To my parents, for their love

Preface

In several scientific domains large data repositories are generated. To find interesting and useful information in those repositories, efficient data mining techniques must be used. Many scientific fields, such as astronomy, biology, medicine, chemistry and earth science, get advantages from data mining analysis. The exploitation of data mining techniques in science help scientists in hypothesis formation and give them a support on their scientific practices, taking advantage from the knowledge that can be extracted from large data sources. Data mining tasks are often distributed since they involve data and tools located over geographically distributed environments, like the Grid. Therefore, it is fundamental to exploit effective paradigms, such as services and workflows, to model data mining tasks that are both multi-staged and distributed.

This thesis presents the results of our research aimed at providing data mining services and workflow systems for analyzing scientific data in a high performance distributed environment such as the Grid. Two alternative approaches have been investigated: one was to extend an existing data mining open source toolkit to make it suitable to support distributed data analysis workflows in grid settings, the second one was to work on a service oriented framework, the Knowledge Grid, to provide a high level visual interface able to support the wide range of distributed data mining scenarios.

The result of our first study is the implementation of Weka4WS, an extension of the open source Weka data mining suite, allowing users to exploit the computing power of distributed Grid nodes; in particular we have extended the Knowledge Flow environment of Weka to allow the execution of its data mining workflows on a set of distributed nodes. In this way, data mining experts can focus on the composition of their applications using a familiar tool, without having to learn a new environment or to learn complex tools and languages for the use and management of a Grid.

Our work on the Knowledge Grid framework focused in three main aspects: (i) designing and implementing Web Services that constitute the core of its jobs execution functionalities, (ii) defining a workflow formalism and (iii) pro-

viding a visual environment, called DIS3GNO, for composing and executing data mining tasks on the Grid.

Experiments have been performed to assess the efficiency of both workflow systems as well as to test effective design of real data mining applications. The results of the tests performed have proven their good scalability and that the overhead introduced by the remote invocations of the algorithms does not affect significantly the performance of the systems.

Finally we focused on fault tolerance issues in data mining workflows. The result of this study is the definition of a fault taxonomy for scientific workflows that may help in conducting a systematic analysis of faults, so that the potential faults that may arise at execution time can be corrected. The proposed taxonomy has a particular focus on workflow environments and is demonstrated through its use in Weka4WS.

Prefazione

In diversi settori scientifici vengono generati repository di dati di grandi dimensioni. Per trovare informazioni utili e interessanti in quei repository, devono essere utilizzate efficienti tecniche di data mining. Molti campi scientifici, come l'astronomia, la biologia, la medicina, la chimica e le scienze della terra, traggono vantaggio dall'analisi di data mining. Utilizzare tecniche di data mining in campo scientifico aiuta gli scienziati nella formazione di ipotesi e d loro un supporto nella loro pratica scientifica, potendo trarre vantaggio dalla conoscenza che pu essere estratta da grandi sorgenti di dati. I task di data mining sono spesso distribuiti in quanto richiedono dati e strumenti situati in ambienti geograficamente distribuiti, come la Grid. Pertanto è fondamentale sfruttare paradigmi efficaci, come i servizi e i workflow, per modellare task di data mining che siano sia a pi fasi che distribuiti.

Questa tesi presenta i risultati della nostra ricerca volti a fornire servizi di data mining e sistemi di workflow per l'analisi dei dati scientifici in un ambiente distribuito ad alte prestazioni come il Grid. Sono stati esaminati due approcci alternativi: uno è stato l'estensione di un toolkit di data mining open source esistente per renderlo adatto a supportare workflow di data mining distribuito in ambienti Grid, il secondo è stato la realizzazione di un framework orientato ai servizi, la Knowledge Grid, per fornire un'interfaccia visuale di alto livello in grado di supportare una vasta gamma di scenari di data mining distribuito.

Il risultato del nostro primo studio è l'implementazione di Weka4WS, un'estensione della suite di data mining open source Weka per permettere agli utenti di sfruttare la potenza di calcolo di nodi distribuiti di Grid; in particolare abbiamo esteso l'ambiente Knowledge Flow di Weka per consentire l'esecuzione dei workflow di data mining su un insieme di nodi distribuiti. In questo modo gli esperti di data mining possono concentrarsi sulla composizione delle loro applicazioni utilizzando uno strumento a loro familiare, senza dover imparare un nuovo ambiente o dover imparare strumenti complessi e linguaggi per l'uso e la gestione di una Grid.

Il nostro lavoro sul framework della Knowledge Grid si è focalizzato su tre aspetti principali: (i) progettazione e implementazione di Web Service che costituiscono il nucleo delle sue funzionalit di esecuzione dei job, (ii) definizione di un formalismo di workflow e (iii) progettazione e sviluppo di un ambiente visuale, chiamato DIS3GNO, per la composizione e l'esecuzione di task di data mining su Grid.

Sono stati condotti degli esperimenti per valutare l'efficienza di entrambi i sistemi di workflow, nonché per testare la progettazione efficace di applicazioni di data mining reali. I risultati dei test effettuati hanno dimostrato la loro buona scalabilit e che l'overhead introdotto dalle invocazioni remote degli algoritmi non influisce in modo significativo sulle prestazioni dei sistemi.

Infine ci siamo focalizzati sugli aspetti di tolleranza ai guasti nei workflow di data mining. Il risultato di questo studio ha portato alla definizione di una tassonomia per workflow scientifici che pu aiutare nello svolgimento dell'analisi sistematica dei guasti, in modo che gli eventuali guasti che possono verificarsi in fase di esecuzione possano essere corretti. La tassonomia proposta è focalizzata in particolare sugli ambienti di workflow ed dimostrata attraverso il suo uso in Weka4WS.

Arcavacata di Rende, Cosenza, Italy *Marco Lackovic*

November 2010 Marco Lackovic

Contents

1 Introduction . . . 1
1.1 Problem context . . . 1
1.2 Motivation . . . 2
1.3 Objectives of the Research . . . 5
1.3.1 Weka4WS . . . 5
1.3.2 Knowledge Grid . . . 5
1.3.3 Fault Tolerance . . . 6
1.4 Publications . . . 6
1.4.1 Book Chapters . . . 6
1.4.2 Papers in refereed conference proceedings . . . 7
1.5 Organization of the Thesis . . . 7

2 Background . . . 9
2.1 Data Mining . . . 9
2.1.1 Examples of applications . . . 10
2.1.2 The data mining process model . . . 10
2.1.3 Data mining techniques . . . 12
2.1.4 Distributed Data Mining . . . 14
2.2 Grid computing . . . 16
2.2.1 Grid History . . . 18
2.2.2 Grid Architecture . . . 19
2.2.3 OGSA . . . 21
2.2.4 Web Services . . . 21
2.2.5 WSRF . . . 23
2.2.6 Globus Toolkit . . . 24
2.3 Workflows . . . 26
2.3.1 Workflow Levels . . . 28
2.3.2 Workflow Models . . . 28
2.4 Fault Tolerance . . . 29
2.4.1 Failure Models . . . 30
2.4.2 Failure Masking . . . 31

2.4.3 Failure Detection . . . 31
2.4.4 Failure Recovery . . . 32

3 Related Work . . . 33
3.1 Askalon . . . 33
3.2 DVega . . . 34
3.3 GridAnt . . . 35
3.4 Gridbus . . . 36
3.5 Grid-Flow . . . 36
3.6 GWES . . . 37
3.7 GRMS . . . 38
3.8 ICENI . . . 39
3.9 Java CoG Kit-Karajan . . . 39
3.10 Kepler . . . 40
3.11 Pegasus . . . 41
3.12 Taverna . . . 42
3.13 Triana . . . 44
3.14 ScyFlow . . . 45
3.15 UNICORE Rich Client . . . 45
3.16 Grid-based Data Mining . . . 46

4 Weka4WS . . . 49
4.1 System Goals . . . 49
4.2 System Architecture . . . 50
4.2.1 User node . . . 52
4.2.2 Computing node . . . 53
4.3 Graphical User Interface . . . 55
4.3.1 Explorer . . . 57
4.3.2 Knowledge Flow . . . 59
4.4 How the system works . . . 62
4.5 Supporting data-parallel workflows . . . 65
4.6 Conclusions . . . 68

5 Knowledge Grid . . . 69
5.1 The DIS3GNO System . . . 71
5.1.1 Workflow Representation . . . 72
5.1.2 Workflow Composition . . . 75
5.1.3 Execution Management . . . 79
5.2 Conclusions . . . 81

6 Workflow level fault tolerance . . . 83
6.1 Methodology . . . 84
6.2 Weka4WS extended architecture . . . 84
6.3 Faults Taxonomy . . . 86
6.4 Fault identification . . . 87

6.5 Corrective Actions 89
6.6 Fault detection & monitoring 90
6.7 Conclusions 92

7 Validation 93
7.1 Weka4WS use cases and performance 93
7.1.1 Classification workflow 93
7.1.2 Clustering workflow 95
7.1.3 Execution on a multi-core machine 97
7.2 Knowledge Grid use cases and performance 98
7.2.1 Parameter sweeping workflow 99
7.2.2 Ensemble learning workflow 101
7.3 Analysis of potential faults 103
7.4 Conclusions 106

8 Conclusions and Future Work 107

References 113

1

Introduction

1.1 Problem context

The past two decades have been characterized by the emergence of increasingly powerful and less expensive ubiquitous computing, and by the large use of the World Wide Web and related technologies [21]. Due to such advances in information technology and high performance computing, digital data volumes are growing exponentially in many fields of human activities. Such technological development generated also new challenges: the world is drowning in a huge quantity of data, which is still growing very rapidly both in the volume and complexity.

This phenomenon is particularly interesting in scientific applications, that continually produce enormous volumes of data. For example, in the biological, medical, astronomic and earth science fields, large data sets are produced, whose analysis could add important contributions to the scientific knowledge. Unfortunately, huge data sets are hard to understand, and in particular models and patterns present in them cannot be comprehended by humans directly. Hence, since the volumes of scientific data are typically enormous, most of data will never be read by humans, but have to be processed and analyzed by computers. To this purpose, the exploitation of data mining techniques in science helps scientists in hypothesis formation and give them a support on their scientific practices and in problem solving environments, getting the benefits coming from knowledge that can be extracted from large data sources. A further important aspect to be considered is that researchers needing to access data belong to different communities and are often geographically distributed.

When data is large and is maintained over geographically distributed sites the computational power of distributed systems can be exploited for implementing knowledge discovery applications. Distributed data mining algorithms are suitable to such a purpose. Moreover, in this scenario distributed computing infrastructures, like Grids and Clouds, can provide an effective computational support for data intensive applications and for knowledge discovery from large and distributed datasets. Due to the complexity of dis-

tributed data mining applications in the scientific domain, it is fundamental to provide effective formalisms, such as workflows, to easily model such applications and to enable their efficient execution on a Grid.

As outlined earlier, data can be collected from many sources and stored at enormous speeds (GBs/hour). Both such aspects imply that scientific applications have to deal with a massive volume of data. Mining large data sets requires powerful computational resources. A major issue in data mining is scalability with respect to the very large size of current-generation and next-generation databases, given the excessively long processing time taken by (sequential) data mining algorithms on realistic volumes of data. In fact, data mining algorithms working on very large data sets take a very long time on conventional computers to get results. In order to improve performances, some distributed approaches have been proposed.

Distributed data mining (*DDM*) works by analyzing data in a distributed fashion and pays particular attention to the trade-off between centralized collection and distributed analysis of data. This approach is particularly suitable for applications that typically deal with very large amount of data (e.g., transaction data, scientific simulation and telecommunication data), which cannot be analyzed in a single site on traditional machines in acceptable times.

In particular, in the last decade, the *Grid computing* systems integrated both distributed and parallel computing, representing a privileged infrastructure for high-performance data and knowledge management. Grid computing was conceived as a paradigm for coordinated resource sharing and problem solving in advanced science and engineering applications. For these reasons, Grids offer an effective support to the implementation and use of knowledge discovery systems by exploiting *Grid-based data mining* approaches.

1.2 Motivation

The advances in information technology and high performance computing are leading to an exponential growth of digital data volumes in many scientific and human fields. This phenomenon has been becoming more and more relevant, especially in the last years. In particular, there are two important trends, technological and methodological, which seem to particularly distinguish the new, information-rich science from the past:

- *Technological issue*. A huge amount of data is collected and warehoused in various repositories distributed over the world: data can be collected and stored at high speeds in local or remote databases. Some examples include data sets from the fields of medical imaging, bio-informatics, remote sensing and several digital sky surveys. This implies a need for reliable data storage, networking, and database-related technologies, standards and protocols.

- *Methodological issue*. Huge data sets are hard to understand, and in particular models and patterns present in them cannot be comprehended by humans directly. This is a consequence of the growth in complexity of information, and mainly its multi-dimensionality. A computational simulation can generate Terabytes of data within a few hours, whereas human analysts may take several weeks to analyze them. For example, the NASA Earth Observing System[1] daily generates over 3 Terabytes of data that correspond to more than 1,000 Terabytes per year.

We can summarize what we foresaid as follows: whereas some decades ago the main problem was the lack of information, the challenge now seems to be (i) *the very large volume of information* and (ii) *the inherent complexity of data analysis processes*. However, the first aspect does not represent a limitation nor a problem for the scientific community: current data storage, architectural solutions and communication protocols provide a reliable technological base to collect and store such abundance of data in an efficient and effective way. Moreover, the availability of high throughput scientific instrumentation and very inexpensive digital technologies facilitated this trend from both technological and economical view point. On the other hand, the computational power of computers is not growing as fast as the demand of such a data computation requires, and this represents a limit for the knowledge that potentially could be extracted.

The impact of the above-mentioned issues in the biological field is well described in literature [64], where it is pointed out that the emergence of genome and post-genome technology has made huge amount of data available, demanding a proportional support for analysis. However, an important factor to be considered is that the number of available complete genomic sequences is doubling almost every 12 months, whereas according to Moore's law, available computing cycles (i.e., computational power) double every 18 months. That is, data to be processed are growing more quickly than computational and technological instruments. Additionally, we have to consider that analysis of genomic sequences requires binary comparisons of the genes involved in it. As a direct consequence of that, the computational overhead is very high. We can see the impact of such issues in Figure 1.1 [64] which contrasts the number of genetic sequences obtained with the number of annotations generated. The figure shows that the knowledge (annotations, models, patterns) has a sublinear rate compared with the available data sequences they are extracted from.

To handle this abundance in data availability, applications are emerging to explore, query, analyze, visualize, and in general, process very large-scale data sets: they are collectively named data intensive applications. Computational science is evolving toward data intensive applications that include data integration and analysis, information management, and knowledge discovery. In particular, knowledge discovery in large data repositories can be effectively

[1] http://eospso.gsfc.nasa.gov/

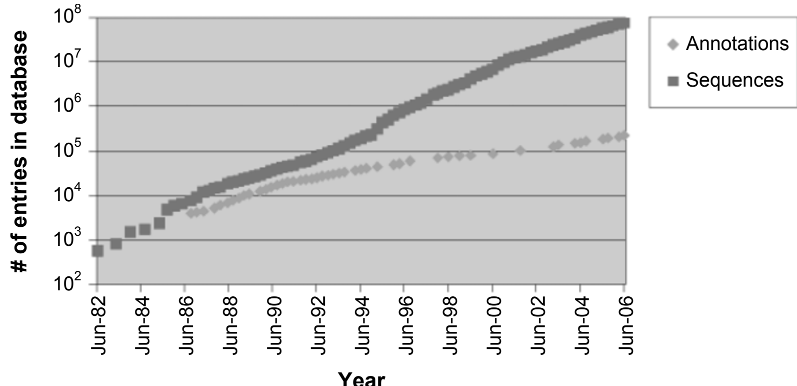

Fig. 1.1. Growth of sequences and annotations between 1982 and 2006 (source: [64]).

used to find what is interesting in them by exploiting data mining algorithms. Some examples on the use of data mining techniques in scientific domains are mentioned in the following [37, 49].

1. In *medical domain*, patient records collected for diagnosis and prognosis include symptoms, body measurements and laboratory test results. Such data are processed by data mining methods to improve decision-making. Examples of such applications are the induction of rules for early diagnosis of rheumatic diseases and neural nets to recognize the clustered micro-calcifications in digitized mammograms that can lead to cancer.
2. In *astronomic domain*, a photographic survey of the night sky (composed of thousands pictures) could contain around a billion faint objects. For the astronomers and astrophysics, the problem is to classify each object as a particular type of star or galaxy. Given the number of features to consider, as well as the huge number of objects, classification algorithms (i.e., decision-tree learning) have been found accurate and reliable for this task.
3. In *biological domain*, genetic data such as the nucleotide sequences in genomic DNA are digital. Unfortunately, experimental data are inherently noisy, making the search for patterns and the matching of subsequences difficult. Data mining algorithms such as neural network and hidden Markov models are a very suitable way to tackle this computationally demanding problem.

Medical, astronomic, and genomic data are just some examples of massive amounts of digital data that today must be stored and analyzed to find useful knowledge in them. As seen in the aforementioned examples, this data and information patrimony can be effectively exploited if it is used as a source to produce knowledge necessary to support decision making. In this context, an

important aspect to be considered is that data is usually collected in different sites. When scientific data is maintained over geographically distributed sites the computational power of distributed systems can be exploited for analyzing data where they are for speeding up knowledge discovery in them. In this case, distributed data mining algorithms are suitable to such a purpose.

The distribution of tasks to third-party, inter-organizational resources makes fault tolerance an important subject for scientific workflows. Some faults may be managed directly by the underlying resource management systems. However, this cannot always be guaranteed and depends on the type of resource manager being used. Consequently, the development of fault tolerance mechanisms at workflow (user) level become important. However, because scientific workflows have no control over external resources, the set of available actions to overcome faults directly, on external systems, is limited. This characteristic implies that, in case of big data mining experiments (big datasets involved requiring long-running analysis), the overall performance maybe hindered by occurring faults.

1.3 Objectives of the Research

1.3.1 Weka4WS

Grid computing can provide an effective data management and computational infrastructure support for distributed data mining. Rather than building a distributed data mining system from scratch, we took an already existing data mining software, which works on a single machine, and we extended it to support distributed data mining applications in a Grid environment. The system we chose for this purpose is Weka: being it a well established and widespread project, cross-platform (written in Java), and open source (available under the GNU General Public License), made it the most suitable candidate. This extended version of Weka has been called Weka4WS, which stands for *Weka for Web Services*, meaning that the data mining algorithms are executed remotely through Web Services. Weka4WS uses the Web Services Resource Framework (WSRF) technology for the implementation of the Grid services and Globus Toolkit for the remote execution of the data mining algorithms and for managing the remote computations. By extending a well established data mining environment, domain experts can exploit the computing power and storage capability of a Grid infrastructure but can focus on designing their data mining applications, without worrying about learning complex tools or languages for Grid submission and management. Weka4WS use cases and performance evaluation are presented at the end of this thesis.

