated by GMSH is adapted to the current geometry. For instance, the grid size is automatically reduced in locations where the geometry exhibits a high curvature, to improve simulation accuracy. For some extreme geometrical cases, as it is the case here, the mesh tends to be so much refined at some locations that its size becomes huge and the mesh generation process becomes exceedingly long.

- Divergence problem at solver level - The flow solver is an iterative process that can possibly not converge, if the numerical parameters chosen by the user are not suitable. This choice is not straightforward since it depends on the mesh quality, the numerical methods used, inlet / outlet flow conditions, etc.

- Incomplete convergence at solver level - Even when the flow solver does not diverge, the convergence may not be complete. As for the previous case, if the numerical parameters chosen by the user are not suitable, the residual error which monitors the convergence may stall instead of tending to zero, yielding a solution of poor quality and a large simulation time.

3.6.2 Practical Detection

The responsibility of translating this erratic behavior for the YAWL Exception Service belongs to the custom service in charge of the task's execution where the exception occurred. From the list of exceptions presented so far, we treat in our applications only timeout errors and resource allocations errors. However most of these errors, especially application exceptions like meshing or solver errors, can be translated easily into time-out errors and thus can be treated by our platform. A numerical optimization expert knows that inside a solver task resided a process of convergence. With a proper experience and some apriori knowledge of the application specifics he can estimate an execution time threshold that when exceeded implies a lack of convergence. This is why in such cases we can trigger an exception of the type *timeout*. The same logic can be applied when dealing with the meshing procedure. It can happen that sometimes these computation intensive tasks respect the execution time threshold imposed by the application designer but they are still affected by errors. In this case a solution is to put some constraints on the output results of the operation. In the case that these constraints are not met an exception can be triggered to signal the event. These are just two examples of how application specific faults can be translated in events that YAWL can understand and treat.

3.7 Platform Design Issues

Figure 3.8 represents how our execution platform translates the test case from figure 3.3 using the standard optimization application stages depicted in figure 3.5. This example shows how simple is to model numerical optimization applications using YAWL workflow system. The logical units are easily translated into one or

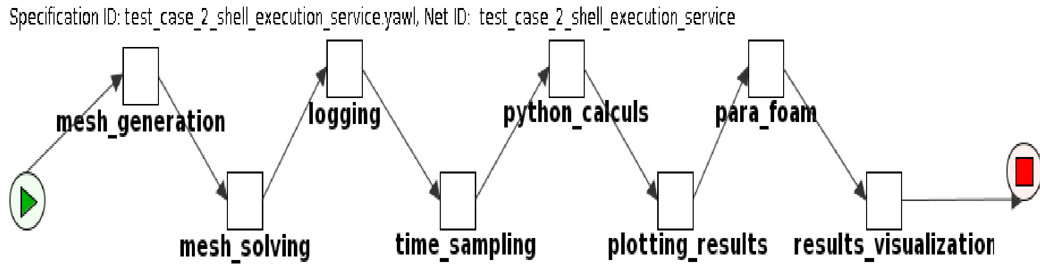


Figure 3.8: YAWL Optimization Abstract Representation

several workflow tasks as is the case for the meshing procedure that is represented by *mesh_generation* and *mesh_solving* tasks. The connections between the tasks are actually a way for transferring parameters values. This architecture ensures a greater flexibility in configuring every logical unit separately from the others and also to better localize an error in the execution chain.

Originally YAWL was not designed to communicate with a distributed computing infrastructure. The already existing services address only aspects like dynamicity and exception handling. Also as described in 2.4.3.3, the internal data management mechanism existing in YAWL is not suited to cope with the amount of data that is usually being transferred when working with distributed computing infrastructures. To address these limitations of YAWL we used a set of tools already present in the YAWL package dedicated to extend the platform when needed. Using the YAWL workflow system we are able to orchestrate the pool of available services so that each task gets associated the service that is suited for its execution. The orchestration is realized with a master-slave architecture. The YAWL engine registers all the available services spread on resources of a computing infrastructure. After associating services to workflow tasks, it delegates work to these services and manages the execution flow (see Figure 3.9). To this purpose there are two main issues:

- Data synchronization
- Resilience

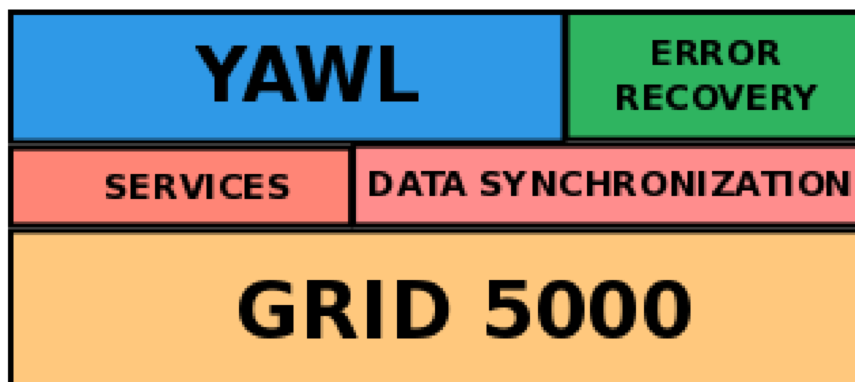


Figure 3.9: Execution Platform Architecture

3.7.1 Data Management and Data Synchronization

At the moment we were implementing our execution platform the data management in YAWL was the one described in 2.4.3.3. This was enough for simple workflow processes for which the input and output data was represented by small application parameters. However the system proved to be limited when dealing with large data flows that had to be transferred from one task to another. Things get even more complicated when tasks execute in a distributed manner and when the computing nodes for each task are located in different clusters. It became clear that a new protocol was needed to handle the data transfer and storage independently of the application's nature.

When we decided to address the data transfer problem we had to choose between two major approaches. The first one is rather decentralized in the sense that the data has to follow the flow of the application as indicated by the workflow description. If the next task will be executed on the same node/cluster as the current one, the cost of data transfer is determined only by the transfer of output data on the local disk. The cost becomes higher if the execution of the next task takes place on a different cluster.

The other solution is to save the data on a central machine (stable storage) accessible by all the other computing nodes. Whenever a task wants to save its output data it will transfer it on the stable storage. At its turn, when a new task

needs input data from the previous ones, it will access the stable storage to get it locally.

In the first method the number of transfers is at the most equal to the number of flow connections between tasks, corresponding to the case where each task is executed on a different site. On the other hand, in the second case every data transfer between two tasks requires two transfers with the central storage. The advantage of the centralized solution is the fact that at each point in the execution the stable storage has a global and coherent state of the application so it suits much better when dealing with faults. If the fault tolerant procedure requires coming back to a previous state of the application, this state can be recovered from the central storage unit. Since this last characteristic meets better our resilience requirements, we chose to implement this second approach.

Starting from the centralized approach we chose to implement a model in which every task has the option of doing a check-out to get data from the central data storage or commit changes in data after its execution. This offers a larger flexibility in choosing a strategy of data backup for resilience purposes, combining the options presented above according to each task's nature and also to important points in the application's semantic. The actual implementation can be done using different existing software. One option is to use a version control system like SVN or GIT. For simplicity and rapidity we chose to use just the Linux based command *rsync*. The details will be presented in section 4.1.

3.7.2 Exception Detection and Recovery Using YAWL

The first step in treating application exceptions is to detect them. For doing that we use the exception handling support offered by YAWL and presented in section 2.4.3.5.

As we have seen in 3.6.1 the exceptions presented there are recognizable by the YAWL system. For detection and signalization we use the Ripple Down Rules system (86, ch.4) integrated in YAWL and described in section 2.4.3.4 of the State of the Art chapter. With it we can obtain the current execution context of the

application and detect that the concerned parameter or timeout value is breaking the rule or set of rules already set for it, thus raising an exception. The system will trigger the right treatment procedure represented by an *exlet* (see section 2.4.3.5 from the State of the Art chapter). Inside the *exlet* the user can take the action he thinks appropriate in order to treat the error and eventually retry an execution of the affected part choosing whether to keep the same computing conditions or adapting them to avoid the same exception again.

Timeout on Task Execution In order to illustrate the exception treatment mechanism presented in the previous section we will use a simple timeout example. Figure 3.10 shows a workflow specification called *Timer_Test* aiming at executing any computation. The first task has a *YAWL Custom Service* associated for its execution and an execution time limit of ten seconds after which an exception is triggered. Using the *Ripple Down Rules* feature of YAWL we've associated a decision tree file to the timeout exception that when triggered will call the execution of an exlet. Inside the exlet we execute a compensation procedure that will indicate to the next task in the main workflow wheather to continue the normal execution or re-execute the first task.

Fault Origin

As shown in figure 3.10 the recovery point is established during the design phase at the beginning of *Timer_Task* task. This means that the user needs advanced knowledge about the application to be able to design the workflow in such a way that the recovery of execution is done at the right point when a specific error occurs. Even though this solution seems cumbersome, doing it differently is very hard. The alternative would be to implement an automatic mechanism that tracks the origin of the fault in the workflow structure and reestablishes the execution flow at that point. The difficulty comes from the fact that a fault occurring during or at the end of the execution of a task is not necessarily a consequence of that task's execution or parameters values. These are a kind of faults that do not provoke an immediate failure but rather propagate in it until they are detected at the level of several tasks after their occurrence. To be able to recover

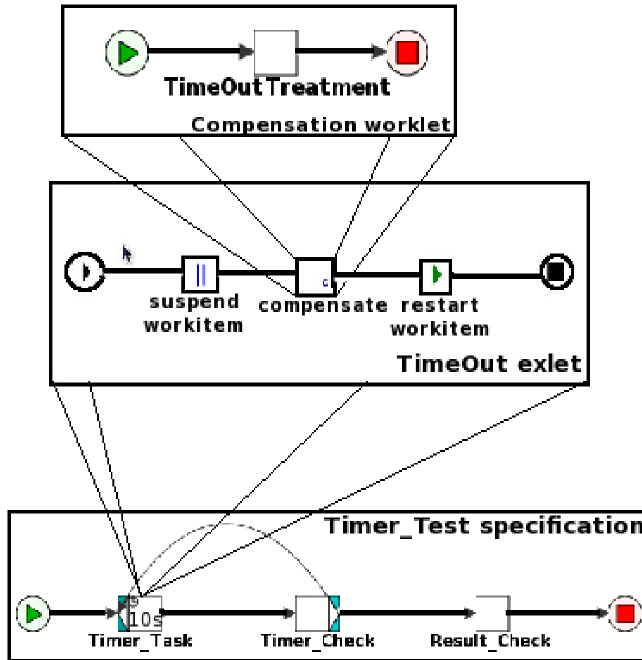


Figure 3.10: Time Out Exception Handling Example

at the right point in the execution the system must know in detail the semantic of the application being executed. This is quite hard to achieve, especially in our case where the execution platform is supposed to support multidisciplinary applications that can greatly vary in context and semantics.

The research topic concerning recovery after failure has been treated since a long time in the literature. For example in (94) the authors want to prove that a reliable way of executing business workflows in the presence of failures is to treat them as transactions. This parallel is especially useful for semantic failures because the workflow system can apply the same rollback mechanism used in database systems that assures reaching a consistent state after a recovery process. However the paper states that it is not possible to present a general solution for workflow recovery because there exist different workflow types which require different recovery approaches.

Continuing the idea of workflow transactional systems, in (4) is presented a deadlock free attack recovery algorithm for coordinated recovery in a distributed transactional system (workflow or database). The emphasis is put on features

like *unrecoverable transactions*, *dependency relations* between tasks/transactions or *concurrency restrictions* during a recovery process. They introduce the concept of *recovery analysis*. This is composed of a *damage tracing* stage where all the damaged tasks/transactions are identified through dependency relations and a *recovery scheme generation* that generates recovery transactions and execution orders according to the result of damage tracing. The workflow model they analyzed is quite theoretical. When trying to apply it for a real domain, like numerical optimization, a lot of obstacles can appear during implementation. In (89) is proposed a prototype recovery system for workflows by the same authors.

In a more advanced system (83) the idea is to use a provenance framework added to the Kepler scientific workflow system that keeps track of data dependencies. The collected data is then used to provide failure recovery.

4. Implementation and Results

4.1 Interface Between YAWL and External Computing Resources

The services that we developed focus on the main activities performed when treating a workflow application: *execution* and *data transfer*. In the following paragraphs we will present details of implementation for each of these services.

1. *Execution Service*

   The first custom service we developed was designed to simply execute a shell command as part of a workflow task execution phase and retrieve its execution status. The aim was to interconnect the YAWL engine can be interconnected with a general external application written in any language. In our case we have used a YAWL Custom Service, which is a java application with HTTP capabilities that was running inside the client's Tomcat server. At the core of each custom service are the following Java classes:

   – A Java servlet interface responsible for receiving event notifications from the engine (*InterfaceB_EnvironmentBasedServer*).

   – Methods that allow a YAWL Custom Service to call specific end-points on the engine side (*InterfaceB_EnvironmentBasedClient*).

   – An abstract utility class that encapsulates much of the functionality of the other two previous classes, designed to be extended by the primary class of each Custom Service (*InterfaceBWebsideController*).

   As required by the YAWL architecture, we first implemented the mandatory method, *handleEnabledWorkitemEvent*, declared in the *InterfaceBWebsideController* which encapsulates all the necessary steps in a task's execution:

- *engine connection* : Using the default credentials (user and password) the service first connects to the YAWL engine before any exchange of messages is done.
- *check-out* : This is an interaction initiated by the service through which it informs the engine that it is ready to execute the workitem. When this occurs, the engine processes the workitems input data, and includes it in the data structure returned to the custom service. The engine also moves the workitem state from *enabled* to *executing*, denoting that a custom service is currently executing the workitem (i.e. the workitem is *in progress*). In our case the input data is represented by the name of the shell script to be executed and the directory where this script is located.
- *execution* : In this step we first extract the parameter values from the data structure with which we construct the command to be executed and we specify also the directory on the computing system where this command has to be executed. Then, by using Java API, we build a new system process that will actually execute the associated work.
- *result mapping*: After execution, we retrieve the status code and we map it in the workitem's output data. This will allow post processing for exception detection or other purposes.
- *check-in* : Another interaction with the engine initiated by the service to indicate that it has completed execution of the workitem. In this step we send also the result data back to the engine, that will move the workitem from *executing* to *completed* state.