1.3.2 Knowledge Grid

The Knowledge Grid is a software system that we developed for providing services to execute distributed data mining tasks in Grid environments. Work-

flows play a fundamental role in the Knowledge Grid at different levels of abstraction. We designed and developed the DIS3GNO component to provide a set of visual programming facilities to design and execute distributed data mining workflows on the Knowledge Grid. The DIS3GNO GUI operates as an intermediary between an end user and the Knowledge Grid allowing the management of the resources and an high-level development of service-oriented high-performance distributed KDD applications. All the Knowledge Grid services for metadata access and execution management are accessed transparently by DIS3GNO, thus allowing the domain experts to compose and run complex data mining applications without worrying about the underlying infrastructure details. Contrary to Weka4WS, DIS3GNO allows the composition of abstract workflows, that is workflows whose nodes may be not completely specified. In this way, a user can concentrate on the application logic, without focusing on the actual datasets or data mining tool to be used. The Knowledge Grid services will take care of finding the resources that fit user specifications. Knowledge Grid use cases and performance evaluation are presented at the end of this thesis.

1.3.3 Fault Tolerance

Inspired by a general fault taxonomy for Grids developed in [42], we have proposed a taxonomy for conducting a systematic analysis on faults within scientific workflows, focusing on fault tolerance capabilities that must be supported at the workflow level, as a user executing a workflow often has limited control of an external resource management system. The taxonomy we propose is general purpose, and can be applied across a variety of different workflow systems. We demonstrate through Weka4WS how the approach can be used for a realistic data mining workflow which includes Web Services (supporting WSRF and Globus).

1.4 Publications

The following publications have been produced while accomplishing this thesis.

1.4.1 Book Chapters

- M. Lackovic, D. Talia, P. Trunfio, "*A Framework for Composing Knowledge Discovery Workflows in Grids*". In: Foundations of Computational Intelligence Vol 6: Data Mining Theoretical Foundations and Applications, Studies in Computational Intelligence, A. Abraham, A. Hassanien, A. Carvalho, V. Snel (Editors), Springer, 2009;

- E. Cesario, M. Lackovic, D. Talia, P. Trunfio, "*A Visual Environment for Designing and Running Data Mining Workflows in the Knowledge Grid*". DATA MINING: Foundations and Intelligent Paradigms, Springer (to appear);
- E. Cesario, M. Lackovic, D. Talia, P. Trunfio, "*Service-Oriented Distributed Data Analysis in Grids and Clouds*". High Performance Scientific Computing with special emphasis on Current Capabilities and Future Perspectives (to appear).

1.4.2 Papers in refereed conference proceedings

- Lackovic M, Talia D, Trunfio P, "*Service-Oriented KDD: A Framework for Grid Data Mining Workflows*". Proc. of 10th International Workshop on High Performance Data Mining (HPDM 2008), in conjunction with ICDM 2008, Pisa, Italy, IEEE Computer Society Press, December 2008.
- Lackovic M, Talia D, Trunfio P, "*A Service-Oriented Framework for Executing Data Mining Workflows on Grids*". Proc. of the 4th International Workshop on Workflow Management (ICWM 2009), in conjunction with GPC 2009, Geneva, Switzerland, IEEE Computer Society Press, May 2009.
- Lackovic M, Talia D, Tolosana-Calasanz R, Banares J A, Rana O F, "*A Taxonomy For the Analysis of Scientific Workflow Faults*". Proc. of the 2nd International Workshop on Workflow Management in Service and Cloud Computing (WMSC 2010), in conjunction with CSE 2010, Hong Kong, China, December 2010;

1.5 Organization of the Thesis

The remainder of this thesis is organized as follows: Section 2 presents a short description of all the technologies involved in the thesis, in Section 3 related work is reviewed, Section 4 outlines the main features of the Weka4WS system, Section 5 describes the Knowledge Grid system with special emphasis on its DIS3GNO component and its workflow formalism, in Section 6 a multi-criteria taxonomy for scientific workflows is proposed and then fault detection and monitoring aspects are discussed, in Section 7 both Weka4WS and Knowledge Grid use cases and performance evaluation are presented and a case study that supports our taxonomy and demonstrate how the taxonomy can be used in practice. Finally, conclusions and future work are then discussed.

2

Background

This chapter presents a short description of all the technologies involved in the thesis.

2.1 Data Mining

In the last decades the ability to generate, acquiring and store data has increased rapidly due to the technology evolution and the increasing computerization of commercial, scientific and administrative sectors. It has been estimated the amount of data stored in the worldwide databases doubles every 20 months. This explosive growth has generated the urgent need of new technologies and automatic tools which could effectively aid in the transformation of this huge amount of data in useful information. Such information may provide scientists, engineers, and business people with a vast new resource that can be analyzed to make scientific discoveries, optimize industrial systems, and uncover financially valuable patterns.

Data Mining (DM) is the science for the extraction of implicit, previously unknown and potentially useful information, from a big amount of data or database. The idea is to automatically analyze databases looking for regularities or patterns. Naturally most of the results of this search will be trivial and not interesting, other will be false, depending by accidental coincidences in the particular database used. Furthermore the database itself may be flawed, some parts may be broken, some other may be missing. Hence the algorithms must be robust enough to treat with imperfect data to extract regularities which may be inaccurate but useful.

The data we are dealing with are in the form of set of examples and takes the name of dataset. Every single example of a dataset, called instance, is characterized from a set of values (nominal or numerical) each one corresponding to specific characteristics, or attributes, of the instance itself. Every dataset is represented by a matrix of instances by attributes, which in the databases jargon is called single relationship, or also flat file.

The result of the analysis generally include a concrete description of a structure which may be used to classify unknown examples. Such structure is typically expressed as set of rules, or decisional tree, and is itself so much important as the predictions which is able to perform, because data mining is frequently used to acquire knowledge and not only to make predictions.

Data mining is a multidisciplinary sector which includes the statistical analysis, machine learning, databases management, pattern recognition, artificial intelligence, neural networks, knowledge based systems (KBS), high performance computing and data visualization. The mastery of data mining in all its complexity is a very hard challenge because it requires the knowledge of these disciplines and in particular the full understanding of both statistical and computational problems.

2.1.1 Examples of applications

Examples of use of data mining are countless and may be found both in the commercial and in the scientific area. Companies may employ data mining to take profitable and immediate decisions and take advantage considerably over competitors, for example by analyzing the behaviour of the past customers with the aim of taking strategic decisions for the future. Lending institutions were the first to adopt data mining techniques, not only to improve the success rate of the loans but also to explain to the applicants the reasons behind the decisions. Data mining may come in help for the identification of false positives in the image screening, with which environmental scientists detect oil stains from the image analysis of the coast waters provided by radar satellites. In power load forecasting, a crucial problem in the electric power industry, data mining allows to obtain more rapidly accurate power demand estimates. In the preventive maintenance of electromechanical appliances such as engines and generators, data mining helps specialists in the diagnosis of the failure type in order to prevent the interruption of industrial processes.

2.1.2 The data mining process model

A data mining process model consists of a set of processing steps that can be followed by practitioners when they execute their data mining projects in order to help to plan, work through, and reduce the cost by detailing procedures to be performed in each of the steps. Data Mining is described to be a part of the Knowledge Discovery in Database (KDD) [24] process, as initially proposed by Osama Fayyad in 1996 [24]. The approach to gain knowledge out of a set of data was separated by Fayyad into five steps:

1. Selection: relevant information is selected, a target dataset is created, focusing on a subset of variables or data samples, on which discovery is to be performed

2. Pre-processing: unimportant elements of the provided data are detected and filtered out. The less noise is contained in data the higher will be the accuracy of the results of data mining. Elements of the pre-processing span from the cleaning of wrong data (data inconsistencies like out-of-range values like "Age = -10" or impossible data combinations like "Gender = male, Pregnant = yes"), the treatment of missing values and the creation of new attributes.;
3. Transformation: data is transformed into a format compatible to data mining algorithms using dimensionality reduction or transformation methods;
4. Data Mining: this stage consists on the searching for patterns of interest in a particular representational form, depending on the DM objective (usually, prediction);
5. Interpretation/Evaluation: this stage reveals whether or not the pattern is interesting, that is whether it contains knowledge at all. For that is important to represent the result in an appropriate way so it can be examined thoroughly. If the located pattern is not interesting, then it's important to study the cause for it and possibly fall back on a previous step for an attempt in another direction.

Other modified approaches to analyze the data followed the original proposal, for instance the former called **5A's** (Assess, Access, Analyze, Act and Automate) proposed by SPSS, or **SEMMA** (Sample, Explore, Modify, Model, Assess), proposed by SAS to be used in the Enterprise Miner. Both of these solutions though are closely connected to vendor products.

A standard that is industry-neutral, tool-neutral, and application-neutral is **CRISP-DM** [12] (Cross-Industry Standard Process for Data Mining): developed in 1996 by analysts representing DaimlerChrysler, SPSS, and NCR provides a nonproprietary and freely available standard process for fitting data mining into the general problem-solving strategy of a business or research unit. A poll of data mining practitioners, conducted by KDnuggets in 2002 [1], 2004 [2], and 2007 [3], revealed that CRISP-DM is the leading methodology used by data miners. It consists on a cycle that comprises six stages:

1. Business understanding: the project objectives and requirements are enunciated clearly in terms of the business or research unit as a whole. These goals and restrictions are translated into the formulation of a data mining problem definition and a preliminary strategy for achieving these objectives is prepared;
2. Data understanding: initial data is collected and exploratory data analysis is used to familiarize with the data and discover initial insights, the quality of the data is evaluated, interesting subsets are selected that may contain actionable patterns;

[1] http://www.kdnuggets.com/polls/2002/methodology.htm
[2] http://www.kdnuggets.com/polls/2004/data_mining_methodology.htm
[3] http://www.kdnuggets.com/polls/2007/data_mining_methodology.htm

3. Data preparation: the final data set that is to be used for all subsequent phases is prepared from the initial raw data, the cases and variables to be analyzed and that are appropriate are selected for the analysis, transformations are performed on certain variables, if needed, the raw data is cleaned so that it is ready for the modeling tools;
4. Modeling: appropriate modeling techniques are selected and applied, model settings are calibrated to optimize results, if necessary we loop back to the data preparation phase to bring the form of the data in line with the specific requirements of a particular data mining technique;
5. Evaluation: one or more models delivered from the modeling phase are evaluated for quality and effectiveness before deploying them for use in the field. It is determined whether the model in fact achieves the objectives set for it in the first phase and established whether some important facet of the business or research problem has not been accounted for sufficiently.
6. Deployment: the models created are used.

2.1.3 Data mining techniques

It is convenient to categorize data mining into types of tasks, corresponding to different objectives for the person who is analyzing the data. A possible categorization [40] is the following:

- Exploratory Data Analysis (EDA): as the name suggests, the goal here is simply to explore the data without any clear ideas of what we are looking for. Typically, EDA techniques are interactive and visual, and there are many effective graphical display methods for relatively small, low-dimensional datasets.
- Descriptive Modeling: the goal of a descriptive model is describe all of the data (or the process generating the data). Examples of such descriptions include models for the overall probability distribution of the data (density estimation), partitioning of the p-dimensional space into groups (cluster analysis and segmentation), and models describing the relationship between variables (dependency modeling). In segmentation analysis, for example, the aim is to group together similar records, where the number of groups here is chosen by the researcher; there is no “right” number. This contrasts with cluster analysis, in which the aim is to discover “natural” groups in data.
- Predictive Modeling (Classification and Regression): the aim here is to build a model that will permit the value of one variable to be predicted from the known values of other variables. In classification, the variable being predicted is categorical, while in regression the variable is quantitative. The key distinction between prediction and description is that prediction has as its objective a unique variable, while in descriptive problems no single variable is central to the model.

- Discovering Patterns and Rules: The three types of tasks listed above are concerned with model building. Other data mining applications are concerned with pattern detection. One example is spotting fraudulent behavior by detecting regions of the space defining the different types of transactions where the data points significantly different from the rest. Another use is in astronomy, where detection of unusual stars or galaxies may lead to the discovery of previously unknown phenomena. Yet another is the task of finding combinations of items that occur frequently in transaction databases (e.g., grocery products that are often purchased together). A significant challenge here is deciding what constitutes truly unusual behavior in the context of normal variability. Background domain knowledge and human interpretation can be invaluable.
- Retrieval by Content: here the user has a pattern of interest and wishes to find similar patterns in the data set. This task is most commonly used for text and image data sets. For text, the pattern may be a set of keywords, and the user may wish to find relevant documents within a large set of possibly relevant documents (e.g., Web pages). For images, the user may have a sample image, a sketch of an image, or a description of an image, and wish to find similar images from a large set of images. In both cases the definition of similarity is critical, but so are the details of the search strategy.

Basic data-mining techniques include clustering, association rule discovery, classification, sequential pattern discovery, and outlier detection. Other techniques exist, but we mention here only the main ones, those either more frequently used or more general.

Clustering is the process of partitioning or grouping a given set of data points into distinct groups, or clusters, such that the similarity between the data points in one cluster is maximized and the similarity between data points in different clusters is minimized. Clustering can be used for a wide range of applications from grouping companies with similar stock behavior or similar growth to identifying genes and proteins that have similar functions.

Given a number of transactions of item sets, **association rule discovery** finds all rules that correlate the presence of one set of items with that of another set of items. One familiar example is the discovery of items that sell together in a supermarket from mining the sales transactions at the point of sale. A management decision based on such findings could be to shelve these items close to one another.

Classification refers to assigning objects to predefined categories or classes. In a credit-evaluation scenario, classification can categorize applicants' credit ratings as good or poor, for acceptance or rejection.

Sequential pattern discovery determines strong sequential dependencies among different events. This process has many applications, from medical diagnosis to sales-transactions analysis to determine which customers are likely to buy a specific product in the near term.

Another application of data mining is the **detection of outliers** (or deviations). Outlier detection finds data points that differ significantly from the majority of the data points in a given data set. Medical diagnosis and credit card detection are examples of outlier detection.

2.1.4 Distributed Data Mining

Distributed data mining (DDM) is a result of further evolution of the data mining technology. The huge size of the available datasets and their high-dimensionality in many emerging applications can make knowledge discovery computationally very demanding, to an extent that distributed computing can become an essential component of the solution. DDM embraces the growing trend of merging computation with communication, accepting the fact that data may be inherently distributed among different loosely coupled sites connected by a network and the sites may have heterogeneous data. If the databases are large it will become nearly impractical to download them on one site to build the global data model. In some cases the transfer of sensible information may cause security problems or privacy violations. DDM promises to build a correct global model by performing partial analysis of data at individual sites and sends out the outcome to other sites. This saves communication over-head, offers better scalability, requires minimal communication of possibly secured data, and sometimes offers the only feasible way to mine distributed data sets.

There are two main paradigms for exploiting algorithm parallelism: Data and Task Parallelism. **Task parallelism** (also called simple parallelism [68]) is a form of parallelism which runs multiple independent tasks in parallel on different locations: it is often used to model embarrassingly parallel computations (for which little or no effort is required to separate the problem into a number of parallel tasks, which mostly happens when there exists no dependency between them) such as parameter-sweep simulations. In **data parallelism** instead a large dataset is split into smaller chunks, each chunk is processed in parallel, and the results of each processing are then combined to produce a single result. Data parallelism may be static or dynamic whether the number of partitions is known in advance, at design-time, or is determined automatically at run-time as a function of the number of the available resources. The data partitioning strategy may be homogeneous or heterogeneous: it's homogeneous when the partitions have the same size, heterogeneous when the data is split unevenly to ensure that all tasks run to completion in the same amount of time.

Both parallelism forms aim to achieve execution time speedup, and a better utilization of the computing resources, but while task parallelism focuses on running multiple tasks in parallel so that the execution time corresponds to the slowest task, data parallelism focuses on reducing the execution time of a single task by splitting it into subtasks, each one operating on a subset of the original data. The data-parallel approach is widely employed in distributed

data mining as it allows to process very large datasets that could not be analyzed on a single machine due to memory limitations, computing time, privacy or security constraints.

Most DDM algorithms are designed upon the potential parallelism they can apply over the given distributed data. Typically, the same algorithm operates on each distributed data site concurrently, producing one local model per site. Subsequently, all local models are aggregated to produce the final model. This schema is common to several distributed data mining algorithms. Among them, *meta-learning*, *collective data mining* and *ensemble learning* are the most important.

The *meta-learning* technique [71] aims at building a global classifier from a set of inherently distributed data sources. Meta-learning is basically a two-step process: first, a number of independent classifiers are generated by applying learning programs to a collection of distributed and homogeneous data sets in parallel. Then, the classifiers computed by local learning programs are collected in a single site and combined to obtain a global classifier.

Collective data mining [50] exploits a different strategy: instead of combining partial local models, it builds the global model through the identification of significant sets of local information. In other words, the local blocks are directly composed to form the global model. This result is based on the fact that any mining function can be expressed in a distributed fashion using a set of appropriate basis functions. If the basis functions are orthogonal, the local analysis generates results that can be correctly used as components of the global model.

Ensemble learning [79] aims at improving classification accuracy by aggregating predictions of multiple classifiers. An ensemble method constructs a set of base classifiers from training data and performs classification by voting (in the case of classification) or by averaging (in the case of regression) on the predictions made by each classifier. The final result is the ensemble classifier, that tends to have higher classification quality than any single classifier composing it.

In the DDM process, identifying optimal ways to combine the base classifiers is a crucial point. Prominent among these are schemes called bagging, boosting (perhaps the most powerful of the three methods), and stacking. They can all, more often than not, increase predictive performance over a single model. And they are general techniques that can be applied to numeric prediction problems and to classification tasks.

Bagging (called voting for classification, averaging for regression) combines the predicted classifications (prediction) from multiple models, or from the same type of model for different learning data. It is also used to address the inherent instability of results when applying complex models to relatively small data sets.

Boosting also combines the decisions of different models, like bagging, by amalgamating the various outputs into a single prediction, but it derives the individual models in different ways. In bagging, the models receive equal

weight, whereas in boosting, weighting is used to give more influence to the more successful ones.