As the execution platform evolved we had to modify this service in order to accommodate new requirements mainly related to integration on a distributed infrastructure(e.g. Grid5000). The most important one is the specification of a data transfer activity along with the direction of transfer(not needed when running both the YAWL engine and client on the same machine)
The final objective for developing such a service was to facilitate a distributed collaborative execution in which each computation step can be performed

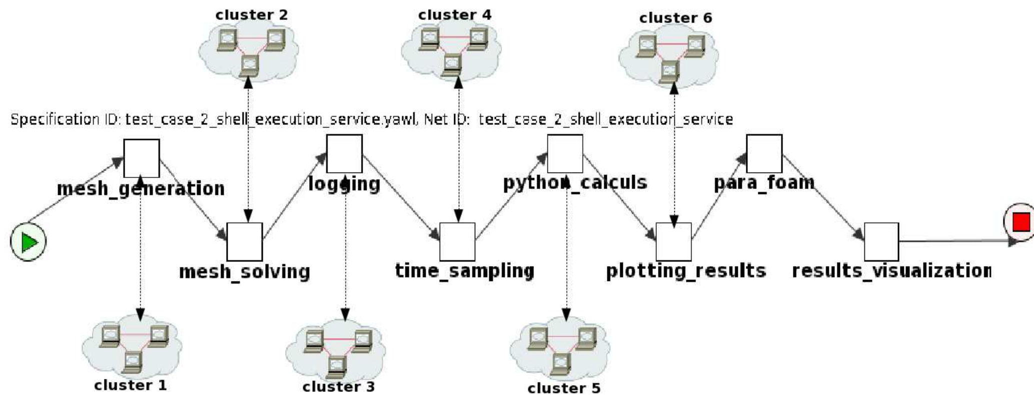


Figure 4.1: YAWL Distributed Collaborative Execution

on a different cluster according to the machine architecture and operating system needed (see Figure 4.1).

2. *Data Transfer Service*

This is not an independent YAWL custom service from a physical point of view but one integrated in the previous service. However it deserves a separated presentation because it is independent from a logical point of view and can be integrated in any other YAWL custom service. As the name indicates, its main functionality is to transfer data produced by different tasks so that it is available to every future task that will need it. The approach followed uses a stable storage location to store the data globally and available for every cluster and an *on-demand* data transfer so that every task decides independently whether it needs a *data update* transfer from the stable storage before execution or a *data store* transfer after it has finished execution and produced new data. Every application will have a global directory representing the data and it is supposed that every task in the application knows the location of this directory on the cluster on which it executes, as well the internal structure of the directory so that any information can be traceable using a combination of absolute and relative path location. The different steps performed by the data transfer service when invoked are presented in the following pseudo-code:

Algorithm 1 Perform Data Transfer

```
transfer_direction ← getTransferDirection(workitem_data)
yawl_transfer_codelet_id ← findDataTransferSpecId(session_handle)
switch (transfer_direction)
case LOCAL_TO_REMOTE:
    transfer_case_id ← launchCase(yawl_transfer_codelet_id, codelet_data)
    waitTransfer()
    result ← execute(workitem_reference)
    break
case REMOTE_TO_LOCAL:
    result ← execute(workitem_reference)
    transfer_case_id ← launchCase(yawl_transfer_codelet_id, codelet_data)
    waitTransfer()
    break
case LOCAL_TO_REMOTE_TO_LOCAL:
    transfer_case_id ← launchCase(yawl_transfer_codelet_id, codelet_data)
    waitTransfer()
    result ← execute(workitem_reference)
    transfer_case_id ← launchCase(yawl_transfer_codelet_id, codelet_data)
    waitTransfer()
    break
default:
    result ← execute(workitem_reference)

end switch
```

Reading the pseudo-code from algorithm 1, we can distinguish four main possibilities:

(a) A new task retrieves data from the stable storage then performs the execution. Any new data produced on the local directory is not transferred on the stable storage. This corresponds to the situation when the new data is needed by a future task that will execute on the same cluster and no other task from a different cluster will need it.

(b) A new task performs the execution then stores the produced data on the stable storage. This corresponds to the situation when the last updated data was already on the executing cluster so there is no need in retrieving it from the stable storage, but future tasks executing on different clusters will need the updated data produced by the current task.

(c) A new task has to first retrieve the updated data from the stable storage, perform execution and then update the stable storage with the new produced data.

(d) A new task has to execute only without any data retrieve or update.

In the above explication of the data transfer pseudo-code, the data storage is respresented in our case by the local machine where the YAWL engine is installed and the data transfer is actually performed using *rsync*. Grid5000 has a quite strict security policy forbidding any communication initiated from the inside of the infrastructure to the external world, including any data transfer procedure too. That is why we had to initiate the transfer from the outside world, the stable storage in our case. To do that we use a command execution codelet, *rsync* being the called command. The codelet is triggered by a YAWL Custom Service that is running on one of Grid5000 nodes. The entire protocol is presented in figure 4.2.

4.2 YAWL and Grid5000

The Grid5000 general architecture has already been described in section 2.1.5 and remembered in figure 4.3. In the following we denote:

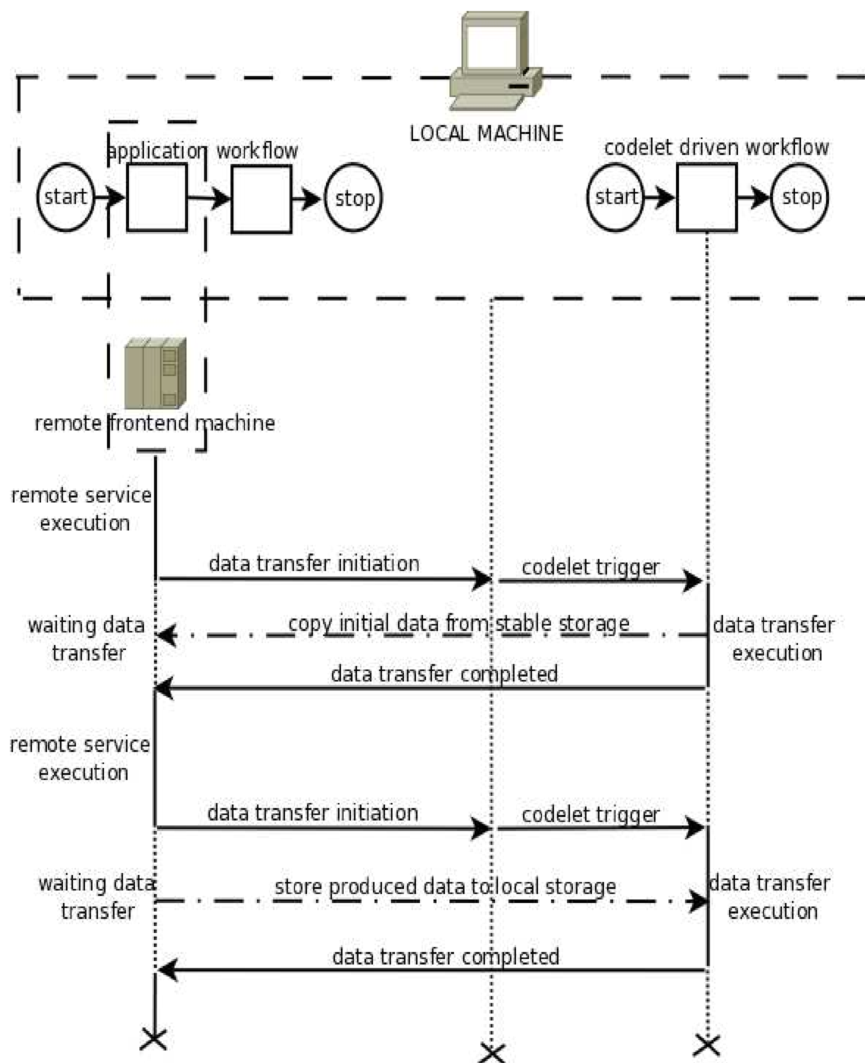


Figure 4.2: Data Transfer Protocol

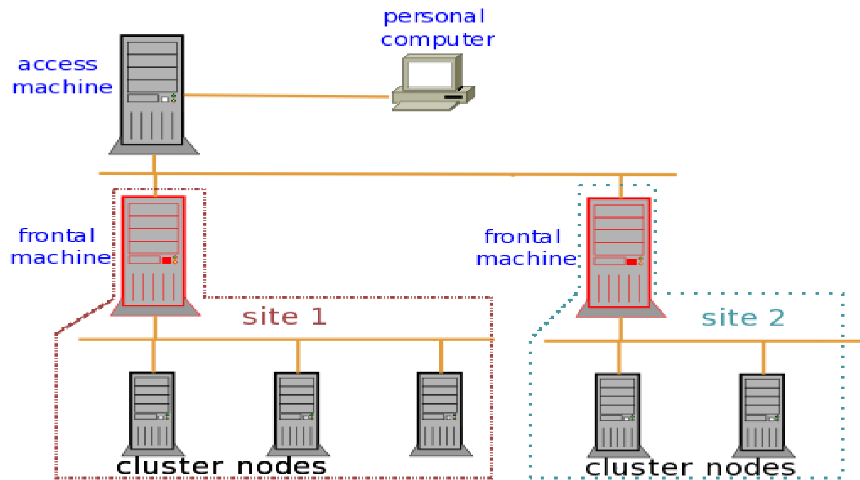


Figure 4.3: Grid5000 Default Architecture

- site - geographical related machines
- cluster - architecturally homogeneous set of machines inside a site
- node - a machine within a cluster that can have multiple processors while each processor can have multiple cores

To emulate the YAWL engine-service communication on this architecture, the placement of the different YAWL components would have been the following:

- YAWL engine is installed on the user's personal computer (hereafter called the *local machine*)
- YAWL Custom Services are installed on each cluster's frontal machine
- jobs are launched by the custom service on the computing nodes

There were two main drawbacks using Grid5000 in this form:

1. A service had to run continuously on the frontal machine. As this is a multi-user machine, this could have an important impact on its performance, especially if more than one service is needed on one frontal machine and would violate the terms of use established by the founders of Grid5000.

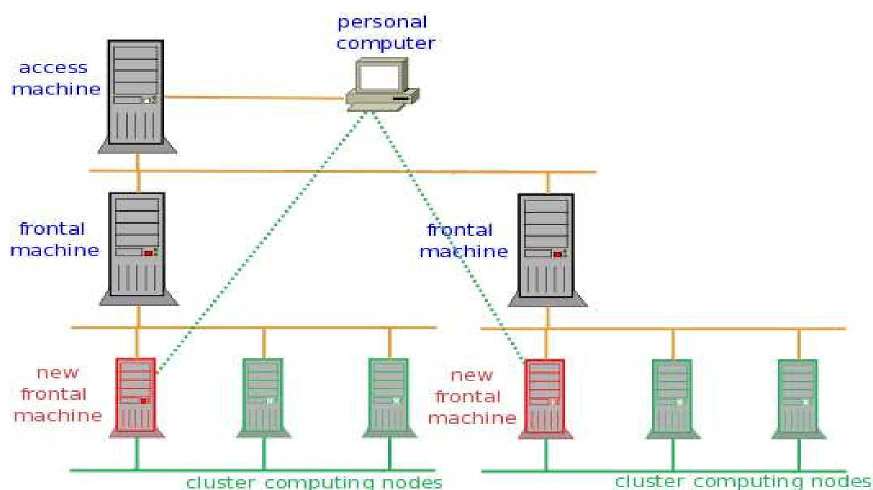


Figure 4.4: Grid5000 Yawl Adapted Architecture

2. Every time a new task was executed, the service on the frontal machine had to perform a custom environment deployment, containing all the software tools needed for the application execution (procedure that will be described in section 4.5), that takes a significant amount of time.

4.2.1 Interface Between YAWL and Grid5000

To avoid this, the technical staff of Grid5000 proposes a solution in which the user recreates the cluster structure at a smaller scale, using only the number of nodes he estimates that will be needed for the application. In this way one of the nodes acquired will play the role of the frontal machine dedicated to only one user. Also, the custom environment mentioned above will be deployed only once for all the nodes as a pre-configuration procedure before the actual execution of an application. Thus the default frontal machine won't be held occupied by any custom service and the deployment time is separated from the execution time. This new architecture is graphically described in figure 4.4 and in grater details in section 4.5.

The communication between the YAWL engine and the custom services is done through a Tomcat server. With the described architecture, this would require

that the HTTP ports on the frontal and access machines are left open, which is against Grid5000 security policy that blocks every communication to the external network. As a consequence we had to use SSH port forwarding in order to wrap the HTTP messages in SSH packets. Problems were in the opposite direction as well. Our user-machine (where the YAWL engine was installed) was protected also by a firewall machine. To reach it through SSH we had to use multi-hop SSH port forwarding.

4.2.2 Resource Reservation and Deployment on Grid5000 Infrastructure

The deployment phase is actually included in the recreation of the cluster structure and all is embedded in a deployment script. Before launching this script, a custom environment is created. By environment in Grid5000 we understand an operating system and a set of programs. Creating custom environments is a feature of Grid5000 that allows users to adapt a pre-existing environment to their needs that will replace the default one installed on the nodes. Thus the users can control the entire software stack for experiments and reproducibility. In our case this was useful because on the default environment we didn't have enough rights to install the software stack needed for our experiments. As an example, on our environment we installed software tools like YAWL, MPI, OpenFOAM, etc. The main logical steps performed are the following:

- Acquisition of a predefined number of nodes in a cluster.
- Deployment of the custom environment on these nodes.
- Configuring a root node on which administration tools are installed. This node will be the new frontal machine.
- Configuring the other nodes as computing nodes.

After this phase, we start the tomcat servers on the user machine and on each of the frontal machines. The first one will actually start the YAWL engine that will guide the application execution. The other ones will deploy the custom services necessary for individual task executions. Then, using multi-hop SSH port

forwarding we established the communication channel between the YAWL engine and each of the custom services. Now the YAWL engine is able to register all the available services so that the user can assign to tasks composing the application the right service from the list.

Figure 4.5 describes an updated scheme of resource reservation and deployment more suited for large scale application where a significant number of services is needed for a workflow execution. In this configuration both node reservation and deployment is launched from a common script placed on a chosen frontal machine of one of Grid5000 site, the script being triggered from the local machine. The allocation is done in a loop until the desired number of nodes is obtained. This loop uses a configuration file where the user has specified for each desired Grid5000 site how many nodes to reserve per iteration (PHASE 1). Once all the nodes are acquired the deployment phase begins. The custom environment archive is located uniquely on the same frontal machine as the script and sent through HTTP to all previously reserved nodes (PHASE 2). At the end of deployment a file will contain addresses of nodes that have been correctly deployed and another file will be filled with the failed ones. Then custom services can be sent on each of the proper nodes and the Tomcat server is started so that the YAWL engine can communicate with them using multi-hop SSH port forwarding (PHASE 3). So this time every deployed node contains a different Yawl Custom Service entity being used at the same time as an interface with the YAWL engine and a computing node. This is useful when dealing with many small tasks. If the workflow contains also computing intensive tasks, we can associate several nodes and parallelize the task using MPI on them.

4.2.3 Distribution of Computation on Grid5000 Infrastructure

We tested the distributed capabilities of our platform with a workflow application that contains two branches modeling the *air-conditioning pipe* test-case (see section 3.3). The execution of the computing intensive tasks of each branch is assigned to custom services located on clusters of Grid5000 with no communication line in-between. The output files of these tasks are then transferred on

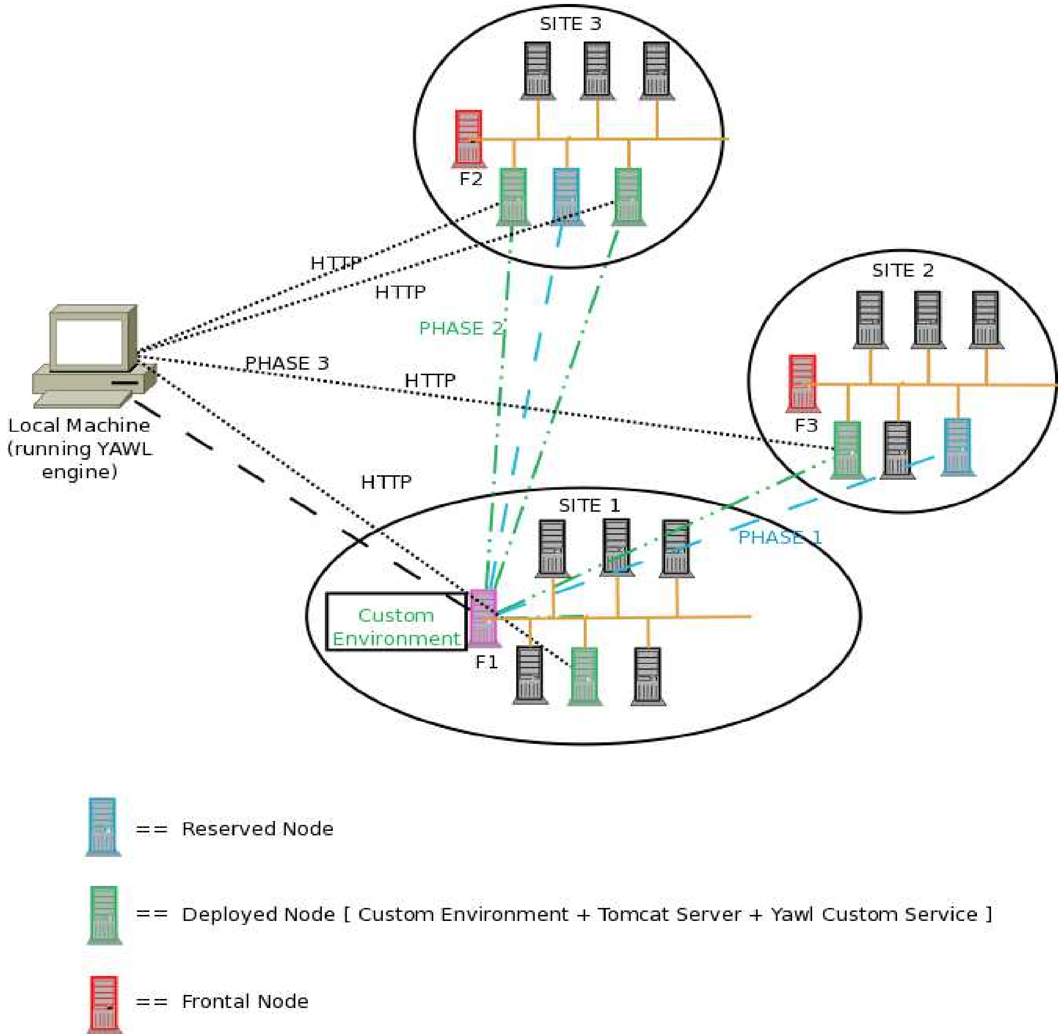


Figure 4.5: Grid5000 Resource Allocation and Deployment

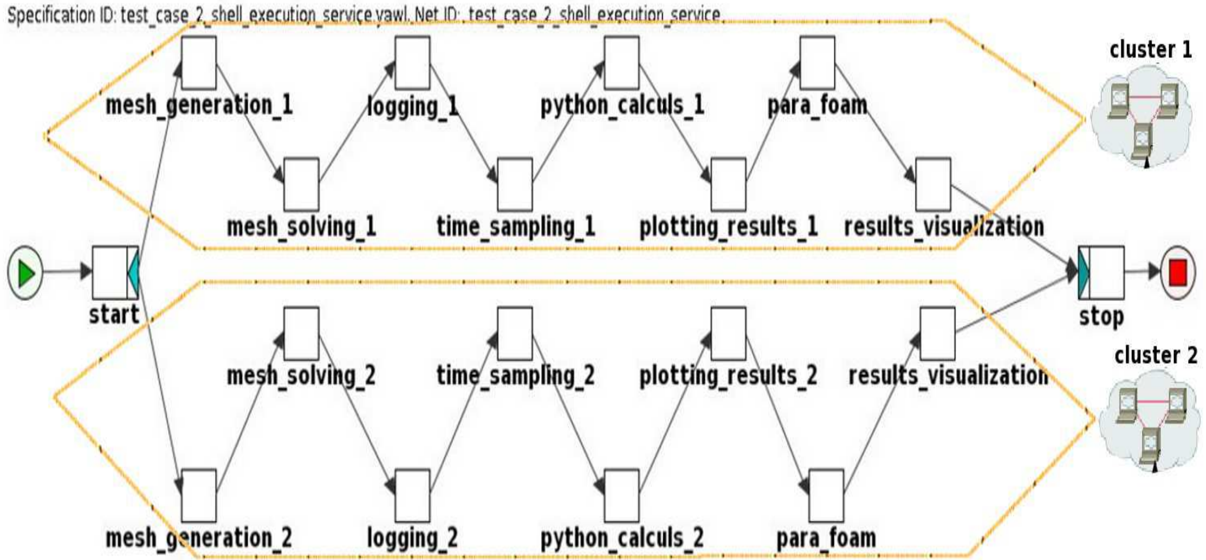


Figure 4.6: Grid5000 Distributed Test Case 2

the local machine and used as input data for the following tasks (see figure 4.6). The purpose is only to show the platform's capability to execute applications in a distributed manner as well as data transfer between the engine and the services. One example of useful case would be to execute the same application on two different clusters with different parameters and then compare the results.

4.2.4 Passing to a Larger Scale

In previous sections we presented how YAWL was configured to communicate with Grid5000 in order to execute workflow tasks on grid resources. The main component that assures this communication is the YAWL Custom Service. The biggest issue when deploying the infrastructure is the important amount of manual actions that have to be performed. Thus, every Custom Service has to be deployed on a grid cluster inside the custom environment previously configured, then the ssh port-forwarding command must be launched to let HTTP messages circulate between the YAWL engine machine and the clusters and finally the Custom Service has to be registered within the YAWL engine. Although time

consuming, all these actions are feasible when dealing with experiments that require only few clusters, but the situation changes radically if an experiment needs a considerable number of tasks executed in parallel. Such a large scale experiment can be modeled in the YAWL workflow language using the concept of *multiple instance atomic tasks* and *multiple instance composite tasks.* They allow the user to run multiple instances of a task concurrently. The user can fix several parameters for a multiple instance task like the following:

- *Minimum Instances* - Represents the minimum number of instances of the task that will be started when the task is activated.
- *Maximum Instances* - Represents the maximum number of instances of the task that can be created.
- *Continuation Threshold* - If the number of instances created exceeds this parameter and the amount equal to this parameter have completed, the multiple instance task itself is considered complete and will trigger relevant outgoing flows from it.

Beside these three parameters, the designer can also specify the *Instance Creation Mode* that can either be static or dynamic. Static means that the number of instances initially created cannot vary once the task is activated while in the dynamic mode more instances can be created even after the activation of the task. In figure 4.7 we modeled in YAWL language the test-case described in section 3.5 of the previous chapter using *multiple instance composite tasks.*

However this approach was design by YAWL developers to run several task instances but on a single service (i.e. machine). This service is regsitered manually through the YAWL administration GUI. In our case, if we want to use 100 Custom Services deployed on 5 different clusters we have to register manually the services with a YAWL engine one by one which can be very inconvenient.

The solution came from the YAWL features and its interface system. Thus a YAWL user has been provided with possibility to implement a special interface,

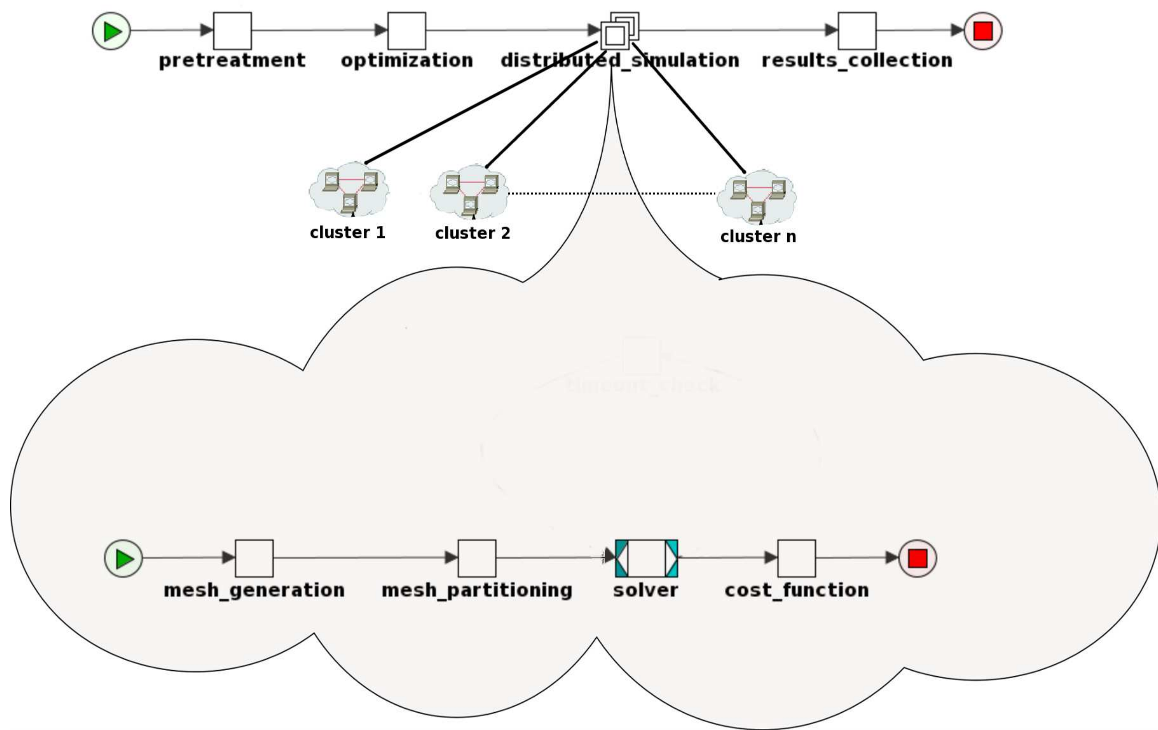


Figure 4.7: Large Scale Optimization Process

Figure 4.8: Classic Style YAWL Custom Service Implementation (taken from (85))

called *Observer Gateway* that allows multiple Java objects, Custom Services included, to register interest with an instance of a YAWL engine so that it receives notifications regarding when an atomic task's workitem of a workflow application becomes available for execution. The registering object has to provide a set of listener style methods that are activated by the reception of events sent by the Observer Gateway implementing class. Since the call into the listener method is executed on the YAWL's run-time thread, it is better to use dedicated threads for every such call. Using the Observer Gateway has several advantages of which we mention:

- the necessity to register each service URL with the YAWL engine is removed.
- with a well implemented manager application interacting with the engine, the communication between a Custom Service and the YAWL engine can be done via any form of net protocol.

Figures 4.8 and 4.9 show the difference between the standard way of managing Custom Services when each service had to register with the service individually and the one proposed by the implementation of an Observer Gateway manager class.

For the implementation part we adopted as a model the *InterfaceB_EngineBasedClient* class that already implements the *ObserverGateway* interface. This way the event announcing system was already implemented. To this we added a queuing system that is holding references to all the available YAWL Custom Services in the

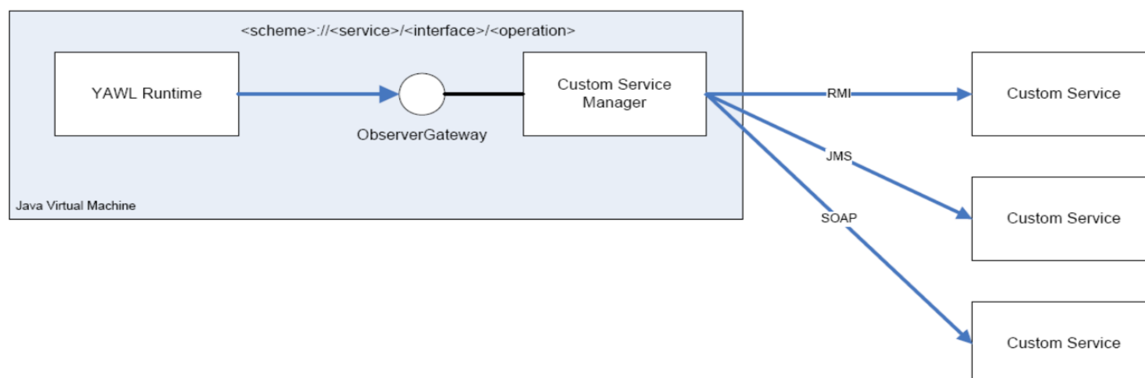


Figure 4.9: Observer Gateway YAWL Custom Service Implementation (taken from (85))

computing platform. The tasks' stack is managed internally by the YAWL engine so no structure had to be added to this purpose. Additionally a dictionary keeps associations between workitems and services so that when another task has to be executed by the same service it will be put on hold until the current task is executed. The liberation of a service is done when a *status complete* event is announced for the workitem currently being executed by the demanded service. Figure 4.10 displays the main stages of the new task execution process described above.