Stacked generalization, or **stacking** for short, is a different way of combining multiple models, but it is less widely used than bagging and boosting. Unlike bagging and boosting, stacking is not normally used to combine models of the same type, instead it is applied to models built by different learning algorithms. Stacking introduces the concept of a metalearner, which replaces the voting procedure. The problem with voting is that it's not clear which classifier to trust. Stacking tries to learn which classifiers are the reliable ones, using another learning algorithm, the metalearner, to discover how best to combine the output of the base learners.

2.2 Grid computing

Grid computing in the vision of its creators intend to build, in analogy to the electricity distribution network (the power grid), a network of computing power and data storage distribution. In such way, a user who may need computing power (or any other computer service) could draw it from a connected infrastructure in a transparent manner, rather than from local resources. This infrastructure, referred to by the term *Grid*, is composed of an arbitrary number of heterogeneous computers, interconnected by a network.

Grid computing has received great attention both from the research community and from industry and governments, watching at this computing infrastructure as a key technology for solving complex problems and implementing distributed high-performance applications. Grid computing differs from conventional distributed computing because it focuses on large-scale dynamic resource sharing, offers innovative applications, and, in some cases, it is geared toward high-performance systems.

The Grid emerged as a privileged computing infrastructure to develop applications over geographically distributed sites, providing for protocols and services enabling the integrated and seamless use of remote computing power, storage, software, and data, managed and shared by different organizations. Basic Grid protocols and services are provided by toolkits such as *Globus Toolkit* [4], *Condor* [5], *gLite* [6], and *Unicore* [7]. In particular, *Globus Toolkit* is the most widely used middleware in scientific and data-intensive Grid applications, and represents a de-facto standard for implementing Grid systems. It addresses security, information discovery, resource and data management, communication, fault-detection, and portability issues.

A Grid is a system [28] which:

[4] http://www.globus.org/toolkit/
[5] http://www.cs.wisc.edu/condor/
[6] http://glite.web.cern.ch/glite/
[7] http://www.unicore.eu/

1. coordinates resources that are not subject to centralized control;
2. uses standard, open and multi-purpose protocols and interfaces;
3. provides non trivial quality of service.

A Grid node is a combination of both physical and logical resources. It is a “farm” of resources (computers, processors, services, applications) that is accessible through a unique address [74]. A minimum operating environment (middleware) is required for its computing power to be advertised and utilized. Each resource is managed under a single hosting environment accessible directly or indirectly through a unique reference. In a hosting environment resources are administrated under a single trust domain and they are advertised and utilized through a single access point called the front-end resource.

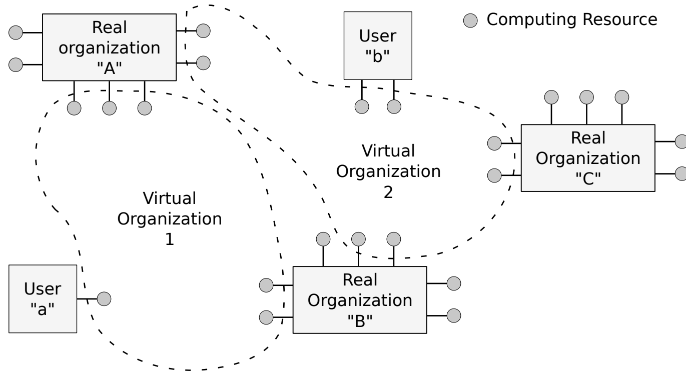

Fig. 2.1. Virtual Organizations

Grid computing allows resources of several distinct organizations, and even of single users, to be dynamically grouped into Virtual Organizations (VO) to solve specific problems (Figure 2.1). This task, far from trivial, requires the use of protocols, methodologies and sophisticated technologies to solve problems such as:

- the choice of the resources which will be part of each virtual organization;
- the choice of the resources to allocated for each specific task and for how long;
- the communication among the resources, taking into account the fact that they are heterogeneous, belonging to different organizations that may use different operating systems and even different computer architectures;
- the splitting of the task into sub-tasks so that they can be executed in parallel on multiple machines;
- the certainty that the shared resources will be used only by users with proper permissions.

In order for a computational problem to benefit from the Grid, it should require a high computational time and a large amount of data, and should be reducible to parallel processes that do not require a intensive intercommunication. Grid technologies provide mechanisms for the sharing and the coordination of various resources, thus enabling, starting from organized and geographically distributed components, the creation of virtual computing systems that are sufficiently integrated to provide the wanted quality of service. These technologies include:

- security solutions which support the management of credentials and policies when the computations span multiple institutions;
- resources management protocols and services that support secure remote access to computing resources and data, and the co-allocation of multiple resources;
- protocols for information requests and services that provide information on the status and configuration of resources, organizations and services;
- data management services that locate and transport datasets between applications and data base systems.

2.2.1 Grid History

The idea of Grid computing, which has been referred to as the next revolution in Information Technology (such as once was for the World Wide Web), dates back to the early 90s. We can identify four distinct phases in its evolution:

1. in the early 90's ad hoc solutions were developed for Grid computing problems: the goal of these systems was simply to run the applications and to explore the possibilities offered by this new computational model. The applications were built directly over the Internet protocols, typically with only limited functionality in terms of security, scalability, and robustness. Interoperability was not a major concern;
2. In 1997, with the release of version 2, Globus Toolkit (GT2) emerged as the de facto standard for Grid computing. Focusing on usability and interoperability, GT2 defined and implemented protocols, APIs, and services used in thousands of applications based on Grid worldwide. Providing solutions to common problems such as authentication, resource discovery, and resources access, GT2 has accelerated the construction of real Grid applications. Furthermore, by defining standard protocols and services, GT2 has pioneered the creation of interoperable Grid systems and introduced significant advances in programming tools for the Grid. Some elements of the set of protocols of GT2 were coded in specific formal techniques, in particular the GridFTP [7] data transfer protocol, and elements of the Grid Security Infrastructure (GSI).
3. The year 2002 saw the birth of the Open Grid Services Architecture (OGSA), a real standard with multiple implementations, including in particular the Globus Toolkit 3 one, released in 2003. In addition to defining

a central set of standard interfaces and showing ways on how to solve some problems, OGSA provides an environment within which it's possible to define a wide variety of portable and interoperable services.
4. The definition of the first OGSA specifications is an important headway, but much remains to be done before the complete vision of the Grid will be realized. Developing over the service-oriented infrastructure of OGSA there will be an expansion of a set of interoperable services, an increment of the degrees of virtualization, and an evolution towards richer forms of sharing. This work will be carried out more and more heavily on the results of advanced computer science research in areas such as peer-to-peer or knowledge-based systems.

Example of Grid Systems are: *EGEE* [8] (Enabling Grids for E-Science in Europe), a Grid project that will give scientists access to computational resource across 27 countries, also responsible for providing the computational power required by the Large Hadron Collider (LHC), the world's largest particle physics laboratory; *TeraGrid* [9], a grid system providing a powerful infrastructure for open scientific research, as of today offering 2 petaflops of computing power and 50 petabytes of distributed storage; *Access Grid* [10], a grid system used for large-scale distributed meetings, collaborative work sessions, seminars, lectures, tutorials and training; *eDiaMoND* [11], an example of how Grid computing can be used for e-Health, this project pools and distributes information on breast cancer treatment, enables early screening and diagnosis, and provides medical professionals with tools and information to treat disease.

2.2.2 Grid Architecture

Creating a complete grid system requires a variety of protocols, services and software development tools. The Grid architecture, organized in layers [51], identifies the fundamental system components, specifies the purpose and function of these components, and shows how these components interact with each other 2.2.

- Fabric: provides the resources to which shared access is mediated by Grid protocols: for example, computational resources, storage systems, catalogs, network resources, and sensors. A "resource" may be a logical entity, such as a distributed file system, computer cluster, or distributed computer pool.
- Connectivity: defines core communication and authentication protocols required for Grid-specific network transactions. Communication protocols

[8] http://public.eu-egee.org/
[9] http://www.teragrid.org/
[10] http://www.accessgrid.org/
[11] http://www.ediamond.ox.ac.uk/

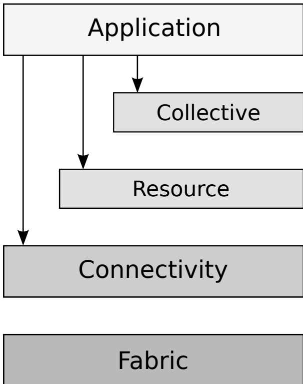

Fig. 2.2. Grid Architecture

enable the exchange of data between Fabric layer resources. Authentication protocols build on communication services provide cryptographically secure mechanisms for verifying the identity of users and resources. These protocols are drawn from the TCP/IP protocol stack: specifically, the Internet (IP and ICMP), transport (TCP, UDP), and application (DNS, OSPF, RSVP, etc.) layers of the Internet layered protocol architecture;

- Resource: built on Connectivity layer communication and authentication protocols, defines protocols (and APIs and SDKs) for the secure negotiation, initiation, monitoring, control, accounting, and payment of sharing operations on individual resources. Resource layer implementations of these protocols call Fabric layer functions to access and control local resources. Resource layer protocols are concerned entirely with individual resources and hence ignore issues of global state and atomic actions across distributed collections;
- Collective: contains protocols and services (and APIs and SDKs) that are not associated with any one specific resource but rather are global in nature and capture interactions across collections of resources. Collective components can implement a wide variety of sharing behaviors like: *directory services*, which allow VO participants to discover the existence and/or properties of VO resources; *co-allocation, scheduling and brokering services* which allow VO participants to request the allocation of one or more resources for a specific purpose and the scheduling of tasks on the appropriate resources; *monitoring and diagnostics services* to support the monitoring of VO resources for failure, adversarial attack ("intrusion detection“), overload, and so forth; *workload management systems and collaboration frameworks*, also known as *problem solving environments* (PSEs), to provide for the description, use, and management of multi-step, asynchronous, multi-component workflows; *software discovery services*, to dis-

cover and select the best software implementation and execution platform based on the parameters of the problem being solved;
- Applications: comprises the user applications that operate within a VO environment.

2.2.3 OGSA

The Open Grid Services Architecture (OGSA) [30], developed by the Global Grid Forum (GGF) is derived from a paper presented in [28] aims at defining a common architecture and open standards for Grid-based applications. Its goal is to standardize virtually all services that can be commonly found in a Grid system (jobs management systems, resource management services, security services, etc.) by specifying a set of standard interfaces for these services. Although still under development, OGSA already defines a set of requirements that these standard interfaces must meet, that is it has already identified the most important services that may be encountered in Grid systems and that can mostly benefit from standardization.

Among the different options (such as CORBA, RMI or RPC) over which OGSA could be based, the Web Services were chosen as the underlying technology. Web Services though are stateless, that is they do not maintain any data between an invocation and the next. Web Services alone thus do not meet one of the most important requirements of OGSA: its infrastructure must be stateful, that is capable of maintaining the status of a job from an invocation to another. To overcome this limitation, from a joint effort by the Grid community and by the Web Service community, the Web Services Resource Framework (WSRF) [18] was established, a family of specifications that, in addition to adding a number of interesting features, defines a set of operations that Web services can implement to become stateful. Ultimately WSRF is the infrastructure on which the OGSA architecture is built on.

2.2.4 Web Services

Web Services are a software system designed to support distributed computing (like CORBA, RMI, EJB, etc.): in short, they allow you to create client / server applications. In contrast to what happens with Internet websites, where information is intended for people, a Web Service is always aimed to a software, never directly to a person. Although Web services make extensive use of Web technologies (like HTTP) they have no relationship with Web browsers and the HTML language.

The advantages of the Web Services over technologically equivalent alternatives are:

- they are platform and language independent since they use standard XML languages;

- the use of HTTP for the transmission of messages makes them extremely attractive for building Internet-scale applications, since most of the existing proxies and firewalls are designed to handle this type of traffic;
- naturally lend themselves to building loosely coupled systems because they are message-oriented, and are based on language-neutral XML dialects to exchange messages and define interfaces. Such systems are much more scalable than tightly coupled systems and impose less architectural demand on the actual implementation of the Web Services.

Web Services have also some disadvantages:

- overhead: the transmission of data in XML format is obviously not as efficient as that made using a proprietary binary code. As in many similar cases, what is gained in portability is lost in efficiency. However, this overhead is usually acceptable for most applications, although it's unlikely to find a real time application which uses Web services;
- lack of maturity: Web Services are a relatively new technology and, although its central specifications that deal with key languages (like XML, WSDL, etc.) and protocols (like HTTP, SOAP, etc.) are fairly stable, the world of Web services is rapidly evolving.

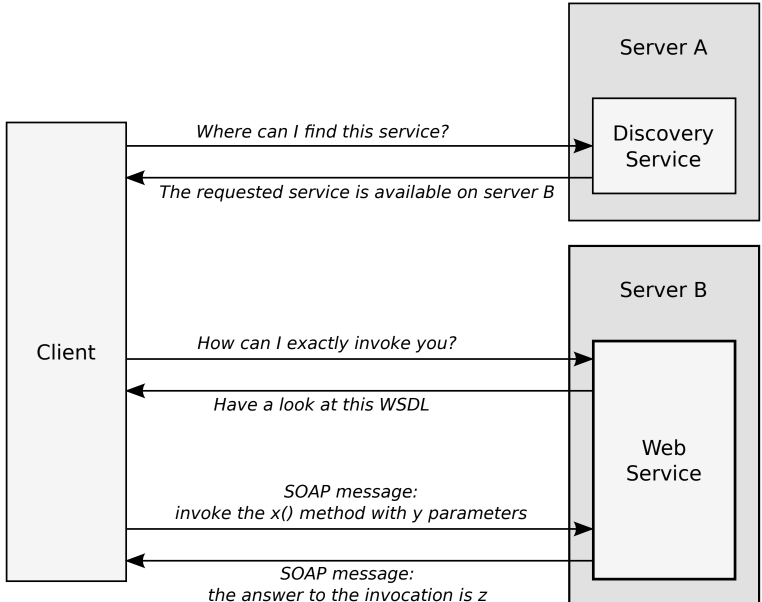

Fig. 2.3. A typical Web Service invocation

Figure 2.3 shows in details how a typical Web Service invocation takes place:

1. the Discovery Service is contacted (which is itself a Web Service) to find out where the desired service is located;
2. the Discovery Service answers with the address (or addresses) of the machine (or machines) that hosts the Web Service that offers the requested service. The address of a Web Service is shown in the form of Uniform Resource Identifier (URI) [82];
3. the Web Service in the previous step is contacted to ask it to describe itself in order to know exactly how it can be invoked;
4. the Web Service replies in a language called WSDL (short for *Web Services Description Language*, an XML-formatted language used to describe a Web service's capabilities as collections of communication endpoints capable of exchanging messages);
5. the service is invoked in a language called SOAP (originally defined as *Simple Object Access Protocol*, it is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks. It relies on XML for its message format, and usually relies on other Application Layer protocols, most notably RPC and HTTP, for message negotiation and transmission);
6. the Web Service responds with a SOAP message containing the response to the request, or possibly an error message if the SOAP request was incorrect.

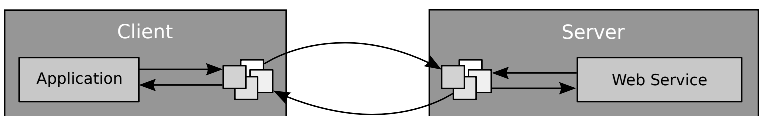

Fig. 2.4. Role of the stubs in the Web Service invocation

Although Web services revolve around many languages and protocols, programmers actually need to focus solely on programming in their language of choice and, in some cases, to write the WSDL. The SOAP code will be generated and interpreted automatically by a piece of software called *stub* (usually generated, automatically, from the WSDL description of the service). The use of stubs, shown in a simplified form in Figure 2.4, greatly simplifies applications developing because it allows to concentrate on writing client and server code, ignoring all the complex details of network communication.

2.2.5 WSRF

As mentioned in Section 2.2.4, Web Services have some limitations that make them inadequate for implementing Grid applications. WSRF improves some aspects of Web Services to make them suitable for Grid applications, in particular by introducing the ability to store its status information: instead of

putting the state in the Web service (thus making it stateful, which is generally regarded as a bad thing) it is kept in a separate entity called a *resource*, which stores all the state information. Each resource will have a unique key, so whenever a stateful interaction with a Web service is required the Web service will be simply instructed to use a particular resource. The coupling of a Web Service with a resource is called *WS-Resource*. There are many ways to access a WS-Resource, but the preferred one is to use a specification called WS-Addressing which provides a mechanism to address Web Services which is much more versatile than plain URIs. WS-Addressing defines a construct called *endpoint reference* (EPR) which may include, in addition to the URI of the Web Service, a resource identifier: in this case the EPR is called *WS-Resource-qualified endpoint reference*, although for simplicity is often called only EPR. Elements within a resource are called *resource properties* and provide a view of the current status of the resource (file name, size, descriptors, etc.).

WSRF is a collection of the following five specifications:

1. WS-Resource: defines a WS-Resource as a composition of a resource and a Web Service through which the resource can be accessed;
2. WS-ResourceProperties: provides a set of interfaces that allow access, modification and querying of the resource properties;
3. WS-ResourceLifetime: provides basic mechanisms to manage the lifecycle of the resource;
4. WS-ServiceGroup: specifies how exactly should be the grouping of the services or of the WS-Resources. It is the foundation for more advanced services (such as the IndexService of Globus Toolkit) allowing to group together different services and to identify them under a single entry point;
5. WS-BaseFaults: provides a standard way to represent errors when something goes wrong during the invocation of a Web Service.

Other specifications related to WSRF, but that does not belong to it, are:

- WS-Notification: allows a Web Service to be configured as a *notifications producer*, and the clients as *notifications consumers* (or subscribers). In this way each time a change takes place in the Web Service, or more specifically in one of its WS-Resources, it will be notified to all its consumers;
- WS-Addressing: as mentioned above, provides a mechanism to address Web services much more versatile than simple URI. In particular, it is possible to use WS-Addressing for addressing the couple Web Service and resource (or the WS-Resource, referred to both).