4.3 Resilience: Scenarios and Implementation

The following test-scenarios were imagined to illustrate the distributed, inter-disciplinary aspect of our platform along with its ability to recover execution after an application exception occurs. These scenarios also put in evidence the evolution stages of the implementation. All of the presented scenarios are based on the 3D car air-conditioning duct test-case described in section 3.3.

4.3.1 Sequential Execution on Multiple Different Machines

The first scenario (figure 4.11) involves the execution of the application respecting the sequence of execution of the composing tasks but each on a different

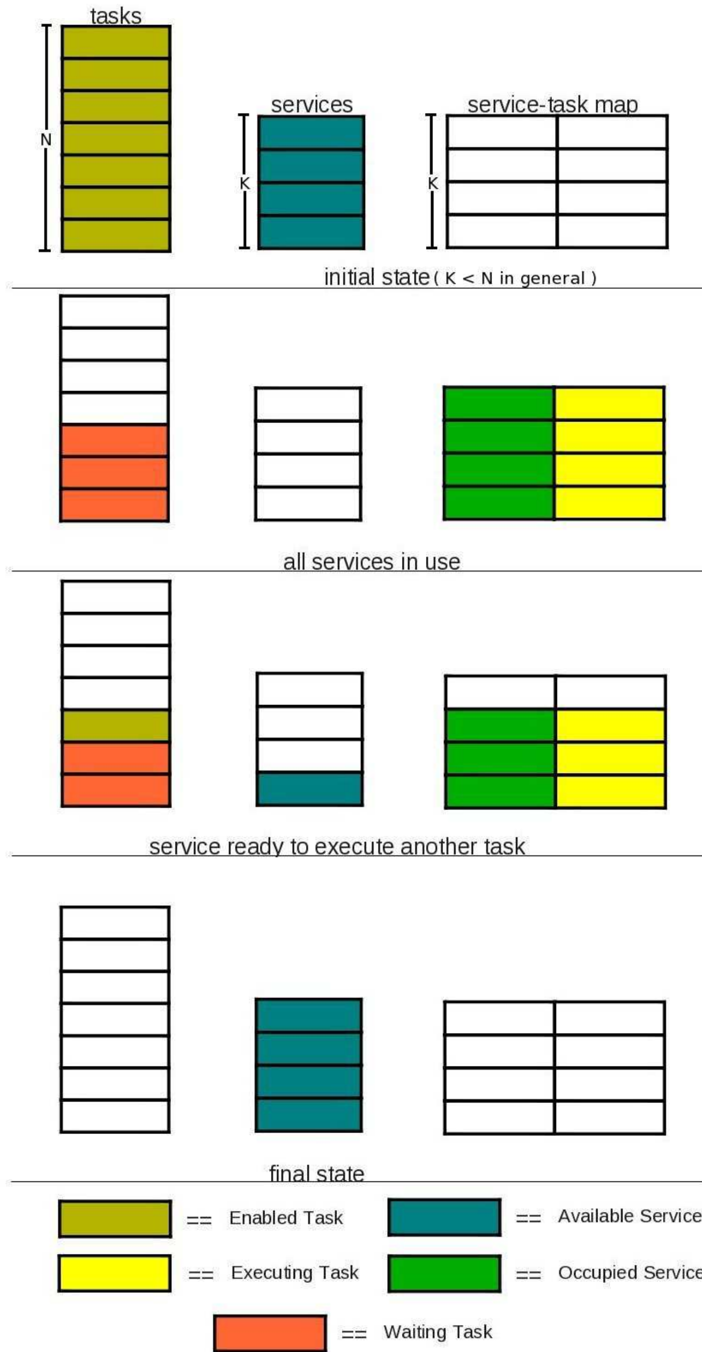


Figure 4.10: Task - Service Association

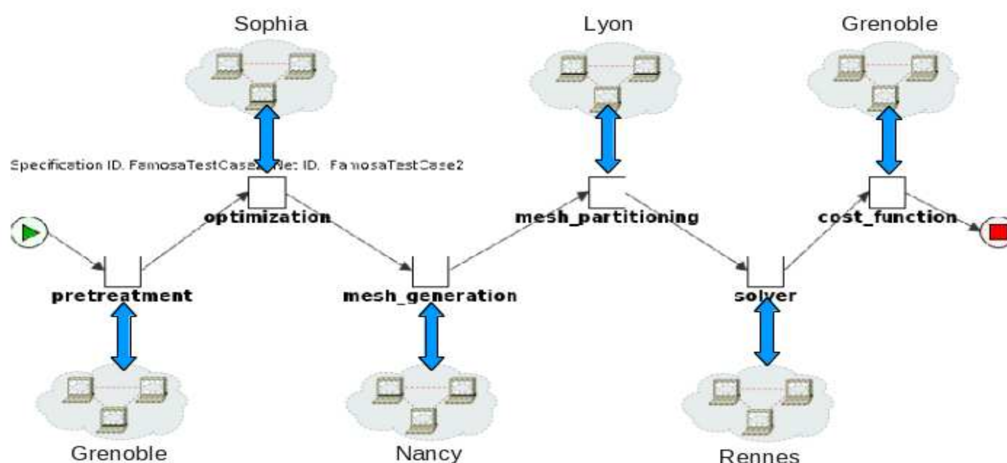


Figure 4.11: Test Case 2 Multiple Cluster

cluster of the Grid5000 infrastructure. As explained before, most of the scientific applications are interdisciplinary by nature and most often the tools required to execute the application are distributed on proprietary sites which do not always coincide geographically. This scenario aims to show that our platform supports such a configuration by associating execution of tasks to clusters that are known to have the needed tools to accomplish their execution. The same execution service has been deployed previously on all the clusters concerned leaving to the YAWL engine the orchestration of these services.

The set of images in figure 4.12 represent snapshots of the outcome given by the linux *top* command launched in parallel on both clusters (Grenoble and Sophia Antipolis) on which the application is being executed. The name of the command currently executing is highlighted, indicating the associated task in the workflow description. In this scenario, at any given time, there is only one active cluster. Note that the *Num3Sis* task has four processes in parallel due to its multi-threaded nature.

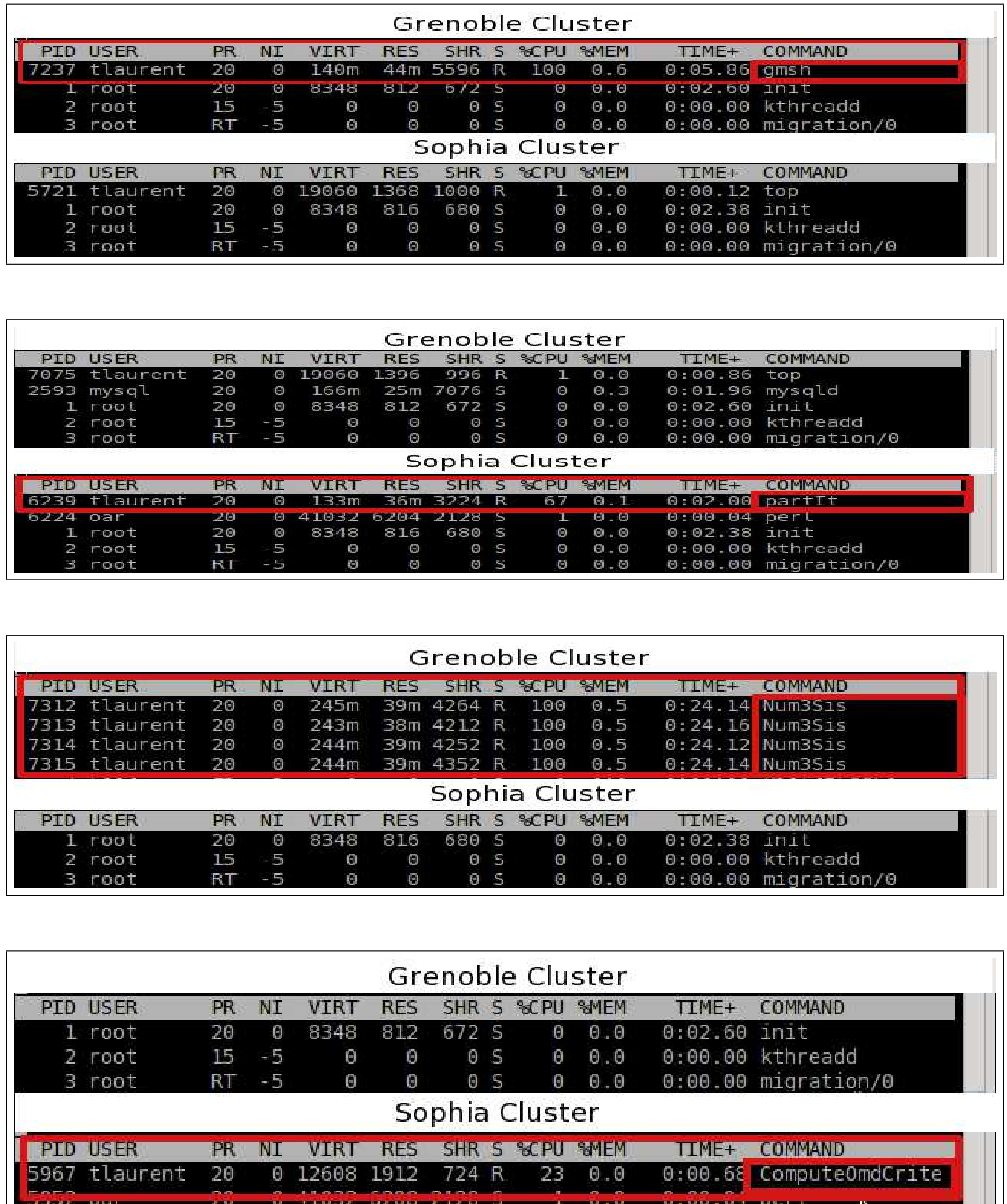


Figure 4.12: Sequential Execution on Multiple Clusters Visualization

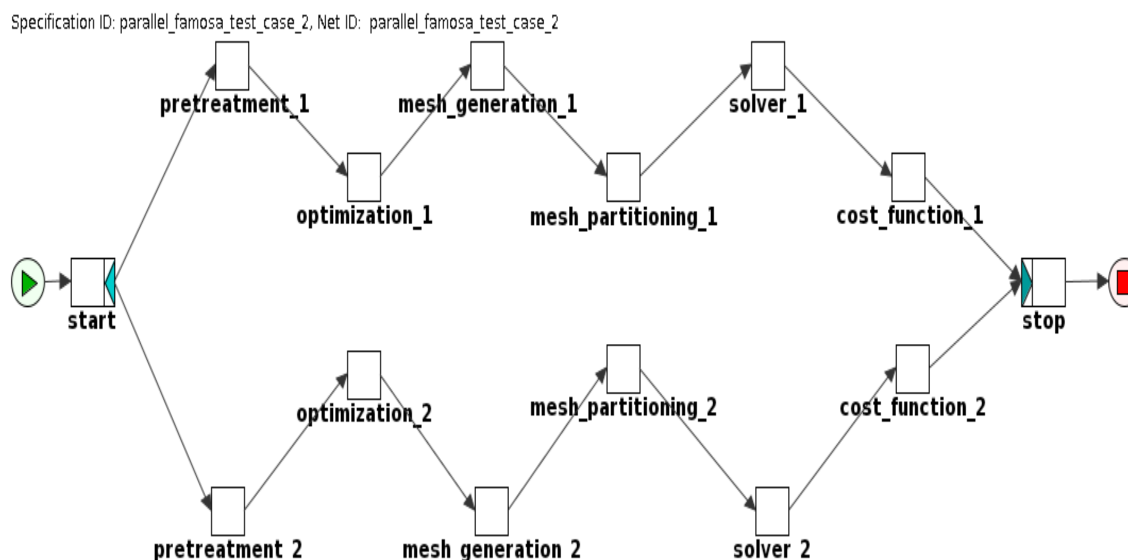


Figure 4.13: Parallel Execution - Test Case 2

4.3.2 Parallel Execution on Two Different Clusters

Sometimes it is useful for a scientist to compare performance of two different solver tools running at the same time. To illustrate we designed the scenario in figure 4.13. It represents the parallel execution of two instances of the same application on two different clusters. Such a configuration can also be seen as an example of a fault tolerance strategy, namely redundancy. When the execution of an application, or just a part of it is critical, we can distribute the execution on independent clusters and choose the result by respecting the criteria of correctness and time. Finally, a scientist can imagine such a scenario when he wants to execute the same application but with a different configuration regarding the values of parameters involved. At the end of the execution he can make a comparison study to evaluate different aspects of the application: result values, platform dependency, convergency, etc.