2.2.6 Globus Toolkit

Globus Toolkit [27] is an integrated set of open source software tools such as services, programming libraries and development tools designed for building

applications based on Grid. In this thesis we are going to consider the version 4.0.x of the framework, referred to as GT4. The components of GT4 are grouped into five broad categories:

1. Security: collectively referred to as *Grid Security Infrastructure* (GSI), these components facilitate secure communications and the application of uniform policies across distinct systems;
2. Data management: they ensure the discovery, transfer and access of large data. These include among others the GridFTP [7] and *Reliable File Transfer* (RFT);
3. Execution management: they deal with the deployment, scheduling and monitoring of executable programs, referred to as *jobs*. The *Grid and Resource Allocation and Management* (GRAM) [26] component, one of the main ones of GT4, belongs to this category;
4. Information Services: commonly referred to as *Monitoring and Discovery System* (MDS), includes a set of components to monitor and discover resources in a virtual organization. The aforementioned IndexService is part of this category;
5. Common Runtime: they provide a set of fundamental tools and libraries for hosting services as well as developing new ones. These include the *C Runtime*, the *Python Runtime* and the *Java Runtime*.

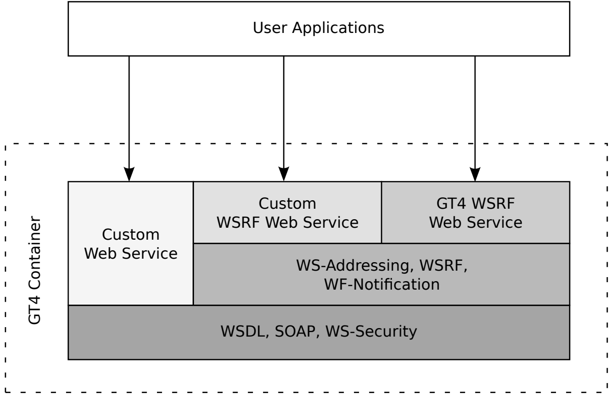

Fig. 2.5. The Java GT4 Web Services Container

Many of the components of GT4 are Web Services and need to run inside a *Web Services Container* (a catch-all term for the combination of a SOAP engine, an application server and possibly also an HTTP server). GT4 includes a simple Web Service Container, showed in Figure 2.5, based on Apache Axis (an open source framework for Web services, XML-based), but also offers the

option of deploying the Web Services into other application servers, such as the more advanced *Apache Jakarta Tomcat* [12].

2.3 Workflows

A workflow is a well-defined and possibly repeatable pattern or systematic organization of activities designed to achieve a certain transformation of data. Workflow, as practiced in scientific computing, derives from several significant precedent programming models that are worth noting because these have greatly influenced the way we think about workflow in scientific Grid applications. We can call these the "dataflow" model in which data is streamed from one actor to another. While the pure dataflow concept is extremely elegant, it is very hard to make work in practice because distributing control in a distributed system can create applications that are not very fault tolerant. Consequently, many workflow systems which use a dataflow model for expressing the computation may have an implicit centralized control program that sequences and schedules each action. An important benefit of workflows is that, once defined, they can be stored and retrieved for modifications and/or re-execution: this allows users to define typical patterns and reuse them in different scenarios.

In the Grid context the workflow can be defined as the automation of the processes, which involves the orchestration of a set of Grid services, agents and actors that must be combined together to solve a problem or to define a new service [32].

Data mining tasks and knowledge discovery in databases (KDD) processes are often composed of multiple stages (e.g., data extraction, data filtering, data analysis, results evaluation) that may be linked each other by different dependencies to form various execution flows. Workflows are well suited formalisms to represent data and execution flows associated to complex data mining tasks. A data mining workflow is a graph in which nodes typically represent data sources, filtering tools, data mining algorithms, and visualizers, and edges represent execution dependencies among nodes.

Of the many possible ways to distinguish workflow computations on Grids, one is to consider a simple complexity scale:

- At the most basic level one can consider *simple linear workflows* in which a sequence of tasks must be performed in a specified linear order. The first task transforms an initial data object into new data object which is the "input" to the next data-transformation task, etc. The execution time for the entire chain of tasks may be a few minutes, or it may be days. In the cases where the execution time is short, the most common workflow programming tool is a simple script written in Python or Perl

[12] http://jakarta.apache.org/tomcat/

or even Matlab. The case of longer running workflows often requires more sophisticated tools.

- At the next level of complexity, one can consider workflows that can be described by an *acyclic graph*, where nodes of the graph represent a task to be performed and edges represent dependencies between tasks. This is harder to represent with a scripting language without a substantial additional framework behind it, but it is not at all difficult to represent with a tool like Ant [13] and it is the foundation of the *DagMan*, a meta-scheduler of *Condor*, a specialized workload management system developed by the University of Wisconsin, and the execution engine of *Pegasus* (see Section 3.11). Applications that follow this pattern can be characterized by workflows in which some tasks depend upon the completion of several other tasks which may be executed concurrently.
- The next level of workflow complexity can be characterized *cyclic graphs*, where the cycles represent some form of implicit or explicit loop or iteration control mechanisms. In this case the workflow "graph" often describes a network where the nodes are either services or some form of software component instances or represent more abstract control objects. The graph edges represent messages or data streams or pipes that channel work or information between services and components.
- The final level of workflow is one in which a compact graph model is not appropriate. This may be the case when the graph is simply too large and complex to effectively "program" it as a graph. However, some tools allow one to turn a Graph into a new first-class component or service, which can then be included as a node in another graph (a workflow of workflows or *hierarchical workflow*). This technique allows graphs of arbitrary complexity to be constructed.

In the case of workflow enactment, there are two aspects to this: efficiency and robustness. In terms of efficiency, the critical issue is the ability to quickly bind workflow tasks to the appropriate Grid resources. It also depends very heavily on the mechanisms used to move data between tasks and services that need them at various stages of the enactment. One cannot assume that web service protocols like SOAP should be used in anything other than "control" and simple message delivery. Real data movement between components of the workflow must be either via an interaction with a data movement service, or through specialized binary-level data channel running directly between the tasks involved.

Robustness is another issue, making the reasonable assumption that some parts of a workflow may fail. It is essential that exception handling include mechanisms to recover from failure as well as detecting it. Also failure is something that can happen to a workflow enactment engine. A related issue is the monitoring of the workflow. In addition to being able to restart the workflow from a failure checkpoint, the user may wish to track progress of the

[13] http://ant.apache.org/

enactment. In some cases the workflow is event driven and a log of the events that trigger the workflow processing can be viewed to see how the workflow is progressing. This is also an important aspect of debugging a workflow. A user may wish to execute the workflow step by step to understand potential errors in the flow logic.

2.3.1 Workflow Levels

Workflow users utilize the interfaces (often graphical) that scientific workflows expose in order to build their workflow specifications. This corresponds to the design stage of a workflow. In general terms, two main workflow levels can be found on this regard, though there are other approaches that even differentiate more abstraction levels [9, 35]:

1. Abstract workflows. At this high level of abstraction the workflow contains just information about what have to be done at each task along with information about how tasks are interconnected. There is no notion of how input data is actually delivered or how tasks are implemented.
2. Concrete workflows. The mapping down to a concrete workflow annotates each of the tasks with information about the implementation and/or resources to be used. Information about method invocation and actual data exchange format are also defined.

In case the user is familiar with the technology and the resources available, they can even specify concrete workflows directly. Once a workflow specification is obtained, it is sent to the workflow engine for the execution stage. At this phase, workflow tasks are mapped onto third-party, distributed and heterogeneous resources and the scientific computations are accomplished.

2.3.2 Workflow Models

Although there is a standard workflow language like *Business Process Execution Language* (BPEL) [52, 59], scientific workflow systems often have developed their own workflow model for allowing users to express workflows. This characteristic makes difficult the sharing of workflow specification of resources and important requirement for scientific workflows. Nevertheless, there are some historical reasons for that, as many scientific workflow systems and their workflow models were developed before BPEL existed.

Existing workflow models can be grouped roughly into two classes [44]:

1. *Script-like* workflow descriptions specify workflows by means of a textual programming language that can be described by a grammar in an analogous way to programming languages. They often have complex semantics and an extensive syntax. In this type of descriptions, tasks are described and typically data dependencies can be established between them. These languages contain specific workflow constructs, such as sequence or loops,

while do, or parallel constructs in order to build up the workflow. Examples of script workflow descriptions are GridAnt [86] or Karajan [87]. A commonly used script-based approach to describe workflows, mainly in the business workflow community, is BPEL and its recent version for Web services that builds on IBM's Web Service Flow Language, WSFL.

2. *Graph-based* workflow models specify the workflow with only a few basic graph elements. Compared with script-based descriptions, graphs are easier to use and more intuitive for the unskilled user mainly because of their graphical representation: nodes typically represent workflow tasks whereas communications (or data dependencies) between different tasks are represented as arcs going from one node to another. Workflow systems which support graph-based models often incorporate graphical user interfaces which allow users to model workflows by dragging and dropping graph elements. Purely graph-based workflow descriptions generally utilize directly acyclic graphs (DAGs). Directed Acyclic Graph-based languages offer only a limited expressiveness, so that it is often hard to describe complex workflows (e.g. loops cannot be expressed directly).

2.4 Fault Tolerance

A characteristic feature of distributed systems that distinguishes them from single-machine systems is the notion of partial failure. A partial failure may happen when one component in a distributed system fails. This failure may affect the proper operation of other components, while at the same time leaving yet other components totally unaffected. An important goal in distributed systems design is to construct the system in such a way that it can automatically recover from partial failures without seriously affecting the overall performance. In particular, whenever a failure occurs, the distributed system should continue to operate in an acceptable way while repairs are being made, that is, it should tolerate faults and continue to operate to some extent even in their presence.

There is a considerable ambiguity in the literature on the meaning of some central terms like *fault* and *failure*. The term fault is usually used to name a defect at the lowest level of abstraction (e.g., a memory cell that always returns the value 0). A fault may cause an error, an error may lead to a failure, meaning that the system deviates from its correctness specification [25]. Faults are generally classified as transient, intermittent or permanent. *Transient* faults occur once and then disappear. If the operation is repeated, the fault goes away. A fault is called *intermittent* when occurs, then vanishes on its own accord, then reappears, and so on. Intermittent faults cause a great deal of aggravation because they are difficult to diagnose. A *permanent* fault is one that continues to exist until the faulty component is replaced.

2.4.1 Failure Models

Traditionally, faults are handled by describing the resulting behavior of the system and grouped into a hierarchic structure of fault classes or fault models [80]. One scheme by which failures may be classified is the following [14]:

- a *crash failure* occurs when a server prematurely halts, but was working correctly until it stopped. An important aspect in this case is that once the server has halted, nothing is heard from it anymore;
- an *omission failure* occurs when a server fails to respond to a request. In the case of a *receive* omission failure, possibly the server never got the request in the first place. This will not generally affect the current state of the server, as the server is unaware of any message sent to it. A *send* omission failure happens when the server has done its work, but somehow fails in sending a response. In contrast to a receive omission failure, the server may now be in a state reflecting that it has just completed a service for the client. As a consequence, if the sending of its response fails, the server has to be prepared for the client to reissue its previous request;
- *timing failures* occur when the response lies outside a specified real-time interval. Providing data too soon may cause trouble for a recipient if there is not enough buffer space to hold all the incoming data. More common, however, is that a server responds too late, in which case a *performance failure* is said to occur;
- a serious type of failure is a *response failure*, by which the server's response is simply incorrect. Two kinds of response failures may happen: in the case of a *value failure* a server simply provides the wrong reply to a request, in the case of a *state transition failure* the server reacts unexpectedly to an incoming request.
- the most serious are *arbitrary failures*, also known as *Byzantine failures* (inspired by the Byzantine Generals' Problem [69]). It may happen that a server is producing output it should never have produced, but which cannot be detected as being incorrect. A faulty server may even be maliciously produce intentionally wrong answers.
- *fail-stop failures* occur when a server simply stop producing output in such a way that its halting can be detected by other processes. In the best case the server may even announce it is about to crash.
- a *fail-silent failure* occur when it is up to the other processes to decide that a server has prematurely halted. However in this scenario the other processes may incorrectly conclude that a server has halted, while instead it may be just unexpectedly slow, that is, it is exhibiting performance failures.
- *fail-safe failures* occur when the server is producing random output, but in a way that it is recognized as incorrect by other processes. The server is the exhibiting an arbitrary failure, but in a benign way.

2.4.2 Failure Masking

A way to make a system fault tolerant is to try to hide the occurrence of failures from other processes. The key technique for masking faults is to use redundancy. Three kinds of redundancy are possible: information, time and physical. With *information redundancy* extra bits (for example a Hamming code) are added to allow recovery from garbled bits. With *time redundancy* an action is performed and then, if needed, it is performed again. Time redundancy is especially helpful when the faults are transient or intermittent. With *physical redundancy* extra equipment or processes are added to make it possible for the system as a whole to tolerate the loss or malfunctioning of some components.

An important issue with using process redundancy to tolerate faults is how much replication is needed. A system is said to be *k fault tolerant* if it can survive faults in k processes and still meet its specifications. If the processes fail silently, then having k+1 of them is enough to provide k fault tolerance. On the other hand, if processes exhibit Byzantine failures then a minimum of 2k+1 processes are needed to achieve k fault tolerance. In the worst case, the k failing processes could accidentally (or intentionally) generate the same reply. However, the remaining k+1 will also produce the same answer, so the client or voter will have just to believe the majority.

2.4.3 Failure Detection

Failure detection is one of the cornerstones of fault tolerance in distributed systems. Detecting process failures can be done essentially in two ways: either processes actively send "are you alive?" messages to each other (for which they obviously expect an answer), or passively wait until messages come in from different processes. In practice, actively pinging processes is usually followed and a timeout mechanism is used to check whether a process has failed. In real settings though, due to unreliable networks, simply stating that a process has failed because it hasn't returned an answer to a ping message may be wrong. In other words, it is quite easy to generate false positives.

Failure detection can take place also through *gossiping* in which each node regularly announces to its neighbors that it is still up and running. Eventually every process will know about every other process, but more importantly, will have enough information locally available to decide whether a process has failed or not: a member for which the availability information is old, will presumably have failed.

A failure detection system should ideally be able to distinguish network failures from node failures. One way of dealing with this problem is not to let a single node decide whether one of its neighbors has crashed: instead, when noticing a timeout on a ping message, a node requests other neighbors to seee whether they can reach the presumed failing node. Positive information can also be shared: if a node is still alive, that information can be forwarded to

other interested parties, who may be detecting a ling failure to the suspected node.

2.4.4 Failure Recovery

Once a failure has occurred, it is essential that the process where the failure happened can recover to a correct state. There are essentially two forms of error recovery:

1. in *backward recovery* the main issue is to bring the system from its present erroneous state back into a previously correct state. To do so, it will be necessary to record the system's state from time to time, and to restore such a recorded state when things go wrong. Each time (part of) the system's present state is recorded, a *checkpoint* is said to be made.
2. in *forward recovery* when the system has entered an erroneous state, instead of moving back to a previous, checkpointed state, an attempt is made to bring the system in a correct new state from which it can continue to execute. The main problem with forward error recovery is that it has to be known in advance which errors may occur.

The most widely used technique for recovering from failures in distributed systems is the backward recovery. Its main advantage is that it is generally applicable independently of any specific system or process; in other words, it can be integrated into (the middleware layer) of a distributed system as a general-purpose service. However, backward recovery also introduces some problems. First, restoring a system or process to a previous state is a generally costly operation in terms of performance. Second, no guarantees can be given that once recovery has taken place, the same or similar failure will not happen again. Finally, although it requires checkpointing, some states can simply never be rolled back to.

3

Related Work

In this chapter a review of the most important Scientific Workflow Management Systems is presented.

3.1 Askalon

ASKALON [23] is an Application Development and Runtime Environment for the Grid. Developed at the University of Innsbruck, Austria, it uses an XML-based language called *Abstract Grid Workflow Language* (AGWL) for describing Grid workflow applications at a high level of abstraction. It has a SOA-based runtime environment with stateful services and uses the Globus Toolkit as Grid platform; its architecture is shown in Figure 3.1. It supports a rich set of constructs for expressing sequence, parallelism, choice, and iteration constructs. It includes mechanisms for monitoring workflow execution and dynamic rescheduling in order to optimize workflow performance.

In the ASKALON Grid application development and computing environment, users compose workflows graphically using UML-based workflow composition and modeling service. Alternatively, users con programmatically describe workflows using AGWL, at a high level of abstraction that does not comprise any details of the underlying middleware platform or resources. The UML-based workflow specifications can be automatically translated into AGWL specifications, though AGWL serves as input to the ASKALON runtime environment. At this point, ASKALON transforms the abstract workflow representations into concrete and executable workflow specifications, and maps workflow abstract tasks onto the available resources. This activity is accomplished by ASKALON's scheduler in its Execution Engine. ASKALON has developed a light-weight just-in-time scheduling strategy [22]. This strategy has been used to support dynamic scheduling of scientific workflows in Grids adapting to the executing resources' availability and conditions. The scheduler implements different fault-tolerance and fair-sharing policies (job restarting in case of failure or job submission to alternative resources). These

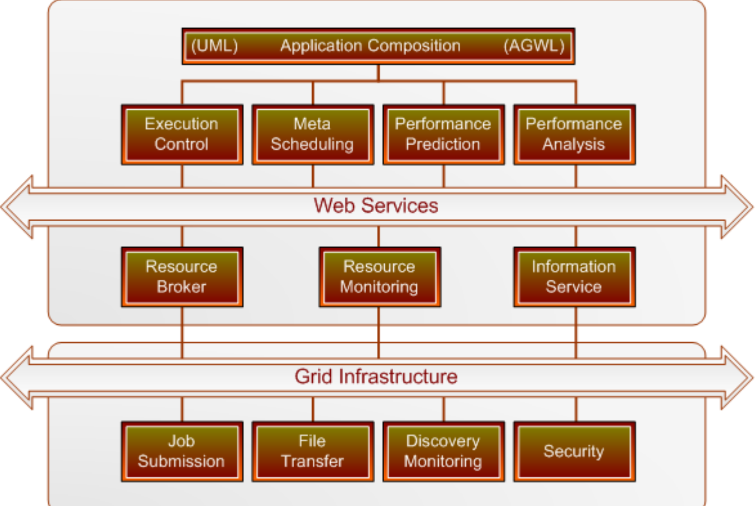

Fig. 3.1. Askalon's architecture

techniques are complemented with checkpointing and rollback at workflow level.

3.2 DVega

DVega [83, 84] is a scientific workflow engine which adapts itself to the changing availability of resources, minimizing the human intervention. DVega utilizes Reference nets [85], a specific class of Petri nets, for composing workflow tasks in a hierarchical way and the Linda [34] communication paradigm for isolating workflow tasks from resources. Workflow tasks interact with resources by exchanging messages: in particular workflow tasks send/receive messages to/from Linda and the existing forwarder-receiver components in DVega are responsible for taking the messages from Linda and sending them to the resources and vice-versa. DVega's architecture, shown in Figure 3.2, is completely built upon service-oriented principles and by means of a tuple space shields workflows from the heterogeneity of the middleware.

One of the aspects of the architecture that is worth highlighting is the mapping between tasks and resources. A workflow task receives its inputs, generates a tuple with them and writes it into the tuple-space (Linda). After that, the workflow task is suspended until the expected result is back: once the tuple is in Linda, the corresponding proxy takes the tuple, transforms it into the suitable format and forwards it to the destination resource. When the result arrives to the proxy, it is transformed into a message and written into the tuple-space. Then, the workflow tasks withdraws the expected tuple

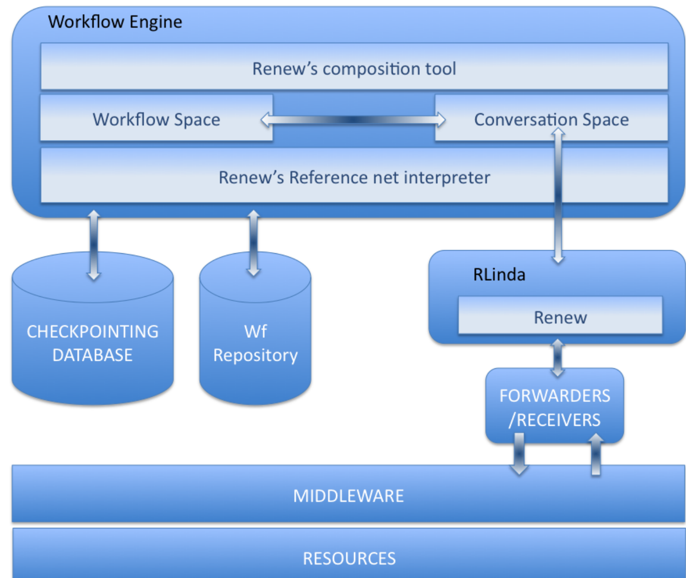

Fig. 3.2. DVega's architecture

containing the result. The main advantage of this approach is that a workflow task will have only to indicate the type of interaction required, without explicitly sending a message to a specific proxy. In consequence, proxies can be added and modified at runtime, without having to stop the execution, and can be shared by workflow tasks.

3.3 GridAnt

GridAnt [86] is a client-side workflow management system that can be used for executing workflows on Grid resources. It extends the Apache Ant build system, an existing commodity tool for controlling build processes in Java, by adding additional components for authenticating, querying and transferring data between Grid resources. Furthermore, it provides a graphical visualization tool for monitoring the progress of the workflow execution. GridAnt is similar in functionality to the Condor DAGMan workflow manager.

Some disadvantages of GridAnt like the inability to concurrently execute targets, the lack of full iteration support, the difficulties in expressing conditional execution, etc, have been behind the motivation of the development of the *Java CoG Kit Karajan* (see Section 3.9), a more streamlined and powerful workflow framework.

3.4 Gridbus

Gridbus [90] is another workflow management system that allows users to specify workflows using a simple XML-based workflow language. A workflow coordinator (WCO) is responsible for monitoring the status of the tasks in the workflow and activating the child tasks when they become eligible. An event service server (ESS) is used for notification purposes. Active tasks register their status with the ESS, which in turn notifies the WCO. Based on the status received from the ESS, WCO may activate the child tasks (similar to DAGMan functionality). It allows users to specify execution resources for each task in the workflow. Alternatively, it is also able to discover resources using Grid information services.

3.5 Grid-Flow

Grid-Flow [38] is another Petri net-based workflow system. The Grid-Flow architecture (shown in Figure 3.3) is designed as a Service-Oriented Architecture with multi-layer component models. Their approach to Petri net-based tools is to employ a generic modeling graphical interface called Generic Modeling Environment (GME) to define the instance of the Petri net user interface tools. The Petri net specification is then compiled into a lower level *Grid-Flow Description Language* (GFDL) which is executed by the workflow engine. Grid-Flow assumes a hierarchical Grid structure consisting of local Grids managed by the Titan resource management and a global Grid that is an ensemble of local Grids, it then simulates workflow execution on the global Grid in order to find a near optimal schedule. The best workflow schedule is enacted on the local Grids using ARMS agents. The system also integrates heterogeneous data and distributed analysis tools through the unified platform WebRun [39].

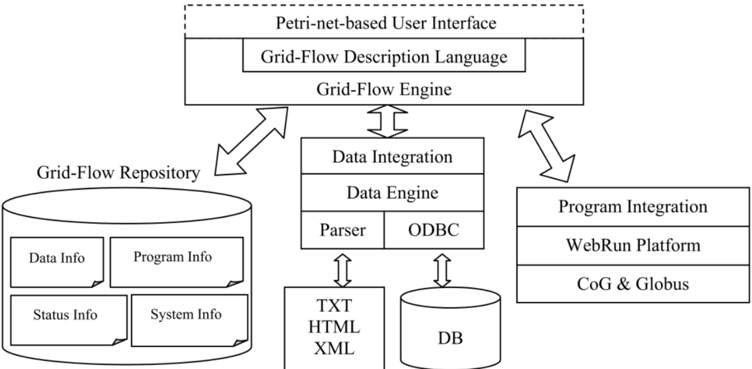

Fig. 3.3. Grid-Flow's architecture

The Petri-net-based user interface, implemented within a graphical modeling environment, can help users design the workflow via a graphical editor, translate the workflow specification into the GFDL, and monitor the execution of the workflow process. The GFDL, which conveys the specifications of workflow processes, acts as a bridge connecting the Petri-net models with the Grid-Flow engine. The Grid-Flow engine layer is responsible for interpreting user-defined workflow processes and responding to users' monitoring. More importantly, it coordinates the activities and execution of user workflow using appropriate services provided by the layer of Grid-enabled data and program integration. The data and program integration layer of the system infrastructure plays a critical role of interconnecting distributed computing resources in the whole system.

3.6 GWES

The *General Workflow Execution Service* (GWES) [45] (formerly the *Grid Workflow Execution Service*) is a workflow system which proposes a multi-level abstraction and semantic-based solution to facilitate the de-coupling between tasks and resources or services. With the implementation of a plugin concept for arbitrary workflow activities, it has now a broader area of application, not limited to the orchestration of Grid and Web Services. The GWES coordinates the composition and execution process of workflows in arbitrary distributed systems, such as SOA, Cluster, Grid, or Cloud environments.

The GWES processes workflows that are described using the *Grid Workflow Description Language* (GWorkflowDL), which is based on the formalism of High-Level Petri nets (HLPNs) [47]. In the workflow specifications, transitions represent tasks and tokens in the nets represent data flowing through the workflow. Hierarchical Petri nets are also exploited to model hierarchical workflow specifications and to support the multi-level abstraction. An abstract task on top of the hierarchy can be mapped dynamically at runtime by refining the workflow structure.

In Figure 3.4 solution for the automatic mapping of dynamic workflows to determine the appropriate and available resources. User requests are first mapped to abstract workflows (yellow). Each activity will be initially assigned with the help of the resource matcher service candidates (blue) that provide the appropriate functionality. A scheduler selects one of the candidates (green) and carries out the activity to the appropriate resources.

In accordance with the Petri net refinement paradigm, places and transitions within the net can be refined with additional Petri nets, thereby facilitating the modeling of large real world systems. However, the GWorkflowDL does not support the inherent modeling of the dynamic refinement process itself and, consequently, the workflow structure is modified dynamically by the workflow engine. This requires from the user a deep knowledge of the workflow engine functionality. For instance, there is no simple construct in the

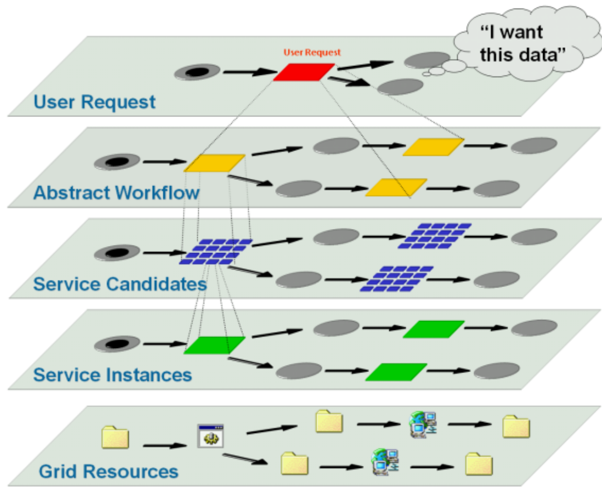

Fig. 3.4. GWES Workflow refinement

GWorkflowDL. Additionally, GWES also supports exception handling in the hierarchical scientific workflows thereby the workflow engine can modify part of the workflow structure upon a failure, providing great levels of dynamism and flexibility. Nevertheless, GWES does not support a clear separation of the exception handling from the application data and control flow, apart from simple rescheduling techniques and checkpoint/restart functionalities.

3.7 GRMS

The *Grid(Lab) Resource Management System* GRMS [53] is a resource management system with a workflow engine that executes and manages jobs on remote Grid resources. One can submit to GRMS workflow experiments based on an XML workflow schema, defining flexible mechanisms for dynamic workflow control, including various types of precedence constraints, different locations of the final data products and executables, etc. All of these features allow end users to speed up remote workflow calculations and improve data management mechanisms. The motivations and ideas for this workflow management were based on some experiences gained in two projects, GridLab and PROGRESS, where the authors could deal with real use cases and end users' requirements for workflow management. GRMS was designed as an independent set of components for resource management processes which can take advantage of various low-level Core Services, in particular taken from Globus Toolkit (GRAM [26], GridFTP [7], GIIS/GRIS [91]) and Mercury Grid [1] Monitoring System.

[1] http://www.lpds.sztaki.hu/mercury/

3.8 ICENI

ICENI [63], the *Imperial College e-Science Network Infrastructure*, is a system for workflow specification and enactment on Grids. The user creates an abstract workflow in an XML-based language. The ICENI system is responsible for making the workflow concrete by finding suitable implementations of the components in the workflow, mapping the components to appropriate resources, and monitoring the instantiation of the concrete workflow on the mapped resources. Once a schedule for the workflow has been computed, the ICENI system tries to reserve the resources at the desired time by negotiating with the resource provider.

The authors distinguish between an *e-Scientists conceptual workflow* to describe tasks to be performed with dependencies and a *middleware workflow* for execution on the Grid. The architecture of ICENI supports deployment, performance, reliability, and charging for resource use. The current ICENI architecture experiences with e-Science projects, such as the Grid Enabled Integrated Earth system model (GENIE), e-Protein, and RealityGrid.

3.9 Java CoG Kit-Karajan

In Java CoG Kit-Karajan [87], users can compose workflows through an XML-scripting language as well as with an equivalent more user-friendly language called *K*. Not only both languages support hierarchical workflow descriptions based on Directed Acyclic Graphs (DAGs), but also have the ability to use control structures such as if, while and parallel in order to express easy concurrency.

The architecture of the Java CoG Kit Karajan framework, displayed in Figure 3.5, contains the workflow engine that interfaces with high level components, namely a visualization component that provides a visual representation of the workflow structure and allows monitoring of the execution, a checkpointing subsystem that allows the checkpointing of the current state of the workflow, and a workflow service that allows the execution of workflows on behalf of a user. A number of convenience libraries enables the workflow engine to access specific functionalities.

Workflow specifications can be visualized in the system. The analysis of dynamism support in Karajan reveals that workflows can actually be modified during runtime through two mechanisms. The first one is through the definition of elements that can be deposited in a workflow repository that gets called during runtime. The second one is through the specification of schedulers that support the dynamic association of resources to tasks. The execution of the workflows can either be conducted through the instantiation of a workflow on the users client or can be executed on behalf of the user on a service. Besides, the execution engine of the system also features workflow checkpointing and rollback.

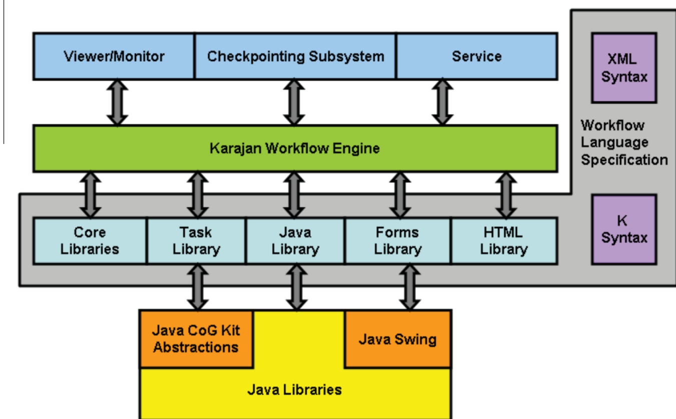

Fig. 3.5. The components of the Java CoG Kit Karajan

3.10 Kepler

Kepler [3, 60] provides a graphical user interface and a run-time engine that can execute workflows (with an emphasis on ecology and geology) either from within the graphical interface or from a command line. Kepler extends Ptolemy II [2], a software system for modeling, simulation, and design of concurrent, real-time, embedded systems. It is a Java-based application that is maintained for the Windows, OSX, and Linux operating systems and freely available under the BSD License.

Kepler is used in several large Grid projects where the management of biological data analysis workflows is critical. The approach Kepler takes is based on an actor-oriented model which allows hierarchical modeling and dataflow semantics. To support the interaction with web services Kepler uses a form of actor proxy for each web services that is invoked. In addition they have created a set of Grid actors for doing GridFTP [7] file management and Globus GRAM [26] execution.

In Kepler, users develop workflows by selecting appropriate components called actors and placing them on a design canvas, after which they can be "wired" together to form the desired workflow graph. Actors have input and output ports which provide the communication interface to other actors. Workflows can be hierarchically structured, yielding composite actors that encapsulate subworkflows. A feature of KEPLER, inherited from Ptolemy II, is that the overall execution and component interaction semantics of a workflow is not defined by the components, but is factored out into a separate component called a director. Taken together, workflows, actors, ports, connections, and directors represent the basic building blocks of actor-oriented modeling and design [57]. In order to support execution over Grid resources, Kepler has

[2] http://ptolemy.eecs.berkeley.edu/ptolemyII/

defined a set of Grid actors for access authentication, file coping, job execution, job monitoring, execution reporting, storage access, data discovery, and service discovery.

Kepler workflow system implements also exception handling: as described in [66], for each node in the hierarchy, its input data is checkpointed in advance so that in case of exception the failed descendant sub-workflow can be replaced by an alternative one. When a sub-workflow within a Checkpoint produces an error event, all execution within the Checkpoint is stopped. The Checkpoint handles the error itself (by re-executing the primary, or running an alternate sub-workflow), or passes it up the workflow hierarchy. The maximum number of times to retry the primary or an alternative sub-workflow is configurable. Once the retry limit is exceeded, the error is sent up the workflow hierarchy to the nearest enclosing Checkpoint. The criteria for selecting a candidate from the alternative candidates is not specified and no evidence is given whether the context in which the exception arose can be taken into account for selecting the candidate. Besides, the candidate list has to be defined at development time, thus limiting the flexibility and dynamism of the approach.

3.11 Pegasus

The Pegasus [19, 20] project encompasses a set of technologies to execute workflow-based applications in a number of different environments, i.e., desktops, campus clusters, grids, clouds, Condor pools, high-performance TeraGrid systems. The worfklow management system of Pegasus can manage the execution of complex workflows on distributed resources and it is provided with a sophisticated error recovery system. Pegasus can use as middleware Condor, Globus, or Amazon EC2, and takes a modular approach to workflow systems, working in partnership with workflow execution engine DAGMan, which was developed by the Condor team at the University of Wisconsin, USA. A simplified view of the system is shown in Figure 3.6.

Pegasus, which stands for Planning for Execution in Grids, focuses on data intensive applications related with atomic physics, astronomy, biology, etc. Its scientific workflows can be modeled at a high level of abstraction without the need to worry about the actual execution environment. The abstract workflows can be constructed by using Chimera [31], thereby scientists are allowed to specify data dependencies among tasks and then, an abstract workflow specification can be obtained. Then, with information about the available resources, Pegasus generates an executable workflow. Alternatively, abstract workflows can also be written by users directly.

In Pegasus, workflows are adapted to a changing environment where resources can come and go suddenly, by mapping only portions of the abstract workflow to resources at a time (also known as deferred mapping). Pegasus' objectives have been [58]: (i) to dynamically adjust resource allocation decisions in the light of run-time feedback on the performance of the clusters

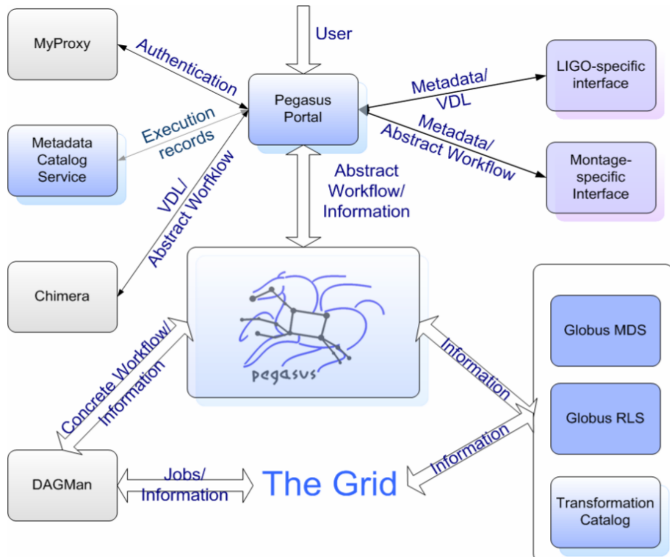

Fig. 3.6. Simplified view of the Pegasus system

onto which workflows are being compiled; and (ii) to obtain that dynamic behavior through minimal intervention into the existing Pegasus infrastructure. Pegasus can consider resource allocation on clusters that might be used by several users at the same time. This allows it to introduce adaptivity into an environment whose performance is not well known in advance, and in which there is limited control over the execution of individual jobs.