The same type of snapshots (figure 4.14), as presented for the previous case, are taken for this application too. This time the two clusters are active both at

Grenoble Cluster

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
7976	tlaurent	20	0	101m	7732	5156	R	26	0.0	0:00.78	gmsh
7959	oar	20	0	41164	6212	2128	S	1	0.0	0:00.04	perl
1	root	20	0	8348	812	680	S	0	0.0	0:02.70	init
2	root	15	-5	0	0	0	S	0	0.0	0:00.00	kthreadd

Sophia Cluster

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
7650	tlaurent	20	0	101m	7732	5156	R	30	0.0	0:00.90	gmsh
7633	oar	20	0	41032	6208	2128	S	1	0.0	0:00.04	perl
1	root	20	0	8348	812	680	S	0	0.0	0:02.40	init
2	root	15	-5	0	0	0	S	0	0.0	0:00.00	kthreadd

Grenoble Cluster

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
8656	tlaurent	20	0	245m	39m	4160	R	94	0.2	0:02.82	Num3Sis
8658	tlaurent	20	0	244m	39m	4144	R	94	0.2	0:02.82	Num3Sis
8657	tlaurent	20	0	243m	38m	4148	R	93	0.2	0:02.80	Num3Sis
8659	tlaurent	20	0	244m	39m	4256	R	93	0.2	0:02.80	Num3Sis

Sophia Cluster

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
8319	tlaurent	20	0	218m	39m	4084	R	100	0.1	0:03.90	Num3Sis
8320	tlaurent	20	0	217m	38m	4040	R	100	0.1	0:03.88	Num3Sis
8321	tlaurent	20	0	217m	38m	4068	R	100	0.1	0:03.90	Num3Sis
8322	tlaurent	20	0	218m	39m	4156	R	100	0.1	0:03.90	Num3Sis

Figure 4.14: Parallel Execution on Multiple Clusters Visualization

the same time because they are executing different instances of the same test-case in parallel.

4.3.3 Sequential Execution with Timeout Exception Handling

A first step before executing this scenario on the real application was to integrate in the platform and test the YAWL Exception Service presented in chapter 2 section 2.4.3. We developed a test-case (figure 4.15) in order to better understand how this service works. The purpose was to generate a timeout exception during the execution of a workflow task that would be caught by the YAWL Exception Service. When this event arrives, an exlet is triggered that would launch a compensatory worklet. The figures below represent the main workflow

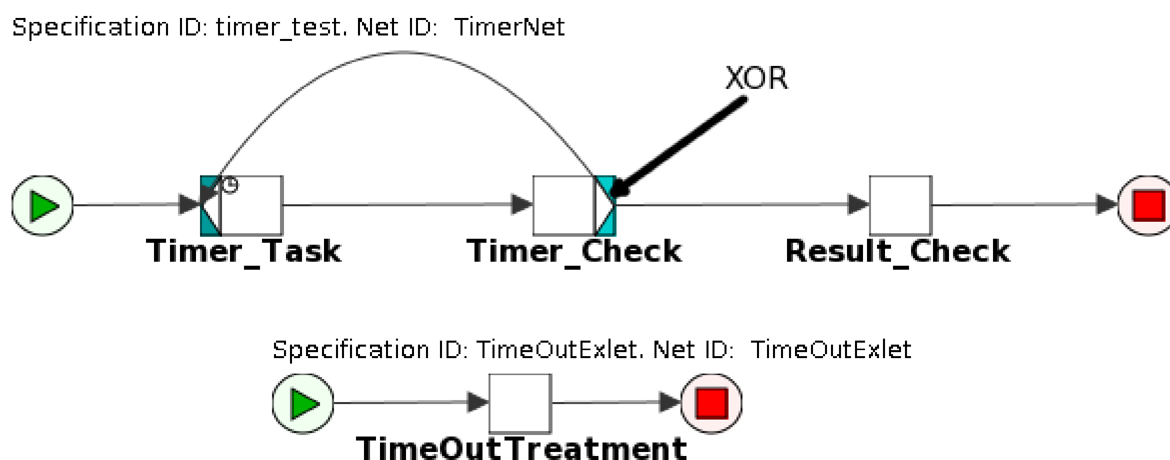


Figure 4.15: Timeout Test and Exception Treatment

specification and the exlet respectively. The task *Timer_Task* has associated a timeout of 10s and for its execution a shell execution YAWL Custom Service is in charge. Beside the common variables for a shell execution, there is also *nr-Restarts* that will be used to decide whether to stay in the restarting loop or break out of it. The *Timer_Check* task is decorated with an XOR split with two branches: restart and continue. A Ripple Down Rule is created for the main specification (called *timer_test.xrs*). This rule specifies that if a timeout exception occurs for the *Timer_Task* task then an exlet is enabled that will suspend the current workitem, execute a compensatory worklet and then continue the execution of the workitem. The compensatory worklet (*TimeOutExlet.yawl*) contains only one task called *TimeOutTreatment* in which the *nrRestarts* variable will be incremented by 1. The *TimeOutTreatment* task must be associated with the custom service *timeOutExceptionHandlingService* that will perform the increment.

The next step was to design a workflow (figure 4.16) for the second test-case that would integrate a dump and restore service for a computing-intensive task, *mesh_solving*, that will archive on the local storage the input data used by this

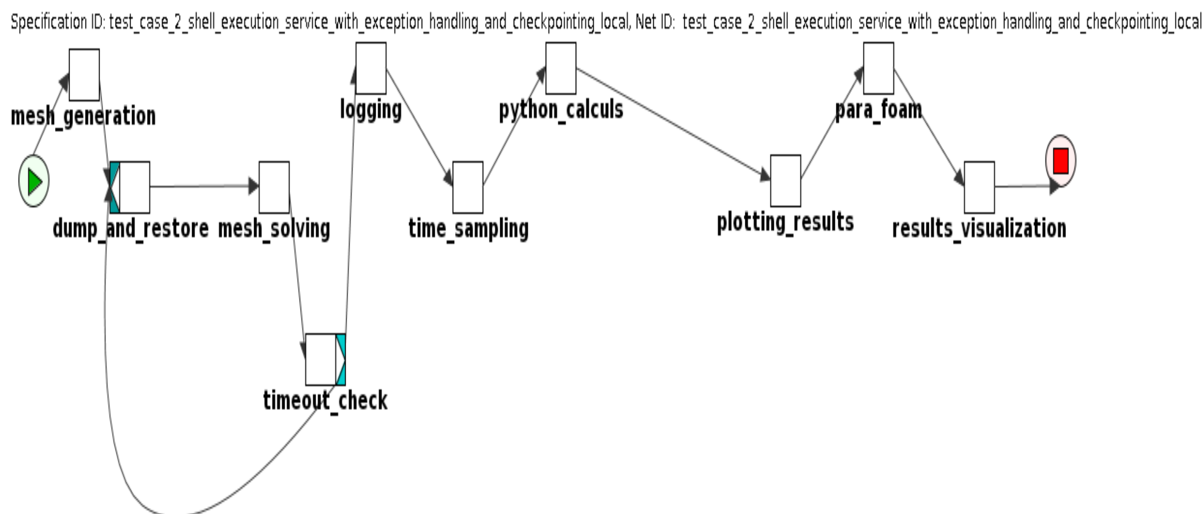


Figure 4.16: Exception Treatment with Dump and Restore Features

task. Also we provide a global parameter, *timeout*, storing information about the maximum time allowed for the task to execute. If the task exceeds the time limit, a global result parameter will be set to 1, otherwise to 0. The *timeout_check* task has the exception service activated and tests the value of the result parameter set at the previous task. If it's 1, an exception treatment procedure will be triggered with an exlet included in which the user can modify the value of *timeout* parameter or any solver parameter that influence the computing time (e.g. the maximum number of iterations) and re-execute the *mesh_solving* task. Before re-execution, the old state is restored by *dump_and_restore* task.

4.3.4 Distributed Execution with Resource Balancing

The last scenario, and most evolved, aims to regroup in one workflow application all the concepts already presented like: distributed parallel execution, data transfer, exception handling, but also to introduce a new one, resource balancing, that increases the level of flexibility for the user. More precisely a scientist can decide, according to the nature of the application, if fast execution is more important than saving resources or just accept a slower execution when not enough

resources are available. He can also increase the computing power at run-time in case more resources are needed but they were not available when the application was initially configured.

In figure 4.17 is presented the workflow description of the air-conditioning duct test-case (same as the one in figure 4.11) with some modifications that allow the parallel execution of the *solver* task on two different clusters with different computing resources (e.g. in terms of memory size or computing power). The small cluster contained 1 node with 2 processors and 4 cores per processor while the big cluster contained 2 nodes with 2 processors per node and 4 cores per processor. We deployed a YAWL execution custom service on each cluster. The internal architecture of the solver task allows parallelization of the execution using MPI by specifying the number of processes to create in parallel. This means that the fastest execution will be achieved when the number of processes created by MPI will be equal to the number of cores available on the cluster. In order to put in evidence the capability of the platform to switch between different clusters when executing a task, we set a timeout execution parameter on the *solver* task with a value that is too small for the 8 cores cluster and big enough for the 16 cores cluster. We also enabled the YAWL Exception Service in order to catch a post-constraint exception generated by the timeout expiry. Another parameter decides which cluster to choose for execution. So, at the beginning the system will choose the execution of the *solver* task on the 8 cores cluster. When the timeout will expire, an exception will be generated and detected by the Exception Service. This will trigger the exception treatment procedure in which we change the cluster. This way we can opt for the 16 cores cluster and re-execute the *solver* task. Because of a faster execution time, the timeout exception will no longer be triggered, allowing the execution of the application to end normally. On the figure one can also notice at the level of which tasks a data transfer procedure is performed and in which direction.

A part of the experiment involved also some measurements to see how execution time varies according to the number of resources used, among those available on the cluster. In this case we required for a total of 16 processes. In this way we

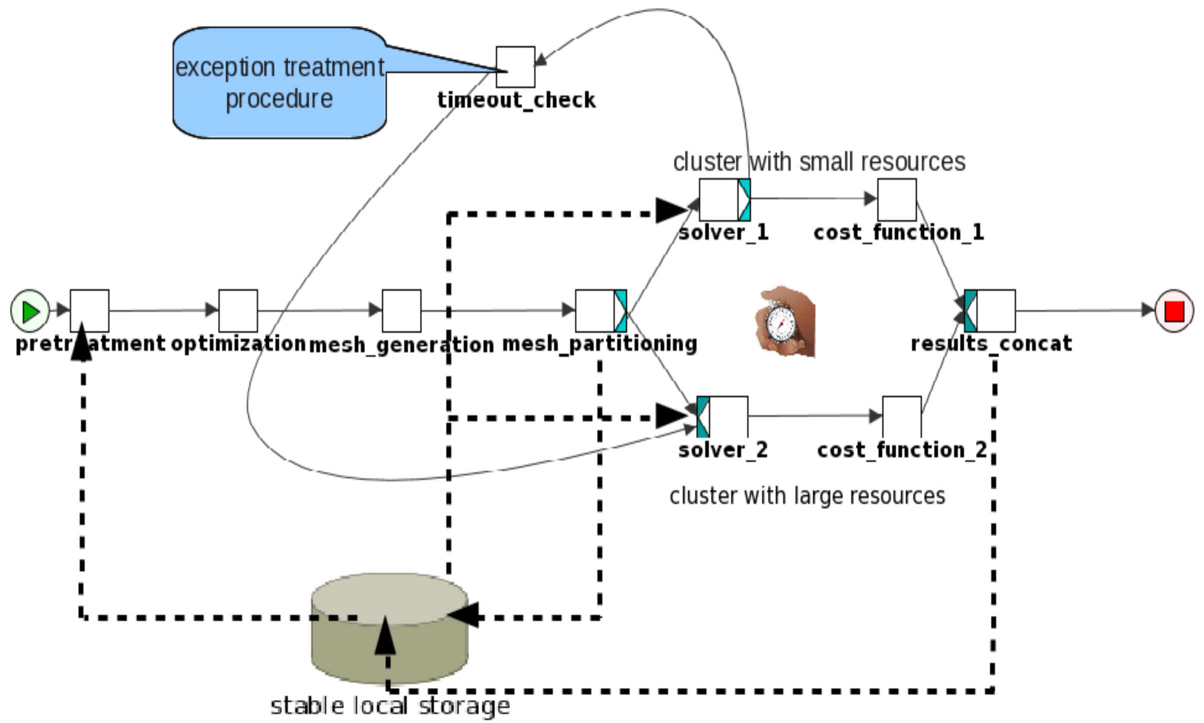


Figure 4.17: Distributed Resource Balancing

could notice that the execution time was decreasing when increasing the number of processes, K, on condition that K was inferior to the number of cores available on the cluster. When this condition was not met any more, the execution time started to increase again. The results presented in figure 4.18 show that the minimum execution time is reached when the available resources are fully used (8 cores in the left part and 16 cores in the right part). However, we notice that the absolute minimum time is obtained in the right part of the figure when the number of processes equals the number of cores in service. Any under-usage or over-usage of the resources causes this time to be larger.

4.3.5 Speed-up Gain in Large Scale Optimization Application

In section 4.2.3 we described how our platform has been extended to support a very large scale execution (typically involving hundreds of machines on 5 different sites). This allowed us to conceive a large scale test-case that uses computing resources at grid level. We added also exception treatment capabilities that with the help of some important YAWL features didn't require radical changes compared to the previous test cases.

The description of the test case has already been presented in the previous chapter in section 3.5. The YAWL workflow description is presented in figure 4.19, top part. The first two tasks correspond to the optimizer code mentioned in the description section 4.2.3. These will produce the N different files containing geometrical parameters. The next task, *distributed simulation*, is a multiple instance composite task meaning that it will contain multiple workflow sub-nets. Each of these sub-nets represents a full simulation process containing tasks for mesh generation, mesh partitioning and solver as depicted in figure 4.19, bottom part. Like in previous test cases, the solver task (the most computing intensive one) has an exception treatment mechanism associated. Every simulation branch is independent from the others with its own set of parameters. Thus if an exception appears on one branch the treatment procedure will not influence the other branches that can continue their execution unhindered. Of course, the service associated with problematic branch will be blocked till the branch successfully

completes. However, the experiment designer has the possibility to specify how many branches from the maximum available have to successfully complete so that the multiple composite task from figure 4.19 is considered completed.

To implement the test-case described above we first created an environment that installed all the necessary software tools (including the Tomcat server) to correctly execute the application. Also the environment will contain the YAWL Custom Service needed for tasks execution. Using the schema depicted in figure 4.5 we allocate as many nodes as we wish from Grid5000 and we deploy the custom environment on them. At design time we set the number of instances we want for the multiple instance composite task representing the mesh and solver processes in figure 4.19. Of course this number must be less or equal to the number of different files produced by the previous tasks (this is called N, the *database size*). The list of deployed services is kept in a file that is accessible by the YAWL engine. This will use only the default service for the registered atomic tasks and the list of all services for the *distributed_simulation* task from figure 4.19. The mechanism used to allocate services to instances of this task is described in section 4.2.4.