3.12 Taverna

Taverna [67] is an open source tool for designing and executing workflows. Its own workflow definition language is characterized by an implicit iteration mechanism (single node implicit parallelism). The Taverna team has primarily focused on supporting the Life Sciences community (biology, chemistry and medical imaging) although does not provide any analytical or data services itself. It supports different types of Web services, including WSDL-based, Soaplab, BioMoby and BioMart services. Originally designed to execute Web service based workflows, Taverna can now interact with arbitrary services. Taverna workflows can have a hierarchical structure and tasks can be abstract which will be refined at runtime. It is a domain-specific system and the workflows are limited to the specification and execution of ad hoc in silico experiments using bioinformatics resources. These resources might include information repositories or computational analysis tools providing a Web service based or custom interface. Workflows are enacted by the FreeFluo enactment

engine, and progress can be monitored from the Taverna workbench. Taverna architectural diagram is shown in Figure 3.7.

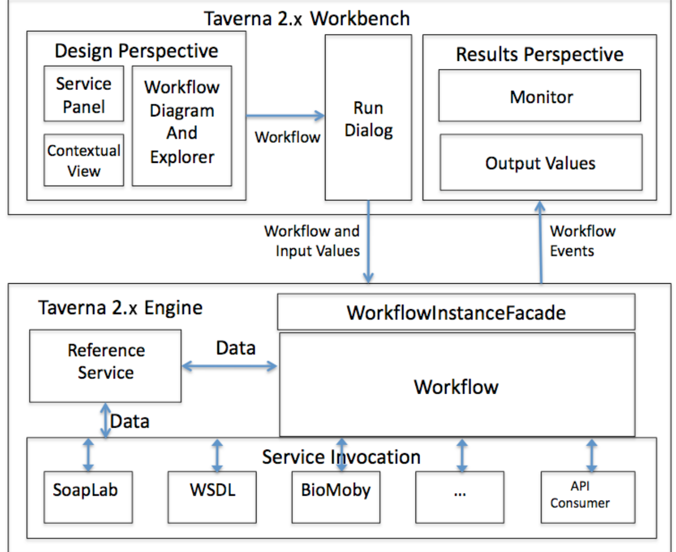

Fig. 3.7. Overview block diagram of Taverna architecture

Taverna is part of the *myGrid* [3] project, which is building middleware to support data-intensive experiments in molecular biology. Taverna provides over 1000 services that can be used as components in workflows. However, solving the problems of service discovery and selection become non-trivial parts of the process when the potential catalog of workflow components is large. Another issue is that by using the service as a single component in a workflow, one bypasses many interesting capabilities that some stand-alone services have, like powerful data analysis and visualization user interfaces. This passes the problem of doing final data analysis and visualization to the end of the workflow process where it is often hard to replace what has been given up along the way. A major problem that Taverna addresses is that of capturing the full metadata context including the provenance of all aspects of the scientific experiment that the workflow represents. This includes the data derivations and the workflow's audit trail of invoked services. A critical feature of e-science is the ability to enable the repeatability of experiments.

Two techniques are provided to the user for dealing with faults [67]: task retry and alternative task. With task retry, workflow designers are allowed to indicate the maximum number of times that a task will be retried in case of execution failure. This can also be applied to sub-workflows whenever they

[3] http://www.mygrid.org.uk/

fail. The alternative task technique allows users to specify a different task in case of failure, after a maximum number of retries have been attempted. However, alternative sub-workflows cannot be specified nor the fault can be propagated up in the hierarchy.

3.13 Triana

Triana [72] is a problem solving environment that combines a visual interface with data analysis tools. It can connect heterogeneous tools (e.g. Web services, Java units, JXTA services) on one workflow. Triana uses its own custom workflow language, although can use other external workflow language representations such as BPEL4WS [6] which are available through pluggable language readers and writers. It comes with a wide variety of built-in tools for signal-analysis, image-manipulation, desk-top publishing, etc.

At the user's level Triana provides a composition tool and a large toolbox of ready-to-use components. Workflows in Triana can have a hierarchical structure and can be defined in an abstract way that is later refined by the execution engine. The Triana workflow environment can operate in heterogeneous Grid and P2P environments simultaneously. It uses the Grid Application Toolkit (GAT) created by GridLab [4] for distributing the workflow components across Grids. Triana communicates with services through the *GAP Interface*, a subset of the GridLab GAT-API. The GAP Interface provides applications with methods for advertising, locating, and communicating with other services. The GAP Interface is, as its name suggests, only an interface, and therefore is not tied to any particular middleware implementation. This provides the obvious benefit that an application written using the GAP Interface can be deployed on any middleware for which there is a GAP binding, as shown in Figure 3.8.

Triana's support for fault tolerance [72] is generally user driven. For example, faults will generally cause workflow execution to halt, display a warning or dialog, and allow the user to modify the workflow before continuing execution. At workflow level lightweight checkpointing and the restart or selection of workflow management services are currently supported. At the middleware and task levels, all the listed faults can be detected by the Engine or GAT, except for deadlock, livelock and memory leaks. At the lowest level, machine crashes and network errors are recognized by GridLab GAT and the Triana Engine respectively, but recovering from these faults or preventing them is only planned for future versions.

[4] http://www.gridlab.org/

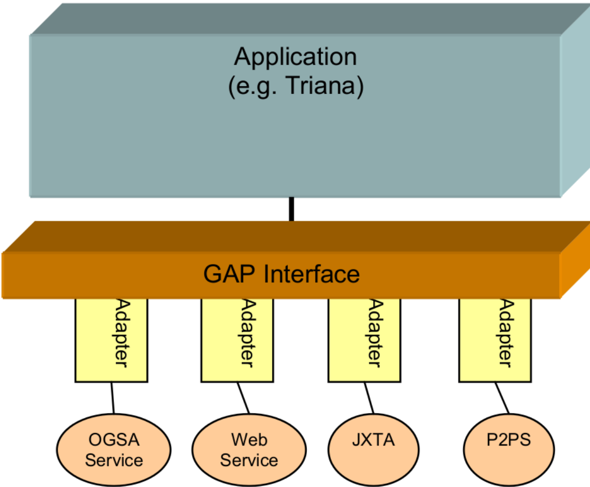

Fig. 3.8. The relationship between Triana, the GAP Interface, and GAP bindings

3.14 ScyFlow

ScyFlow [62] is a directed graph based workflow tool that is designed to manage NASA's large scale simulation and data analysis work. SkyFlow can handle both control flow and parameterized data flow within any given workflow.

ScyFlow is part of a larger set of component applications, whose coordination and interaction is managed by a "container" application called *ScyGate*. Other components include one for parameter study, one for transforming one data format into another data format, one for parsing ASCII data file, etc. ScyFlow consists of two major components: (i) *ScyFlowVis* which provides the visual interface for specifying workflows, translates the visual flow graph into internal formats (XML, dependency list, pseudo-code) for storing and for communicating with the other components, and can also be used as a visual interface to monitor the execution of workflows, and (ii) *ScyFlowEngine* which provides the set of services required for the overall execution of the workflow across distributed grid resources, taking into account the specified control and data dependencies.

3.15 UNICORE Rich Client

UNICORE [76] is a project to develop a Grid infrastructure and a computing portal for users to access the Grid resources seamlessly. Its job model supports directed acyclic graphs with temporal dependencies. The Unicore graphical tools allow a user to create a job flow that is then serialized by a Unicore user client and sent to a server for enactment. The server is responsible for dependency management and execution of the jobs on the target resources.

UNICORE has a two layered design with a separation of the workflow engine and the service orchestrator in order to achieve better scalability, but also to offer the possibility to plug-in domain-specific workflow languages and workflow engines. The workflow engine originates from the EU-project Chemomentum [73]. Besides simple job-chains, while and for loops, workflow variables and conditional execution are supported.

The workflow capabilities are offered to the users via the *UNICORE Rich Client* (URC). URC was based on the Eclipse platform [5] with the intent of lowering the entry barrier for new users, as many of them are already familiar with the tool, having a smoother integration into different platforms and making it extremely extensible. The client allows to detail resource requirements for jobs (e.g. required number of CPUs, amount of RAM); dedicated panels deal with setting up security options so that users can specify whom they trust and how to identify themselves on the Grid.

The UNICORE workflow system offers basic workflow constructs like while-loops for repeating certain tasks automatically and if-statements for altering the flow of execution based on the evaluation of conditions. These constructs can also be nested in order to address more complex tasks. In addition, the system supports the parallel execution of many similar jobs or sub-workflows by introducing the for-each-loop. This construct helps to minimize the effort of creating workflows with many jobs and reduce the size of the resulting workflow descriptions. For covering these scenarios, the for-each-loop offers two different modes of operation: (i) iteration over sets of differing parameter values, (ii) iteration over a set of files.

3.16 Grid-based Data Mining

A wide set of applications is being developed for the exploitation of Grid platforms. Since application areas range from scientific computing to industry and business, specialized services are required to meet needs in different application contexts. In particular, *data Grids* have been designed to easily store, move, and manage large data sets in distributed data-intensive applications. Besides core data management services, *knowledge-based Grids*, built on top of computational and data Grid environments, are needed to offer higher-level services for data analysis, inference, and discovery in scientific and business areas [65]. In some papers [5, 16, 48] it is claimed that the creation of *knowledge Grids* is the enabling condition for developing high-performance knowledge discovery processes and meeting the challenges posed by the increasing demand of power and abstractness coming from complex problem solving environments.

Several systems of distributed data mining exploiting the Grid infrastructure have been designed and implemented [77]. Some of them are:

[5] http://www.eclipse.org/

- The *DataMiningGrid* [75] is an environment suitable for executing data analysis and knowledge discovery tasks in a wide range of different application sectors, including the automotive, biological and medical, environmental and ICT sectors. Based on open-source Grid middleware, it provides functionality for tasks such as data manipulation, resource brokering and application searching according to different data mining tasks and methodologies, and supporting different types of functionality for parameter sweeps. In summary, it is a Grid software with all the generic functionality of its component middleware, but with additional features that ease the development and execution of complex data mining tasks.
- *Discovery Net* [1] allows users to integrate data analysis software and data sources made available by third parties. The building blocks are the so-called *Knowledge Discovery Services*, distinguished in *Computation Services* and *Data Services*. Discovery Net provides services, mechanisms and tools for specifying knowledge discovery processes. The functionalities of Discovery Net can be accessed through an interface exposed as Web service.
- *GridMiner* [8] aims at covering the main aspects of knowledge discovery on Grids. Key components in GridMiner are *Mediation Service*, *Information Service*, *Resource Broker*, and *OLAP Cube Management*. These are the so called GridMiner Base services, because they provide basic services to GridMiner Core services. GridMiner Core services include services for data integration, process management, data mining, and OLAP.

In the reminder of the thesis we will introduce the two frameworks which have been developed during this research work: Weka4WS, and the Knowledge Grid.

4

Weka4WS

Weka4WS is a framework which extend the widely used Weka toolkit for supporting distributed data mining on Grid environments. Weka provides a large collection of machine learning algorithms written in Java for data pre-processing, classification, clustering, association rules, and visualization, which can be invoked through a common graphical user interface. In Weka, the overall data mining process takes place on a single machine, since the algorithms can be executed only locally. The goal of Weka4WS is to extend Weka to support remote execution of the data mining algorithms through WSRF Web Services. In such a way, distributed data mining tasks can be concurrently executed on decentralized Grid nodes by exploiting data distribution and improving application performance.

Distributed data mining applications can be built in Weka4WS using either *Explorer* or *Knowledge Flow*, two of the four Weka front ends which have been extended to allow to run single or multiple data mining tasks on remote hosts of the Grid. In Explorer a data mining application is set up using menu selection and form filling: at the end of the set up a drop down menu shows a list of locations where the algorithm may be executed. Knowledge Flow allows to build workflows representing data mining applications: by clicking on the nodes of the data mining algorithms the user can choose the locations where they will be executed. The locations, both in Explorer and in Knowledge Flow, may be either specified by the user or automatically chosen by the system.

4.1 System Goals

The objective that guided us in the design of Weka4WS is allowing users to perform distributed data mining on the Grid in a easy and effective way. In particular, the goals of the system are supporting both:

- the execution of a single data mining task on a remote Grid node; and

- the execution of multiple data mining tasks, defined as a workflow, on multiple nodes of a Grid.

Supporting remote execution of single or multiple data mining tasks allows users to exploit the computational power and the distribution of data and algorithms of a Grid.

To make as easier as possible the use of our framework, we decided to start from a well established data mining environment (the Weka toolkit) and to extend it with remote execution features. In this way, domain experts can focus on designing their data mining applications, without worrying about learning complex tools or languages for Grid submission and management. Indeed, the Weka4WS visual interface allows users to set up their data mining tasks or workflows as in Weka, with the additional capability of specifying the Grid nodes where to execute the data mining algorithms.

To achieve integration with standard Grid environments, Weka4WS uses the *Web Services Resource Framework* (*WSRF*) as enabling technology. WSRF is a family of technical specifications concerned with the creation, addressing, inspection and lifetime management of *stateful resources* using Web Services. To enable remote invocation, Weka4WS exposes all the data mining algorithms originally provided by Weka through a WSRF-compliant Web Service, which can be easily deployed on the available Grid nodes.

4.2 System Architecture

Weka4WS has been developed using the Java *Web Services Resource Framework* (WSRF) libraries provided by *Globus Toolkit* (GT) and uses its services for the standard Grid functionalities such as security and data transfer. The application is made up of two separate parts:

- **User node**: it is the part where the client side of the application runs. It is made by an extension of the Weka Graphical User Interface, a *Client Module* and the Weka library. It requires the presence of the *Globus Java WS Core* (a component of Globus Toolkit);
- **Computing node**: the server side of the application, it allows the execution of data mining tasks through Web Services. A Globus Toolkit full installation is required.

Data to be mined may be located either at the user node, or at the computing node, or at some other remote resource (for example some shared repositories). When data are not available at the computing node they are transferred by means of GridFTP [2], a high-performance, secure and reliable data transfer protocol which is part of Globus Toolkit.

Figure 4.1 shows the Weka4WS software components of the user node and the computing node, and the interactions among them. A user operates on the *Graphical User Interface* to prepare and control the data mining tasks:

those to be executed locally will be managed through the local *Weka Library*, whereas those to be executed on a remote host will be handled by the *Client Module* which will interact with the computing node using the services offered by the Globus Java WS Core installed on the user node machine.

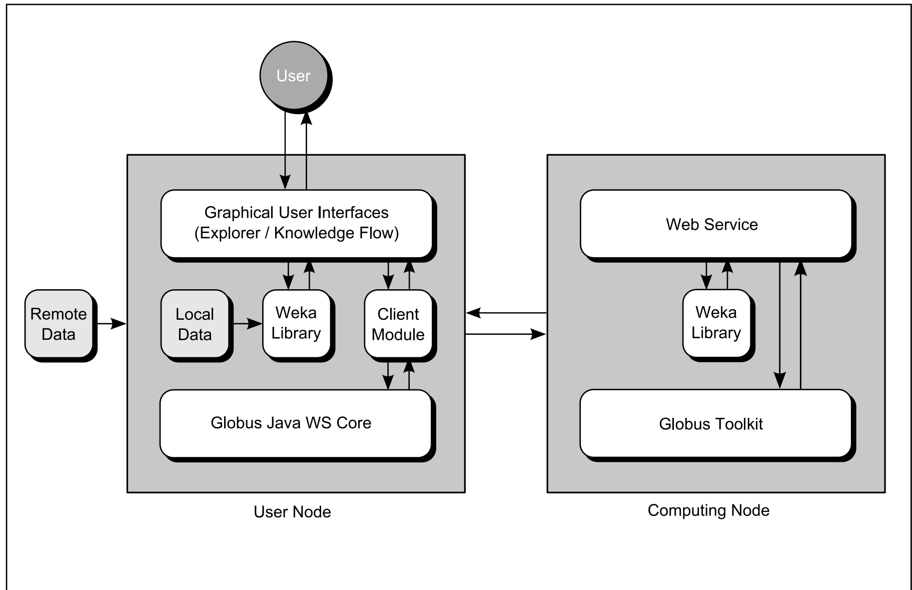

Fig. 4.1. Weka4WS user node and computing node

In the best network scenario the communication between the client module and the Web Service is based on the "push-style" [88] mode of the *NotificationMessage* delivery mechanism, where the client module is the *NotificationConsumer* and the Web Service is the *NotificationProducer*. The client module invokes the service and waits to be notified of the required task completion.

There are certain circumstances in which the basic "push-style" of NotificationMessage delivery is not appropriate, for example when the client resides behind a NAT Router/Firewall as shown in Figure 4.2, because the messages sent to the client will be blocked unless they are over a connection initiated by the client itself.

When the client subscribes to notification of the completion of the required task it also passes to the service a port number, randomly generated, to which the client will be listening to receive the notification: from that moment the client will act as server, awaiting for notifications to the given port. When the results are ready at the computing node a connection is initiated by the computing node but the attempt to send the notification fails because it is blocked by the user node NAT Router/Firewall. In the scenario just depicted the only way for the client module to get the results is to work in "pull-style"

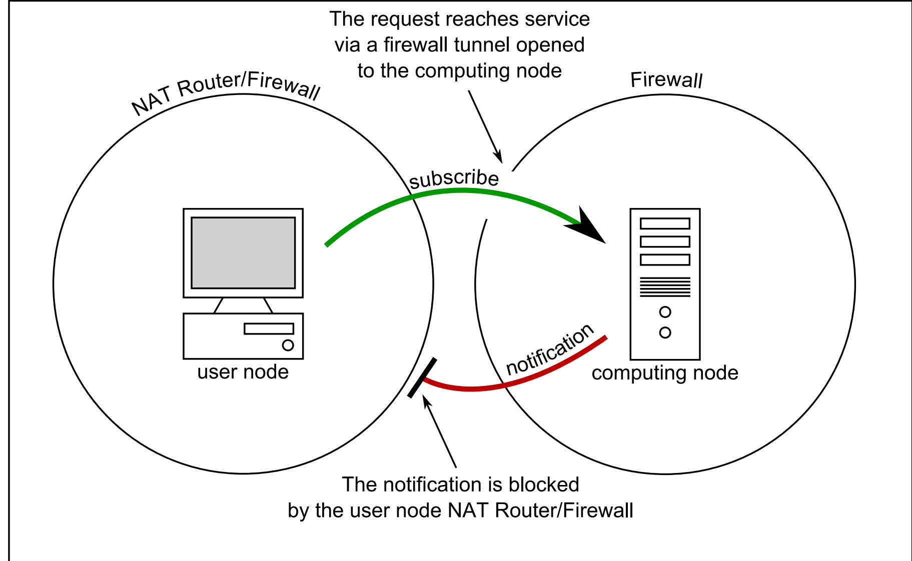

Fig. 4.2. Notifications are blocked when the client runs on a machine behind a NAT Router/Firewall

mode, that is to continuously try to *pull* (retrieve), at certain given intervals, the results from the computing node until they will be available.

In order to make the system to automatically adapt to all possible network scenarios in a way transparent to the user, the user node starts in pull-mode by default. At the moment of the notification subscription it also asks for the immediate send of a dummy notification whose solely purpose is to check whether the user node may receive notifications: when and if the user node will receive this dummy notification it will switch to the "push-style" mode, otherwise it will persist in the "pull-style" one.

At the computing node side, the *Web Service* uses the *Globus Toolkit* services to interact with the user node and answers to its requests by invoking the requested algorithm in the underlying *Weka Library*. The relation between Web Services and Globus Toolkit is threefold: Web Services are built using some libraries provided by Globus Toolkit, they are deployed and hosted over the Globus Toolkit container, and they use some Globus Toolkit services like security and the notification mechanism [36] to access data and interact with the user node.

Let's now examine the two components in more details.

4.2.1 User node

The user node is formed by three components: the *Graphical User Interface* (GUI), the *Client Module* and the Weka library. The GUI is made by the Weka

Explorer and *Knowledge Flow* components, extended to support the execution of remote data mining tasks. Explorer is a tool for "exploring" data through data preprocessing, mining and visualization. Knowledge Flow has essentially the same features of the Explorer but with a drag-and-drop interface which allows to build data mining workflows.

The local tasks are executed invoking the local *Weka library*, while the remote ones are performed through the *Client Module* which acts as intermediary between the GUI and the remote Web Services. Each task is carried out in a thread of its own thus allowing to run multiple tasks in parallel, either using Explorer or Knowledge Flow.

The remote hosts addresses are loaded from a text file located inside the application directory. This text file is read in background when the application is launched and for each remote host is checked that:

- the Globus container and GridFTP are running;
- the Weka4WSService is deployed;
- the versions of the client and the service are the same.

Only those hosts which pass all the checks are made available to the user in the GUI. In order to take into account possible alteration of the Grid network configuration without having to restart the application, the remote hosts addresses may be reloaded at any time simply by pressing a given button provided for the purpose.

A static list of addresses stored in a text file on the client machine is not actually the ideal solution, as Globus Toolkit already provides a point of inquiry (the *Globus Index Service*) regarding the characteristics of a physical system in a Grid and allows clients or agents to discover services that come and go dynamically on the Grid. The choice of using that file has been taken as temporary solution in order to speed up the development of a Weka4WS prototype and focus on more demanding and crucial aspects of the application. The use of the Globus Index Service in place of the text file is planned to be introduced in the near future versions of Weka4WS.

4.2.2 Computing node

A computing node includes two components: a *Web Service* and the *Weka Library*. The *Web Service* answers the user node query by invoking the requested algorithm in the underlying *Weka Library*. The invocation of the algorithms is performed in an asynchronous way, that is, the client submits the task in a non-blocking mode and results are either notified to it whenever they are computed (push-style mode) or they are repeatedly checked for readiness by the client (pull-style mode) depending on the network configuration, as described in Section 4.2.

Table 4.1 lists the operations provided by each Web Service in the Weka4WS framework. The first three operations are used to request the execution of a particular data mining task; operations in the middle row of

the table are some useful extra operations, while the last four operations are related to WSRF-specific invocation mechanisms.

Table 4.1. Operations provided by each Web Service in the Weka4WS framework

Operation	Description
`classification`	Submits the execution of a classification task.
`clustering`	Submits the execution of a clustering task.
`associationRules`	Submits the execution of an association rules task.
`stopTask`	Explicitly requests the termination of a given task.
`getVersion`	Returns the version of the service.
`notifCheck`	Checks whether the client is able to receive notifications.
`createResource`	Creates a new stateful resource.
`subscribe`	Subscribes to notifications about resource properties changes.
`getResourceProperty`	Retrieves the resource property values.
`destroy`	Explicitly requests destruction of a resource.

The `getVersion` operation is invoked at the moment of the hosts check, described in Section 4.2.1, and is used to check whether the client and the service versions are the same. The `notifCheck` operation is invoked just after the `subscribe` operation to check whether the client is able to receive notifications, as described in Section 4.2. All the other operations are described in Table 4.1.

The parameters required for each operation are shown in Table 4.2; the operations `getVersion` and `notifCheck` do not require any parameter.

The `taskID` field is required solely for the purpose of stopping a task. The `algorithm` field is of a complex type, shown in Table 4.3, which contains two subfields: `name`, a string containing the Java class in the Weka library to be invoked (e.g., `weka.classifiers.trees.J48`), and `parameters`, a string containing the sequence of arguments to be passed to the algorithm (e.g., `-C 0.25 -M 2`). The `dataset` and `testset` are other two fields, of a complex type which contains four subfields: `fileName`, a string containing the name of the file of the dataset (or test set), `filePath`, a string containing the full path (file name included) of the dataset , `dirPath`, a string containing the path (file name excluded) of the dataset, and `crc` which specifies the checksum of the dataset (or test set) file.

The `classIndex` field is an integer designating which attribute of the dataset is to be considered the class attribute when invoking a classifying or clustering algorithm. The `testOptions` field is of a complex type containing three subfields: `testMode`, an integer representing the test mode to be applied to the classifying or clustering algorithm (1 for *Cross-validation*, 2 for *Percentage split*, 3 to *use the training set* and 4 to *use a separate test*

Table 4.2. Weka4WS Web Services input parameters

Parameters type	Field name	Field type
classificationParameters	`taskID`	long
	`algorithm`	algorithmType
	`dataset`	datasetType
	`testset`	datasetType
	`classIndex`	int
	`testOptions`	testOptionsType
	`evalOptions`	evalOptionsType
clusteringParameters	`taskID`	long
	`algorithm`	algorithmType
	`dataset`	datasetType
	`testset`	datasetType
	`classIndex`	int
	`testOptions`	testOptionsType
	`selectedAttributes`	array of int
associationRulesParameters	`taskID`	long
	`algorithm`	algorithmType
	`dataset`	datasetType
stopParameters	`taskID`	long

set), `numFolds` and `percent` are two optional fields used when applying a *Cross-validation* or a *Percentage split* test mode respectively.

The `evalOptions` field is used specifically for the classification algorithms and contains several subfields like `costMatrix` (to evaluate errors with respect to a cost matrix), `outputModel` (to output the model obtained from the full training set), and others shown in Table 4.3. The field `selectedAttributes` is used specifically for the clustering algorithms and contains those attributes in the data which are to be ignored when clustering.

4.3 Graphical User Interface

The application starts with the three windows shown in Figure 4.3:

- the *Gui Chooser* (left side in Figure 4.3), used to launch Weka's four graphical environments;
- the remote hosts list checking window (top right side in Figure 4.3), used to give a visual confirmation of the hosts checking described in Section 4.2.1;
- the Grid Proxy Initialization window (middle right side in Figure 4.3), automatically loaded at startup only if the user credentials are not available or have expired.

Table 4.3. Weka4WS Web Services input parameters fields types

Fields type	Subfield name	Subfield type
algorithmType	`name`	string
	`parameters`	string
datasetType	`fileName`	string
	`filePath`	string
	`dirPath`	string
	`crc`	long
testOptionsType	`testMode`	int
	`numFolds`	int
	`percent`	int
evalOptionsType	`costMatrix`	string
	`outputModel`	boolean
	`outputConfusion`	boolean
	`outputPerClasss`	boolean
	`outputSummary`	boolean
	`outputEntropy`	boolean
	`rnd`	int

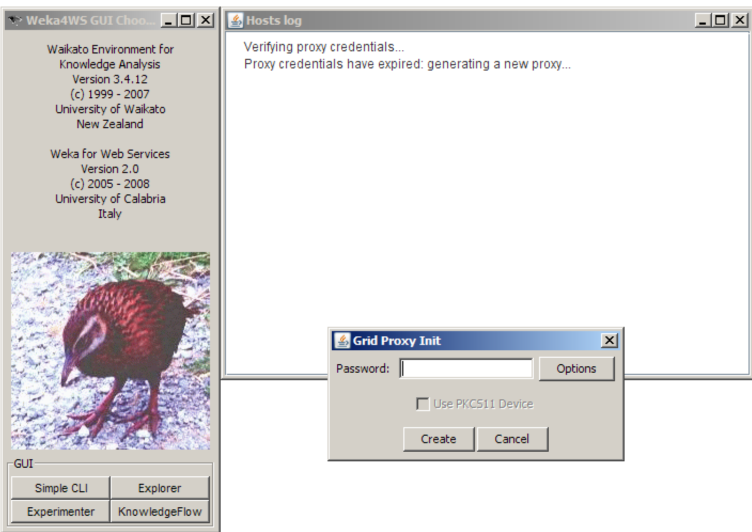

Fig. 4.3. Weka4WS start up

We will now examine the two Weka components which have been extended in Weka4WS: *Explorer* and *Knowledge Flow*.

4.3.1 Explorer

Explorer, Weka's main graphical user interface, is a comprehensive tool with six tabbed panes, each one dedicated to a specific Weka facility like data preprocessing (loading from file, URL or database, filtering, saving, etc.), data mining (classification, clustering, association rules discovery) and data visualization. A more detailed description of Explorer may be found in [89].

In Weka4WS the Explorer component is essentially the same as the Weka one with the exception of the three tabbed panes associated to classification, clustering and association rules discovery: in those panes the two buttons for starting and stopping the algorithms have been replaced with a *Control Panel*, and a button named *Proxy* has been added in the lower left corner of the window. Modifications are highlighted in Figure 4.4.

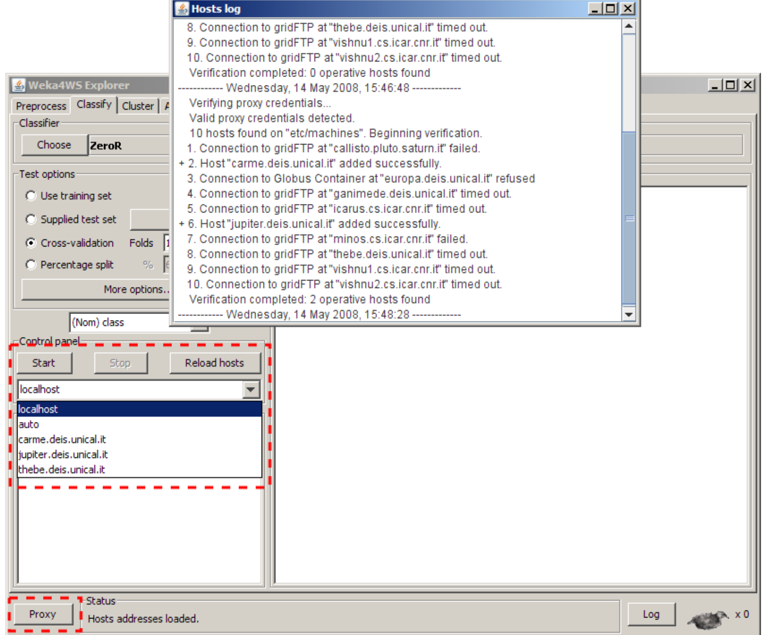

Fig. 4.4. Weka4WS Explorer: Control Panel and hosts reloading

The drop down menu in the Control Panel allows to choose either the exact Grid location where we want the current algorithm to be executed (where *localhost* will make the algorithm be computed on the local machine) or to let the system automatically choose one by selecting the *auto* entry. The currently

used strategy in the *auto* mode is round robin: on each invocation the host in the list next to the previously used one is chosen.

The *Reload hosts* button, when pressed, brings up the hosts list checking window and starts the hosts checking procedure described in Section 4.2.1. The *Proxy* button, when pressed, brings up the Grid Proxy Initialization window described earlier.

Once the Grid node is chosen, be that local or remote, the task may be started by pressing the *Start* button and stopped by pressing the *Stop* button. As mentioned in Section 4.2.1 in Weka4WS, unlike in Weka, a task is carried out in a thread of its own thus allowing to run multiple tasks in parallel. In Figure 4.5, in the lower right corner of the window, the number of running tasks is displayed. The list of started tasks is displayed in the *Result list* pane, just below the Control panel.

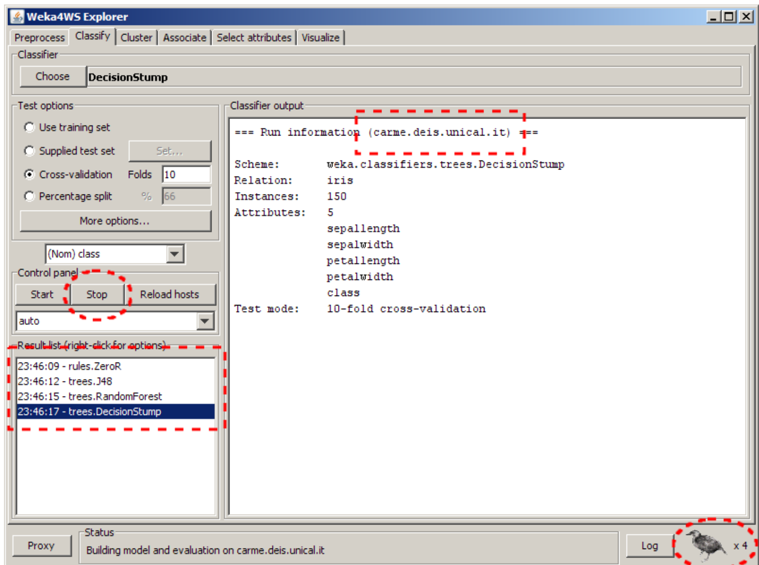

Fig. 4.5. Weka4WS Explorer: multiple tasks execution

The *Output panel*, at the right side of the window, shows the run information and results (as soon as they are known) of the task currently selected in the Result list; at the top of the Output Panel, as highlighted in Figure 4.5, is shown the host name where the task is being computed.

It is possible to follow the remote computations in their very single steps as well as to know their execution times through the log window, which is shown by pressing the *Log* button in the lower right corner of the Window, as highlighted in Figure 4.6.

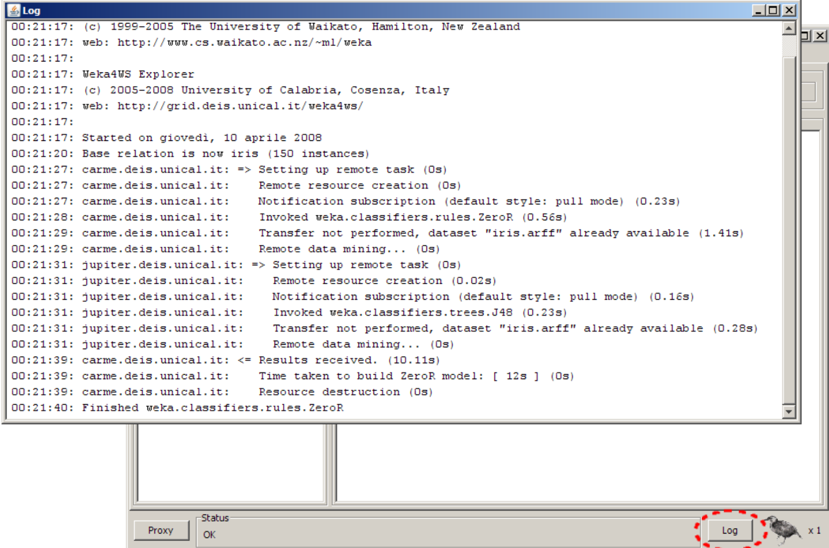

Fig. 4.6. Weka4WS Explorer: detailed log

4.3.2 Knowledge Flow

Knowledge Flow is the component of Weka which allows to compose workflows for processing and analyzing data. A workflow can be done by selecting components from a tool bar, placing them on a layout canvas and connecting them together: each component of the workflow is demanded to carry out a specific step of the data mining process. A more detailed description of Knowledge Flow may be found in [89].

We extended the Weka Knoledge Flow to allow the execution of distributed data mining workflow on Grids by adding annotations into the Knowledge Flow. Through annotations a user can specify how the workflow nodes can be mapped onto Grid nodes.

In Figure 4.7 some of the changes introduced in Weka4WS are highlighted: in the upper right corner three buttons have been added whose purpose is, from top to bottom, to start all the tasks at the same time, to stop all the tasks and to reload the hosts list, as seen in Section 4.2.1. A button named *Proxy* has been added in the lower left corner of the window which when pressed, just like in the Explorer component, brings up the Grid Proxy Initialization window described earlier. The labels under each component associated to an algorithm indicate, during the computation, the hosts where that algorithm is being computed.

The choice of the location where to run a certain algorithm is made into the configuration panel of each algorithm, accessible right clicking on the given algorithm and choosing *Configure*, as shown in Figure 4.8: within the highlighted area it can be seen the part added in Weka4WS which, as previously

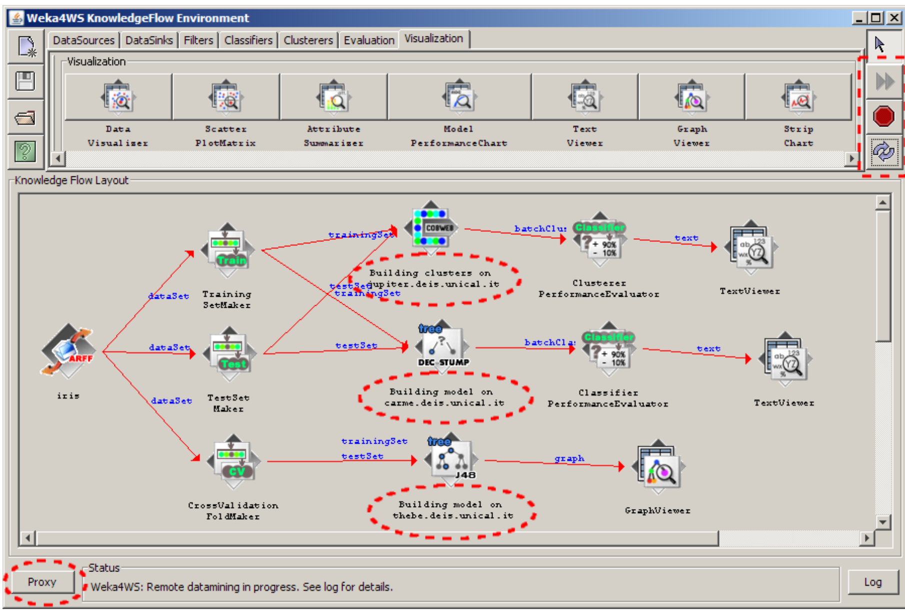

Fig. 4.7. Weka4WS Knowledge Flow: simple flow and Control Panel

seen for the Control Panel of the Explorer component, consists of a drop down menu containing the available locations where the selected algorithm can be executed.