During our experiments we varied the database size (N) from 2 to 64 and we run the workflow on various configurations varying from 2 to 64 different machines. We encountered various technical obstacles of which the most important one is related to the number of different threads used by the engine. When the values of the database size were large, the number of threads produced by the YAWL engine was bigger than the maximum set in the operating system. By increasing this value we could finally execute normally a large scale application.

The purpose of this large scale application was to measure the speed-up execution time that we get when we increase the number K of custom services gradually while the database size is fixed to a value of $N = 64$. When the number of services is less than 64 then each service will be used $k = \frac{64}{K}$ times resulting a longer execution time. The shortest execution time is of course obtained for $K = 64$ services running in parallel. if we note t_0 the average time (this is an average

time since the G5K clusters are heterogeneous, in particular the number of cores per processor as well as the type of processor may slightly vary) it takes to one custom service to execute one instance of the task, then the total execution time is $T = \frac{t_0 N}{K}$ and the speed-up $S = \frac{t_0 N}{T} = K$.

We took execution time measurements for different number of services varying from 1 to 64. We chose to perform around 20 iterations for one *distributed_simulation* instance which gave a $t_0 = 10min$. In consequence, the execution of an application with 64 instances for the *distributed_simulation* task associated to only one computing node lasted around 11 hours. On the opposite, when we were using 64 nodes with 4 cores each, we obtained an execution time of around 10 minutes. We excluded from this measurements the custom environment deployment time which varied considerably with the number of nodes. If sending the operating system image on every node was done in parallel using Grid5000 tools, the installation and activation of some additional software tools was performed iteratively in a loop which generated the overhead. The average deployment time for one node was around 5 minutes and could go up to 30 minutes when deploying on 64 nodes.

We can see in figure 4.20 how the speed-up increases when the number of services responsible for executing the multiple composite task *distributed_simulation* is increased. The variations is somehow linear depending however of the heterogeneity of Grid5000 clusters and to a certain point of the relation between the number of instances to execute and the available number of services in charge of execution. Complementary, figure 4.21 illustrates the descending curve of the execution time measured in seconds relatively to the increasing number of computing nodes (i.e. services).

This test case shows us that executing a large scale numerical simulation application in a distributed manner can produce a significant speed-up but the user must wisely adapt the number of nodes working in parallel so that he takes full advantage of their computing power without wasting too much of it.

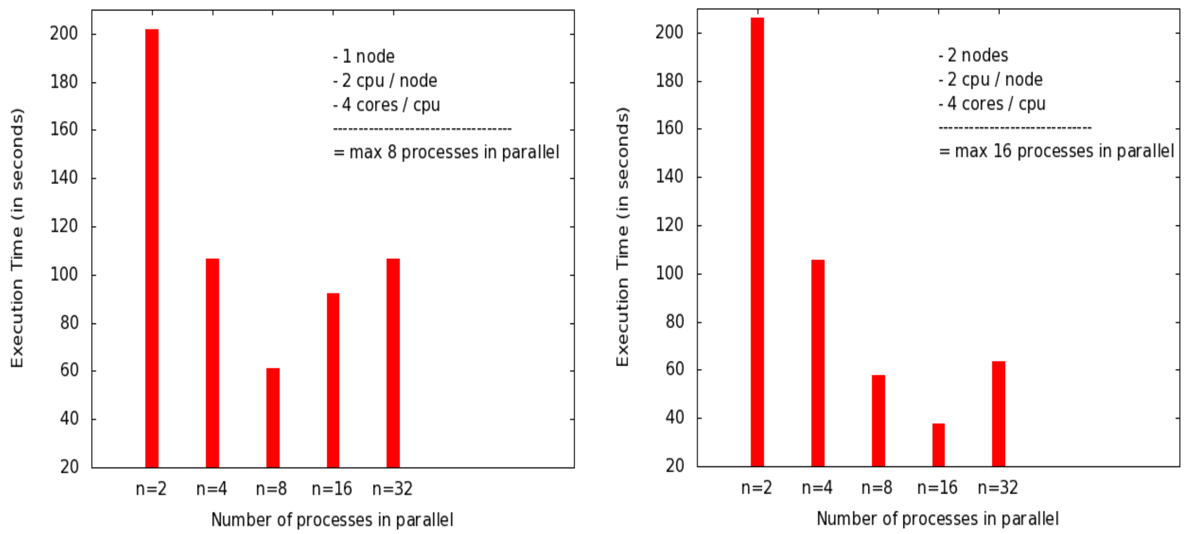


Figure 4.18: Execution time vs resources for the execution of 16 processes on 2 different clusters

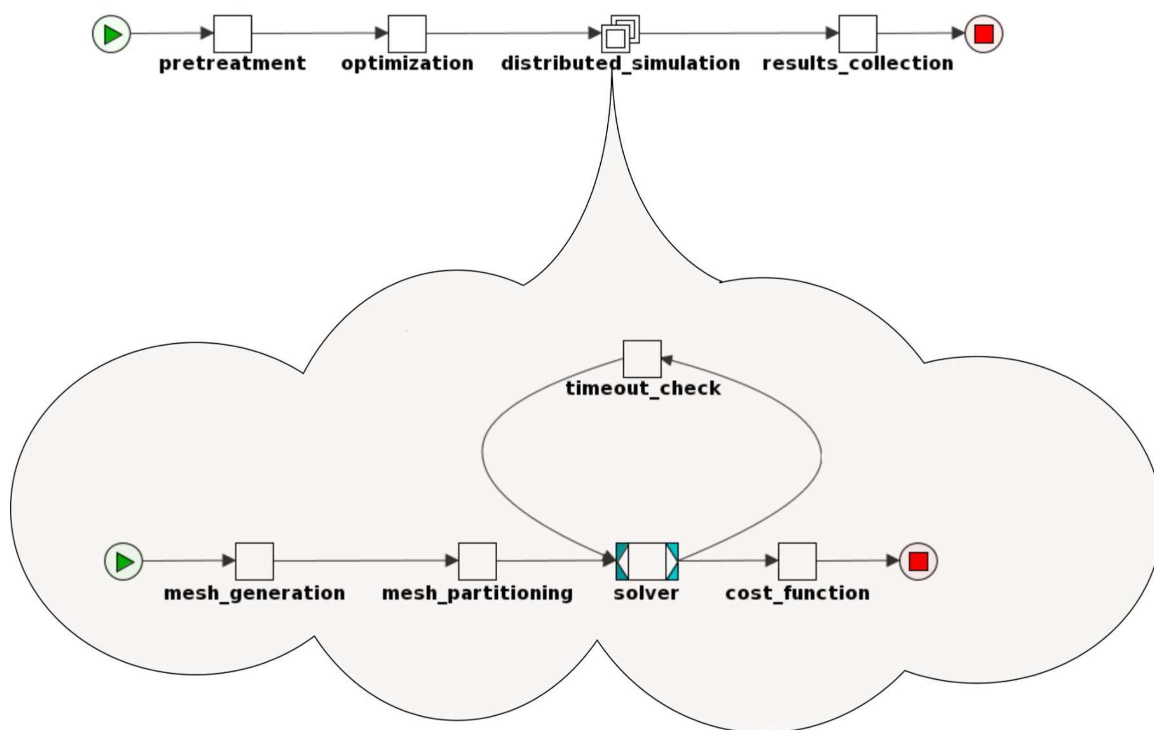


Figure 4.19: Large Scale Optimization Process With Exception Handling

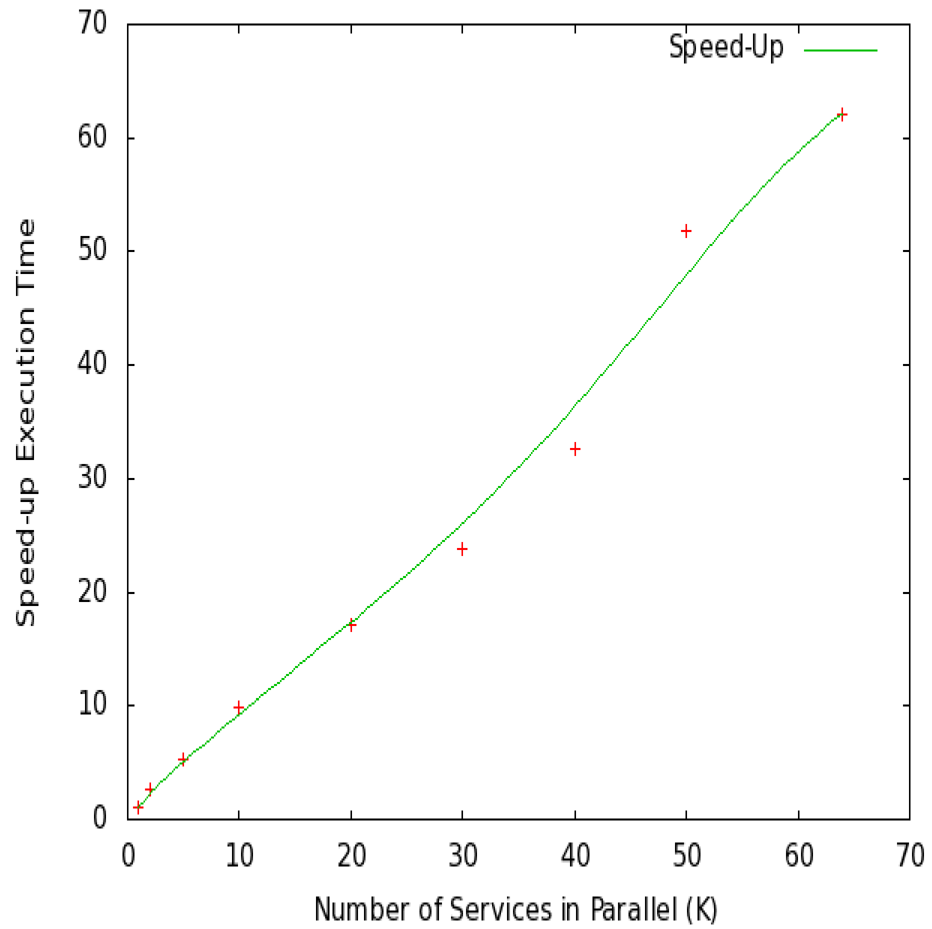


Figure 4.20: Speed-Up Execution Time

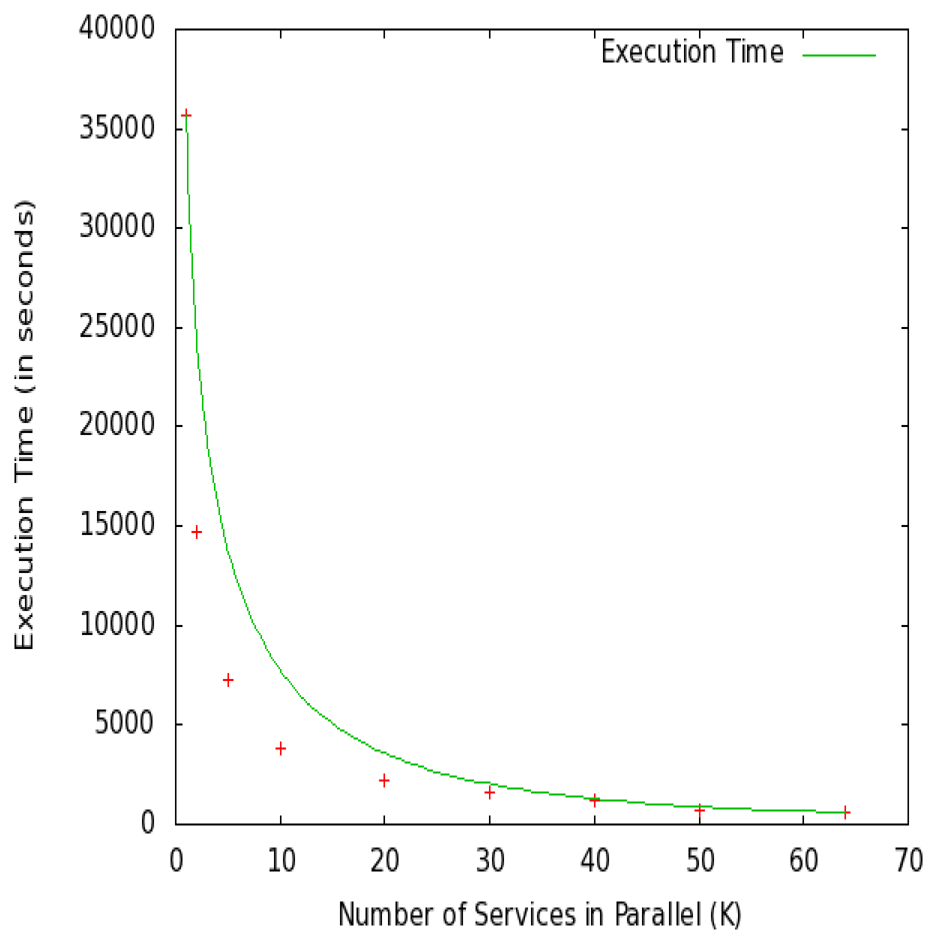


Figure 4.21: Large Scale Execution Time

5. Conclusions and Perspectives

The study performed during this thesis was founded on two major research directions: construction of a flexible multidisciplinary workflow execution platform for computing intensive numerical optimization applications and resiliency at application level. The chapters composing this manuscript approached these research themes gradually from general concepts and past research activity in the field, to a specific implementation proposed by this thesis.

In chapter 2 we have first presented an overview of existing distributed computing systems based on their hardware and middleware. We presented the evolution of this kind of systems in time from a hardware point of view and also the different types we can find according to their purpose and structure. Then we analyzed more in depth the Grid5000 grid system by emphasizing its architecture and different tools used for resource management. Going up one level, we analyzed these systems judging by their middleware layer. We detailed the ones we have tested for our platform, like OAR part of Grid5000. Among all these options we have chosen to use Grid5000 since it's a research dedicated computing infrastructure with a great flexibility concerning resource management. Another reason was the quality of the technical support and most important its proximity.

We continued this chapter with workflow systems, a major component in the construction of our execution platform. We described the architectures on which modern workflow systems are based and how the concept migrated from management applications to the HPC field. Then we explained the features a workflow system requires, to be considered dynamic and fault tolerant and why the current fault tolerant techniques cannot apply efficiently to a workflow system. In comparison to other scientific workflow systems like Askalon or Kepler, we showed that YAWL has the right properties and a good potential to answer to our needs of dynamicity and fault tolerance. Another reason for which YAWL proved to be a good choice is its flexibility to add extensions through YAWL Custom Services.

Chapter 3 was designed to explain the concepts directly related to the thesis objectives. We first presented the structure of a numerical optimization application with its optimization loop, showing the different stages from generating the mesh to computation of cost functions. With this structure in mind we detailed the main test-case (2D and 3D format of the air conditioning pipe) proposed by the OMD2 project. This was the cornerstone for all the workflow variations we proposed in the thesis to support our experiments. The necessary tools to execute the test-case were provided by our colleagues in Sofia Antipolis. We presented the logical execution chain and the modifications we proposed. The test-case section is completed with the description of a large scale application.

Afterwards we discussed the type of exceptions we can encounter during the execution of a numerical simulation application. Starting from a general classification of faults that are common to distributed environments, no matter the type of application, we emphasized the exceptions that are specific to the numerical field. The low level exceptions like CPU errors or out of memory situations are common to all type of applications since they are related to the hardware platform and only influence the application execution in an indirect manner. On the other side we have the application exceptions which are characteristic to each type of applications. In the case of numerical simulation applications the application exceptions are consequences of bad functioning of different execution stages like meshing, solver or the simulation algorithm in general. We concluded this presentation of exceptions with how we translate them in YAWL language so that they are easily intercepted and treated by the execution platform.

At the end of the chapter we showed the logical steps we propose to treat exceptions. We gave a theoretical view of how we want to deal with concepts like data management, data synchronization and also exception detection and recovery. We showed that, for YAWL, a lot of application exceptions translate into parameter exceptions and timeout exceptions. Finally we tackled a sensitive topic in exception treatment, that of fault origin. We have seen that our choice of manually pointing the origin point of fault is the desired one giving the time constraints imposed to this thesis.

Chapter 4 is where we proposed our implementation version of the concepts presented in the previous chapter. We described how we connected YAWL with Grid5000 and all the service extensions we have added to YAWL so that the resulting platform can meet the needs for executing tasks on Grid5000. We gave detailed explanations of the algorithms implementing data transferring and storage but also the technical modifications we introduced between YAWL and Grid5000 so that the Grid5000 resources are accessible to YAWL engine as well. We completed the execution platform implementation with the modifications we applied to YAWL engine so that it can handle a large scale distributed application.

In the second part of this chapter we presented a series of scenarios that we conceived to illustrate the distributed computation and resilience capabilities of our platform. We began with some basic scenarios like a sequential execution on multiple clusters or a distributed execution on two different clusters to highlight the basic characteristics. Then we showed how a timeout exception scenario is manageable for the platform. Composing and extending the previous scenarios we finished the tests with two complex scenarios. The first one showed that our platform is capable of adapting during run-time to the users needs and computing infrastructure constraints when the main goal is the execution results. The second complex scenario proved that the execution platform can also support large scale executions by managing tens of YAWL Custom Services located on as many different computing nodes with no boundaries regarding the hosting infrastructure site location.

Even if at the end of this thesis we have a functional multidisciplinary execution platform for scientific numerical applications along with a prototype of a fault tolerant mechanism, there is still space for many improvements so that what we propose can be widely accepted and used by non computer scientists.

In the short term perspective we think that YAWL should include in its standards an extension for dealing with real distributed workflows. Even though it contains the concept of *multiple instance composite task*, we have seen that this was mostly designed to run several instances of a task on a single service, not on a lot of services. When a workflow application scales to a considerable number

of instances that should be allocated to custom services, the current procedure of allocating tasks to services proved to be very cumbersome and time consuming. Also the YAWL engine is not prepared to handle the significant amount of threads that are generated when a large scale application executes. A version of the algorithm presented in section 4.2.4 ought to be added to YAWL standard so that the platform can have better chances of being adopted by scientists.

In the current version, our platform separates the computing platform deployment from the actual application execution. We consider that the deployment step should be translated into a YAWL Custom Service that gives the user the possibility to specify the computing constraints for his application. This could mean specifying the number of nodes, their type, the duration and secondary alternatives in case the main reservation fails. To really support the user in this matter a graphical user interface can be envisioned. Moreover, the current GUI used for execution monitoring in YAWL should be modified so that the user can have a better visibility regarding which task is currently executing, what are the intermediary results or on which node/nodes on the platform is executed.

The way we use Grid5000 as a computing infrastructure can be categorized as *Hardware as a Service*(HaaS) i.e. we reserve and address each not individually. Despite a better access to the level of the computing nodes, it remains a difficult way of managing resources especially for a non computer scientist. A medium term perspective could be to modify the platform so as to access an *Infrastructure as a service* (IaaS) model (e.g. a cloud service).

To increase the potential of our platform we could extend and adapt it for applications from other scientific fields than numerical simulation. One example are the DNA *Next Generation Sequencing*(NGS) methods (95). These represent innovations in DNA sequencing that allow the standard process to be parallelized in order to lower the costs and increase the DNA sequence throughput. These methods produce a very large quantities of data that need to be processed and analyzed. With the evolution of distributed computing platforms and algorithms that are able to process these amounts of data, the NGS methods have continuously improved. A very popular programming model that adapts very well to

this kind of applications is the *Map Reduce* model (96). It is designed to process and generate large data sets and to run on large distributed computing infrastructures. This way the programmer can focus more on the algorithm than on the hardware infrastructure.

Developing the platform around application exceptions we neglected the low-level exceptions leaving them in the responsibility of the underlying computing infrastructures. Despite the fact that most modern infrastructures are managed by middleware layers that take responsibility for a lot of hardware errors, we can never be sure how those errors are treated or if their impact is just hidden to the user. Sometimes these errors can affect the normal execution of an application influencing the results or just stopping the execution completely. It would be useful that when these errors are not actually treated by the computing platform to raise them at application level and to develop treatment procedures the same way we did for application exceptions. This would increase the robustness of our execution platform, no matter the type of exceptions it must face.

Finally we would like to present another important topic that we did not explore so deeply: the dynamicity of our workflow system. We believe that an important improvement for our platform would be to render the workflow applications fully reconfigurable at run-time. This implies that if an exception occurs or some parameters values have changed after the execution started, the workflow is able to change its structure dynamically and seamless to the execution process. A starting point to achieve this is given by the *Ripple Down Rules* that we already used. We have only used this technique for the exception treatment procedure but it can be used as well for choosing an execution direction, at run-time, according to the current context of the application.

Exploring better the dynamicity of our platform we can support another important perspective for this research activity, the designation of the root cause of an exception. At the moment the point in the workflow where the application will restart the execution after an exception treatment procedure is indicated statically at design time by the workflow architect. This can work very fine for a

few well defined exceptions for which the root cause can be known in advance. But most of the exceptions produce visible effects only after affecting a significant part of intermediary results (see (4)). By re-executing the applications from states corresponding to those visible effects we risk to reproduce the same bad behavior increasing the costs of the execution. With an automatic root cause designation algorithm we can simplify the workflow design and be more efficient in workflow exception recovery procedures.

Designating automatically the root cause is somehow related to having a sort of learning mechanism that accompanies the execution thread and learns the structure of the workflow at run-time. This way the best re-execution point, based on the root cause, is chosen in case an exception occurs. Some good starting articles in this direction can be found in the following list: (4, 10, 89).In the general case, the existence of such an algorithm looks very unlikely. However it may be possible to design one within a particular application context, like numerical optimization.

References

[1] R. Gupta, P. Beckman, B.-H. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine, and J. Dongarra. **CIFTS: A Coordinated Infrastructure for Fault-Tolerant Systems**. In *Parallel Processing, 2009. ICPP '09. International Conference on*, pages 237 –245, september 2009. iii, 28, 46

[2] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, C. Xuebin, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, I. Yutaka, J. Fred, K. Sanjay, K. Richard, K. David, K. Bill, L. Jesus, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. Van Der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick. **The International Exascale Software Project roadmap**. *International Journal of High Performance Computing Applications*, **25**(1):3–60, February 2011. 1, 2, 45, 46

[3] K. Plankensteiner, R. Prodan, T. Fahringer, A. Kertesz, and P. Kacsuk. **Fault-tolerant behavior in state-of-the-art Grid Workflow Management Systems**. Technical report, Institute for Computer Science, University of Innsbruck, 18 October 2007. 3

[4] W. Zang, M. Yu, and P. Liu. **A Distributed Algorithm for Workflow Recovery**. *International Journal of Intelligent Control and Systems*, March 2007. 3, 79, 118

[5] **YAWL: Yet Another Workflow Language**. `http://www.yawlfoundation.org/`, 14 October 2009. 3

[6] **YAWL - User Manual**. `http://www.yawlfoundation.org/manuals/YAWLUserManual2.3.pdf`, 2012. 3, 42, 50, 51, 53, 55

[7] **Workflow Patterns Home Page**. `http://www.workflowpatterns.com/`, 2010-2011. 3, 50

[8] W. M. P van der Aalst, M. Adams, A. H. M. ter Hofstede, M. Pesic, and H. Schonenberg. **Flexibility as a Service**. Technical report, Business Process Management Center, 2008. 3

[9] M. Adams, A. H. M. ter Hofstede, W. M. P. van der Aalst, and D. Edmond. **Dynamic, Extensible and Context-Aware Exception Handling for Workows**. In *Proceedings of the 15th International Conference on Cooperative Information Systems*, November 2007. 3

[10] P. Compton, G. Edwards, B. Kang, L. Lazarus, R. Malor, T. Menzies, P. Preston, A. Srinivasan, and S. Sammut. **Ripple Down Rules: Possibilities and Limitations**. In *AAAI Knowledge Acquisition for Knowledge*, 1991. 4, 118

[11] N. Antonopoulos, G. Exarchakos, M. Li, and A. Liotta. *Handbook of Research on P2P and Grid Systems for Service-Oriented Computing: Models, Methodologies and Applications*. IGI Global, 2010. 10

[12] D. C. Marinescu, G. M. Marinescu, Y. Ji, L. Boloni, and H. J. Siegel. **Ad Hoc Grids: Communication and Computing in a Power Constrained Environment**. In *Performance, Computing, and Communications Conference, 2003. Conference Proceedings of the 2003 IEEE International*, pages 113 – 122, april 2003. 11

[13] L. W. McKnight, J. Howison, and S. Bradner. **Guest Editors' Introduction: Wireless Grids–Distributed Resource Sharing by Mobile, Nomadic, and Fixed Devices**. *Internet Computing, IEEE*, **8**(4):24 – 31, july-august 2004. 11

[14] W. Li, Z. (Tommy) Xu, B. Li, and Y. Gong. **The Vega Personal Grid: A Lightweight Grid Architecture**. 11

[15] R. Desmarais and H. Muller. **A Proposal for an Autonomic Grid Management System**. In *Software Engineering for Adaptive and Self-Managing Systems*, page 11, may 2007. 11

[16] C. Hewitt. **ORGs for Scalable, Robust, Privacy-Friendly Client Cloud Computing**. *Internet Computing, IEEE*, **12**(5):96 –99, sept.-oct. 2008. 12

[17] K. Krauter, R. Buyya, and M. Maheswaran. **A Taxonomy and Survey of Grid Resource Management Systems**. *Software Practice and Experience*, **32**:135–164, 2002. 12

[18] B. Barney. **Message Passing Interface**, October 2012. 13

[19] F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, and O. Richard. **Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform**. In *Grid'2005 Workshop*. IEEE/ACM, November 13-14 2005. 14

[20] Grid5000 website. **Grid5000 Network Interlink**, January 2013. 17

[21] M. C. Cera, Y. Georgiou, O. Richard, N. Maillard, and P. O. A. Navaux. **Supporting MPI Malleable Applications upon the OAR Resource Manager**. 2009. 16

[22] E. Jeanvoine, L. Sarzyniec, and L. Nussbaum. **Kadeploy3: Efficient and Scalable Operating System Provisioning for HPC Clusters**. Rapport de recherche RR-8002, INRIA, June 2012. 16

[23] The Diet Team. **Diet User Manual**, July 2012. 19, 22

[24] I. Foster. **Globus Toolkit Version 4: Software for Service-Oriented Systems**. In *IFIP International Conference on Network and Parallel Computing*, number 3779 in LNCS, pages 2–13. Springer-Verlag, 2005. 20, 21

[25] A. Amar, R. Bolze, A. Bouteiller, A. Chis, Y. Caniou, E. Caron, P. K. Chouhan, G. Le Mahec, H. Dail, B. Depardon, F. Desprez, J.-S. Gay, and A. Su. **DIET: New Developments and Recent Results**. Technical Report RR2006-31, Laboratoire de l'Informatique du Parallélisme (LIP), october 2006. 20

[26] OAR Team. **OAR Documentation**, June 2012. 23

[27] OAR Team. **OAR Wiki Documentation**, September 2011. 23

[28] **ProActive Scheduling Manual**. http://proactive.inria.fr/release-doc/Scheduling/pdf/ProActiveSchedulerManual.pdf, January 2011. 23

[29] **ProActive Resource Manager Manual**. http://proactive.inria.fr/release-doc/Resourcing/pdf/ProActiveResourceManagerManual.pdf, January 2011. 24

[30] R. Lauwereins. **The Importance of Fault-Tolerance for Massively Parallel Supercomputers**. June 1992. 26

[31] M. Treaster. **A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems**. *ACM Computing Research Repository (CoRR)*, **501002**:1–11, 2005. 26

[32] B. Schroeder and G. A. Gibson. **Understanding failures in petascale computers**. *Journal of Physics: Conference Series*, **78**(1), 2007. 27

[33] X. Besseron. *Tolérance aux fautes et reconfiguration dynamique pour les applications distribuées à grande échelle*. These, Institut National Polytechnique de Grenoble - INPG, April 2010. 27

[34] A. S. Tanenbaum and S. van Maarten. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. 27, 31, 32, 33

[35] E. Heien, D. Kondo, A. Gainaru, D. LaPine, B. Kramer, and F. Cappello. **Modeling and tolerating heterogeneous failures in large parallel systems**. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 45:1–45:11, 2011. 27

[36] L. Zhiling and L. Yawei. **Adaptive Fault Management of Parallel Applications for High-Performance Computing**. *IEEE Transactions on Computers*, **57**(12):1647–1660, 2008. 28

[37] L. Yawei and L. Zhiling. **Exploit failure prediction for adaptive fault-tolerance in cluster computing**. In *Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006. CCGRID 06.*, **1**, pages 8 pp. –538, may 2006. 28

[38] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. **Toward Exascale Resilience**. *Int. J. High Perform. Comput. Appl.*, **23**(4):374–388, November 2009. 28

[39] C. Zizhong and J. Dongarra. **Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources**. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, april 2006. 28, 46

[40] F. Cappello. **Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities**. *International Journal of High Performance Computing Applications*, **23**(3):212–226, 2009. http://www.odysci.com/article/1010112993228110. 29, 30, 33

[41] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello. **Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications**. In *Parallel Distributed Processing Symposium (IPDPS)*, pages 989 –1000, may 2011. 31

[42] A. Oliner, L. Rudolph, and R. Sahoo. **Cooperative checkpointing theory**. In *Parallel and Distributed Processing Symposium*, page 10, april 2006. 32

[43] N. Naksinehaboon, Y. Liuand, C. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott. **Reliability-Aware Approach: An Incremental Checkpoint/Restart Model in HPC Environments**. In *Proc. 8th Intl. Symp. on Cluster Computing and the Grid (CCGRID)*, 2008. 32

[44] X. Besseron, S. Jafar, T. Gautier, and J.-L. Roch. **CCK: An Improved Coordinated Checkpoint/Rollback Protocol for Dataflow Applications in Kaapi**. pages 3353 –3358, 2006. 32

[45] A. Bouteiller, P. Lemarinier, K. Krawezik, and F. Capello. **Coordinated checkpoint versus message log for fault tolerant MPI**. In *2003 IEEE International Conference on Cluster Computing, 2003. Proceedings.*, pages 242 – 250, december 2003. 32, 45

[46] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. **FTI: high performance fault tolerance interface for hybrid systems**. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 32:1–32:32. ACM, 2011. 33

[47] J. Bent, B. Mcclelland, G. Gibson, P. Nowoczynski, G. Grider, J. Nunez, M. Polte, and M. Wingate. **Plfs: A checkpoint filesystem for parallel applications**. Technical report, 2009. 33

[48] J.S. Plank, K. Li, and M.A. Puening. **Diskless checkpointing**. *IEEE Transactions on Parallel and Distributed Systems*, **9**(10):972 –986, October 1998. 33

[49] B. Nicolae and F. Cappello. **BlobCR: efficient checkpoint-restart for HPC applications on IaaS clouds using virtual disk image snapshots**. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 34:1–34:12, 2011. 34

[50] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. Pocock, A. Wipat, and P. Li. **Taverna: a tool for the composition and enactment of bioinformatics workflows.** *Bioinformatics*, **20**(17):3045–3054, November 2004. 34

[51] P. Fisher, H. Noyes, S. Kemp, R. Stevens, and A. Brass. **A Systematic Strategy for the Discovery of Candidate Genes Responsible for Phenotypic Variation**. In Keith DiPetrillo, editor, *Cardiovascular Genomics*, **573** of *Methods in Molecular Biology*, pages 329–345. Humana Press, 2009. 34

[52] M. Caeiro-Rodriguez, T. Priol, and Z. Nemeth. **Dynamicity in Scientific Workflows**. Technical report, Institute on Grid Information, Resource and Workflow Monitoring Services, 31 August 2008. 34, 36, 38, 40, 43

[53] M. Ghanem, N. Azam, M. Boniface, and J. Ferris. **Grid-Enabled Workflows for Industrial Product Design**. In *Second IEEE International Conference on e-Science and Grid Computing, 2006. e-Science '06.*, page 96, december 2006. 34

[54] M. Vasko and S. Dustdar. **A View Based Analysis of Workflow Modeling Languages**. In *Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 293–300, Washington, DC, USA, 2006. IEEE Computer Society. 34

[55] E. Deelman and Y. Gil. **ManagingLargeScaleScientificWorkflowsInDistributedEnvironmentsExperiencesAndChallenges**. In *e-Science and Grid Computing, 2006. e-Science '06. Second IEEE International Conference on*, page 144, december 2006. 34

[56] E. Deelman, S. Callaghan, E. Field, H. Francoeur, R. Graves, V. Gupta, T. H. Jordan, C. Kesselman, P. Maechling, G. Mehta, D. Okaya, K. Vahi, and L. Zhao. **Managing Large-Scale Workflow Execution from Resource Provisioning to Provenance Tracking: The CyberShake Example**. In *In Proceedings of the Second IEEE international Conference on E-Science and Grid Computing*, pages 4–6, 2006. 34

[57] W. Jianwu, I. Altintas, C. Berkley, L. Gilbert, and M. B. Jones. **A High-Level Distributed Execution Framework for Scientific Workflows**. In *IEEE Fourth International Conference on eScience, 2008. eScience '08.*, december 2008. 34

[58] J. Montagnat, D. Lingrand, and X. Pennec. **Flexible and efficient workflow deployement of data-intensive applications on grids with MOTEUR**. In *GRIDS with MOTEUR, in quot;International Journal of High Performance Computing Applicationsquot;, To appear in the special issue on Workflow Systems in Grid Environments*, page 3, 2007. 34

[59] The Diet Team. **Workflow management in Diet**. In *Diet User Manual*. July 2012. 34

[60] **Workflow Management Coalition - Terminology Glossary**. `http://www.wfmc.org/index.php?option=com_docman&task=doc_download&gid=93&Itemid=72`, February 1999. 35

[61] Z. Zhao, A. Belloum, H. Yakali, P. Sloot, and B. Hertzberger. **Dynamic Workflow in a Grid Enabled Problem Solving Environment**. In *CIT '05 Proceedings of the The Fifth International Conference on Computer and Information Technology*, 2005. 36

[62] W. Van Der Aalst, K. Van Hee, Prof. Dr. Kees, M. Hee, R. De Vries, J. Rigter, E. Verbeek, and M. Voorhoeve. **Workflow Management: Models, Methods, and Systems**, 2002. 36, 37

[63] **Finite State Automata**. `http://www.eti.pg.gda.pl/katedry/kiw/pracownicy/Jan.Daciuk/personal/thesis/node12.html`. 37, 38

[64] **Grafcet**. `http://fr.wikipedia.org/wiki/Grafcet`. 38

[65] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. **Workflow Patterns**. *Distributed and Parallel Databases*, **14**:5–51, 2003. 10.1023/A:1022883727209. 40, 50

[66] S. Shankar and D. J. DeWitt. **Data driven workflow planning in cluster management systems**. In *Proceedings of the 16th international symposium on High performance distributed computing*, HPDC '07, pages 127–136, 2007. 40, 44

[67] M. Shields. **Control- Versus Data-Driven Workflows**. In Ian J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 167–173. Springer London, 2007. 41

[68] **Workflow Patterns Official Website**. `http://www.workflowpatterns.com/`. 41, 50

[69] M. Adams, Ter A. H. M. Hofstede, D. Edmond, and W. M. P. van der Aalst. **Implementing Dynamic Flexibility in Workflows using Worklets**. Technical report, BPM Center Report BPM-06-06, BPM-center.org, 2006. 43, 53

[70] D. Abramson, C. Enticott, and I. Altinas. **Nimrod/K: Towards massively parallel dynamic Grid workflows**. In *International Conference for High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008.*, pages 1 –11, november 2008. 44

[71] K. Plankensteiner, R. Prodan, and T. Fahringer. **A New Fault Tolerance Heuristic for Scientific Workflows in Highly Distributed Environments Based on Resubmission Impact**. In *Fifth IEEE International Conference on e-Science, 2009. e-Science '09*, december 2009. 44

[72] J. Yu and R. Buyya. **A Taxonomy of Workflow Management Systems for Grid Computing**. Technical report, JOURNAL OF GRID COMPUTING, 2005. 44, 49

[73] G. Kandaswamy, A. Mandal, and D. A. Reed. **Fault Tolerance and Recovery of Scientific Workflows on Computational Grids**. In *8th IEEE International Symposium on Cluster Computing and the Grid, 2008. CCGRID '08.*, may 2008. 45

[74] X. Besseron and T. Gautier. **Optimised Recovery with a Coordinated Checkpoint/Rollback Protocol for Domain Decomposition Applications**. In *Modelling, Computation and Optimization in Information Systems and Management Sciences*, **14** of *Communications in Computer and Information Science*, pages 497–506. 2008. 45

[75] L. ZongWei. **Checkpointing for workflow recovery**. In *Proceedings of the 38th annual on Southeast regional conference*, ACM-SE 38, pages 79–80, 2000. 45

[76] C. Buligon, S. Cechin, and I. Jansch-Pôrto. **Implementing rollback-recovery coordinated checkpoints**. In *Proceedings of the 5th international conference on Advanced Distributed Systems*, ISSADS'05, pages 246–257, 2005. 45

[77] L. Ramakrishnan, C. Koelbel, Y.-S. Kee, R. Wolski, D. Nurmi, D. Gannon, G. Obertelli, A. Yarkhan, A. Mandal, T. M. Huang, K. Thyagaraja, and D. Zagorodnov. **VGrADS: Enabling e-Science Workflows on Grids and Clouds with Fault Tolerance**. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 47:1–47:12, New York, NY, USA, 2009. ACM. 45

[78] W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. **A checkpoint-on-failure protocol for algorithm-based recovery in standard MPI**. In *Proceedings of the 18th international conference on Parallel Processing*, Euro-Par'12, pages 477–488, 2012. 45, 46

[79] T. Fahringer, R. Prodan, D. Rubing, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek. **ASKALON: a Grid application development and computing environment**. In *The 6th IEEE/ACM International Workshop on Grid Computing, 2005.*, pages 10 pp.–, 2005. 47, 48

[80] R. Duan, R. Prodan, and T. Fahringer. **DEE: A Distributed Fault Tolerant Workflow Enactment Engine for Grid Computing**. In *High Performance Computing and Communications*, **3726** of *Lecture Notes in Computer Science*, pages 704–716. Springer Berlin / Heidelberg, 2005. 47

[81] I. Altintas, A. Birnbaum, Kim K. Baldridge, W. Sudholt, M. Miller, C. Amoreira, Y. Potier, and B. Ludaescher. **A Framework for the Design and Reuse of Grid WorkFlows**. In *International Workshop on Scientific Aspects of Grid Computing*, pages 120–133. Springer-Verlag, 2005. 49

[82] PtolemyII website. **Ptolemy II Project and System**. 49

[83] D. Crawl and I. Altintas. **A Provenance-Based Fault Tolerance Mechanism for Scientific Workflows**. In *In Intl. Provenance and Annotation Workshop (IPAW)*, 2008. 49, 80

[84] W. M. P. van der Aalst and Ter A. H. M. Hofstede. **Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages**. In *of DAIMI, University of Aarhus*, pages 1–20, 2002. 50

[85] **YAWL - Technical Manual**. `http://www.yawlfoundation.org/manuals/YAWLTechnicalManual2.1.pdf`, 2010. 51, 52, 95, 96

[86] M. Adams, Ter A. H. M. Hofstede, W. M. P. van der Aalst, and N. Russell. *Modern Business Process Automation*. Springer, 2010. 53, 54, 77

[87] T. Nguyên, L. Trifan, and J.-A. Désidéri. **A Distributed Workflow Platform for Simulation**. In *Proceedings of the 4th International Conference on Advanced Engineering Computing and Applications in Sciences*, ADVCOMP 2010, Florence, Italy, 2010. 57

[88] E. Sindrilaru, A. Costan, and V. Cristea. **Fault Tolerance and Recovery in Grid Workflow Management Systems**. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*, pages 475 –480, february 2010. 61

[89] M. Yu, P. Liu, and W. Zang. **The implementation and evaluation of a recovery system for workflows**. *Journal of Network and Computer Applications*, **32**(1):158–183, January 2009. 61, 80, 118

[90] **OMD2 project website**. `http://omd2.scilab.org/`. 66

[91] R. Le Riche, D. Caromel, and R. Duvigneau. **Optimization tools and applications developed during the OMD & OMD2 projects**. In *Forum Teratech 2011, Complex systems engineering workshop (atelier ingénierie des systèmes complexes)*, Palaiseau, France, June 2011. 67

[92] A. Zerbinati, J.-A. Désidéri, and R. Duvigneau. **Application of Metamodel-Assisted Multiple-Gradient Descent Algorithm (MGDA) to Air-Cooling Duct Shape Optimization**. In *ECCOMAS - European Congress on Computational Methods in Applied Sciences and Engineering - 2012*, Vienna, Autriche, Sep 2012. 67

[93] **Opale Team website**. `https://team.inria.fr/opale/software/`. 72

[94] J. Eder and W. Liebhart. **Workflow Recovery**. In *in IFCIS Conference on Cooperative Information Systems*, pages 124–134. Society Press, 1996. 79

[95] M. P. Kumar, K. Nayong, L. Andre, K. Joohyun, and J. Shantenu. **Understanding mapreduce-based next-generation sequencing alignment on distributed cyberinfrastructure**. In *Proceedings of the 3rd international workshop on Emerging computational methods for the life sciences*, ECMLS '12, pages 3–12. ACM, 2012. 116

[96] D. Jeffrey and G. Sanjay. **MapReduce: simplified data processing on large clusters**. *Commun. ACM*, **51**(1):107–113, January 2008. 117

Declaration

I herewith declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This paper has not previously been presented in identical or similar form to any other French or foreign examination board.

The thesis work was conducted from 02/12/2009 to 30/04/2013 under the supervision of Nguyen TOAN at INRIA.

Grenoble,