Although the algorithms and their performance evaluators are represented by two separate nodes, the model building and its evaluation are actually performed in conjunction at the computing node when the chosen location is not local.

For complex configurations of workflows the sub-flow grouping feature of Knowledge Flow turns out to be useful in order to easily and quickly set the remote hosts for the execution of the algorithms. Through this feature it is possible to group together a set of components of the flow which will then be represented graphically by only one component, the black-to-gray faded one shown in Figure 4.9: right clicking on this component it is possible to either set to *auto* all the computing locations of the algorithms belonging to the group, or choosing the specific location of each algorithm by accessing the relative configuration listed in the menu.

The computations may be started, as shown in Figure 4.10, either by selecting the *Start loading* entry in the right-click context menu of each loader component of the flow (just like usually done in the conventional Weka Knowledge Flow) or by pressing the *Start all executions* button in the right-top corner of the window (which is more convenient in flows with multiple loader components).

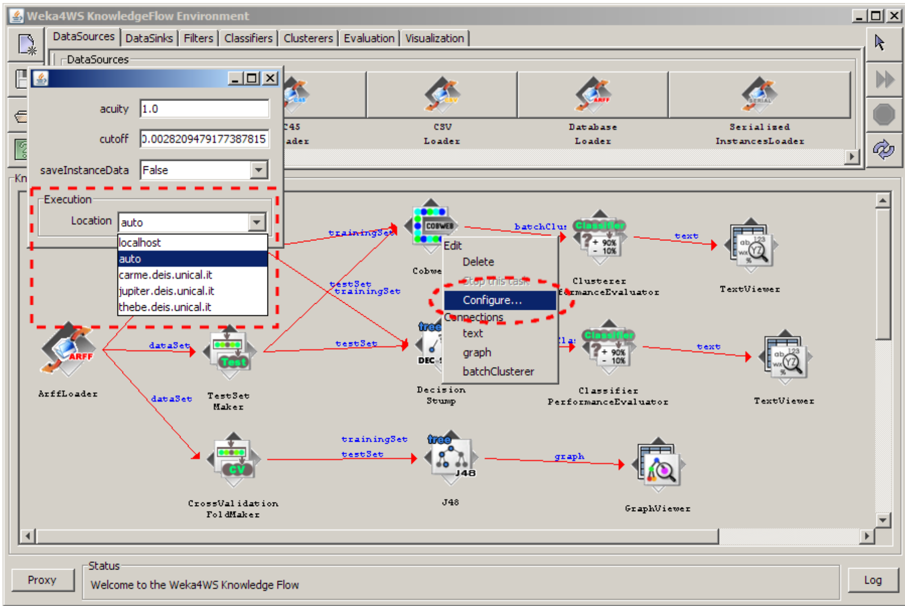

Fig. 4.8. Weka4WS Knowledge Flow: selection of the remote host

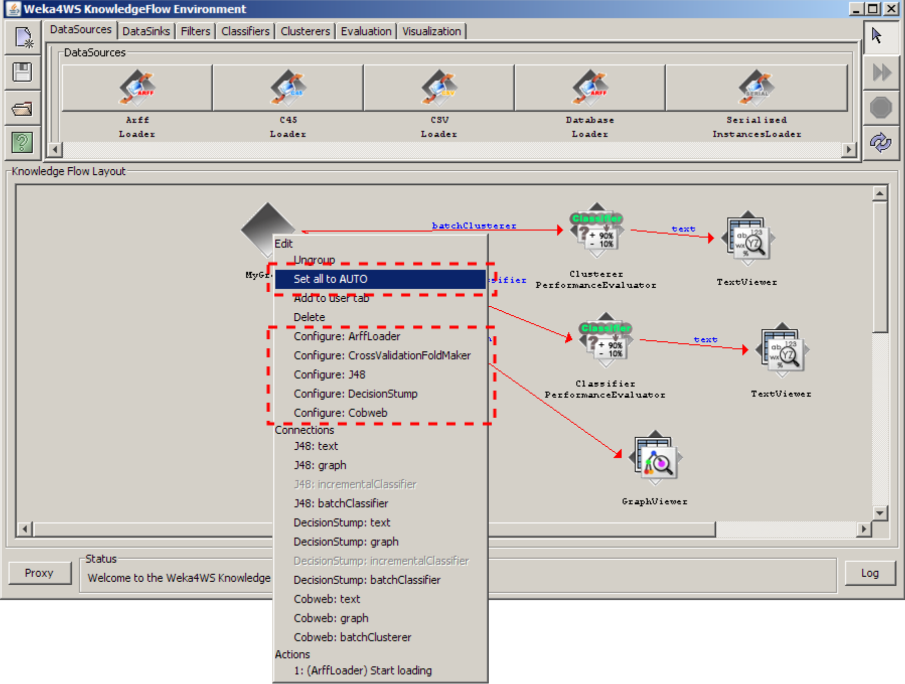

Fig. 4.9. Weka4WS Knowledge Flow: computing locations selection

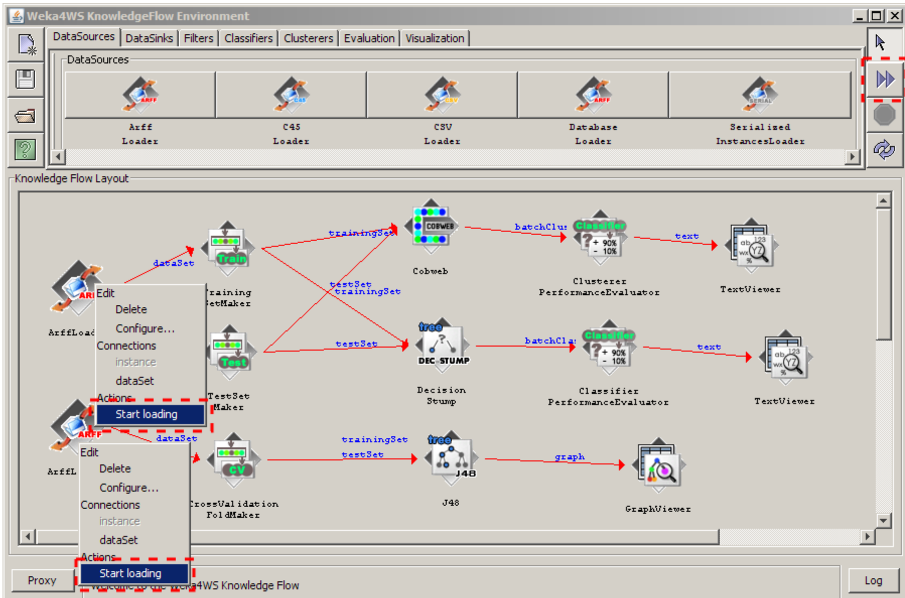

Fig. 4.10. Weka4WS Knowledge Flow: computations start

As for the Explorer component, pressing the Log button in the lower-right corner it is possible to follow the computations in their very single steps as well as to know their execution times.

4.4 How the system works

In this section we are going to see in details, through an invocation example, all the steps and mechanisms involved in the execution of one single data mining task on the Grid; these steps are the same regardless on whether the task is invoked from the Explorer or the Knowledge Flow interface.

In this example we are assuming that the client is requesting the execution of a classification task on a dataset which is present on the user node, but not on the computing node. This is to be considered a worst scenario because in many cases the dataset to be mined is already available (or, more specifically, replicated) on the Grid node where the task is to be submitted.

When a remote data mining task is started an unambiguous identification number, called `taskID`, is generated: this number is associated to that particular task and is used when a `stopTask` operation is invoked to identify the task among all the others running at the computing node.

The whole invocation process may be divided in the 8 steps shown in Figure 4.11:

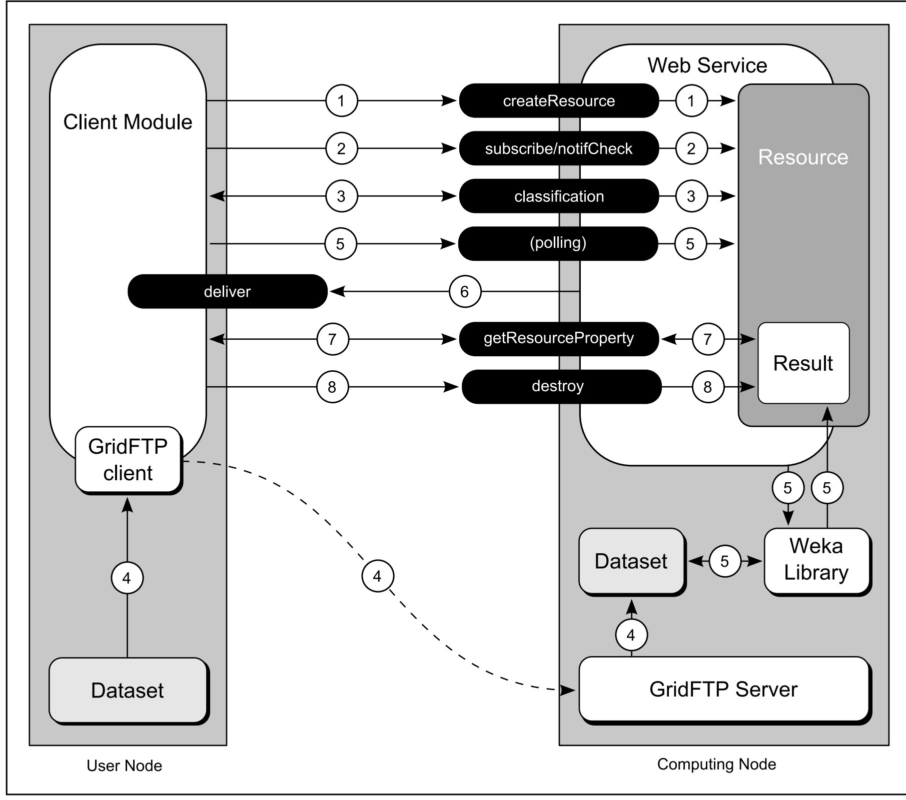

Fig. 4.11. Weka4WS: task invocation

1. **Resource creation**: the *createResource* operation is invoked to create a new resource that will maintain its state throughout all the subsequent invocations of the Web Service until its destruction. The state is stored as properties of the resource; more specifically a `Result` property, detailed in Table 4.4, is used to store the results of the data mining task.

Table 4.4. Result resource property composition

Field name	Type	Subfield names
`model`	ModelResult	`model`, `models`
`evaluation`	EvalResult	`summary`, `classDetails`, `matrix`
`visualization`	VisResult	`predInstances`, `predictions`, `plotShape`, `plotSize`
`exception`	Weka4WSException	`thrown`, `message`
`stopped`	boolean	
`ready`	boolean	

The first three fields of the property stores the inferred model and/or models, the evaluation outcomes and additional information about visualization and prediction. The last three fields, `exception`, `stopped` and `ready` are used only when certain circumstances arise:

- if during the computation phase something goes wrong and an exception is thrown then the field `exception` is set accordingly: its boolean parameter `thrown` is set to true and in its string parameter, `message`, is stored the generated exception message;
- if during the computation a request of termination is received through the `stopTask` operation then the boolean field `stopped` is set to true;
- after the end of the computation the results are put into the Result property and the field `ready` is set to true: this field is used by the client when it's unable to receive notifications, like in the scenario depicted in Figure 4.2, to periodically check whether the results have been computed;

After the resource has been created the Web Service returns the endpoint reference (EPR) of the created resource. The EPR is unique within the Web Service, and differentiates this resource from all the other resources in that service. Subsequent requests from the Client Module will be directed to the resource identified by that EPR;

2. **Notification subscription and notifications check**: the *subscribe* operation is invoked in order to be notified about changes that will occur to the `Result` resource property. Upon this property value change (that is upon the conclusion of the data mining task) the Client Module will be notified of it. Just after the *subscribe* operation the *notifCheck* operation is invoked to request the immediate send of a dummy notification to check whether the client is able to receive notifications, as described in Section 4.2: when and if the user node will receive this dummy notification it will switch to the "push-style" mode, otherwise will persist in the "pull-style" one;
3. **Task submission**: the *classification* operation is invoked in order to ask for the execution of the classification task. This operation requires the 7 parameters shown in Table 4.2, among which the `taskID` previously mentioned. The operation returns the `Response` object, detailed in Table 4.5.

Table 4.5. Response composition

Field name	Type
`datasetFound`	boolean
`testsetFound`	boolean
`dirPath`	string
`exception`	Weka4WSException

If a copy of the dataset is not already available at the computing node, then the field `datasetFound` is set to false and the `dirPath` field is set to the URL where the dataset has to be uploaded; similarly, when a validation is required on a test set which is different from the dataset and the test set is not already available at the computing node, the `testsetFound` field is set to false. The URL where the test set has to be uploaded is the same as for the dataset. If during the invocation phase something goes wrong and an exception is thrown then the field `exception` is set accordingly: its boolean parameter `thrown` is set to true and in its string parameter, `message`, is stored the generated exception message;

4. **File transfer**: since in this example we assumed that the dataset was not already available at the computing node, the Client Module needs to transfer it to the computing node. To that end, a Java GridFTP client [2] is instantiated and initialized to interoperate with the computing node GridFTP server: the dataset (or test set) is then transferred to the computing node machine and saved in the directory whose path was specified in the `dirPath` field contained in the `Response` object returned by the classification operation;
5. **Data mining**: the classification analysis is started by the Web Service through the invocation of the appropriate Java class in the Weka library. The results of the computation are stored in the `Result` property of the resource created on Step 1;
6. **Notification reception**: as soon as the `Result` property is changed a notification is sent to the Client Module by invoking its implicit `deliver` operation. This mechanism allows the asynchronous delivery of the execution results whenever they are generated. In those cases where the client is unable to receive notifications the client will be periodically checking the results for readiness through the value of the Result's field `ready` (see Table 4.4);
7. **Results retrieving**: the Client Module invokes the operation `getResourceProperty` in order to retrieve the `Result` property containing the results of the computation;
8. **Resource destruction**: The Client Module invokes the `destroy` operation, which eliminates the resource created on Step 1.

4.5 Supporting data-parallel workflows

The workflow presented in the previous section employs *simple parallelism* [68], a form of parallelism which runs multiple independent tasks in parallel on different processors, available on a single parallel machine or on a set of machines connected through a network like the Grid.

Another form of parallelism that could be effectively exploited in data mining workflows is *data parallelism* [68], where a large data set is split into

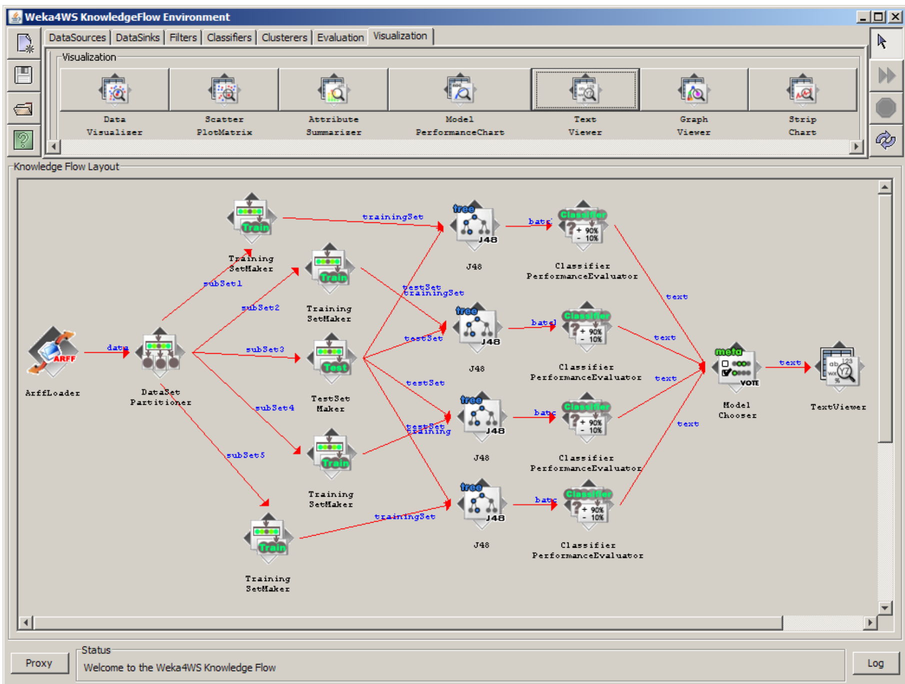

Fig. 4.12. An example of data-parallel workflow using the DataSetPartitioner and ModelChooser components.

smaller chunks, each chunk is processed in parallel, and the results of each processing are then combined to produce a single result.

Both parallelism forms aim to achieve execution time speedup, and a better utilization of the computing resources, but while simple parallelism focuses on running multiple tasks in parallel so that the execution time corresponds to the slowest task, data parallelism focuses on reducing the execution time of a single task by splitting it into subtasks, each one operating on a subset of the original data.

The data-parallel approach is widely employed in distributed data mining as it allows to process very large datasets that could not be analyzed on a single machine due to memory limitations and/or computing time constraints. An example of data-parallel application is *distributed classification*: the dataset is partitioned into different subsets that are analyzed in parallel by multiple instances of a given classification algorithm; the resulting “base classifiers” are then used to obtain a global model through various selection and combining techniques [11].

To support the data parallelism paradigm, we developed a new component in the KnowledgeFlow, called *DataSetPartitioner*. A DataSetPartitioner component receives one dataset in input, divides it into a number of parti-

tions equal to the number of its outgoing arcs, and assigns each partition to one workflow node representing a data mining algorithm (for example, a classification algorithm). In case the data mining algorithm is annotated to be executed on a remote computing node, the subset assigned to it will be transferred to that computing node as described in Section 4.4.

In the DataSetPartitioner the output partitions have by default the same number of instances: however the size of each partition, expressed as a percentage of the original dataset size, may be changed by the user through the configuration panel of the component.

We also developed a *ModelChooser* component that receives a set of base classifiers, resulting from the classification tasks performed on the different partitions, and returns the "best" model based on the chosen criterion (e.g., the lowest error rate).

An example of workflow using these two components is the one shown in Figure 4.12: it includes, from left to right, an ArffLoader node, used to load a dataset from file. A DataSetPartitioner node, used to generate five subsets of equal size of the incoming dataset. Each subset is sent to a different evaluation component, precisely to four TrainingSetMakers and one TestSetMaker.

Each one of the four TrainingSetMakers marks the incoming dataset partition as training set and sends it to one of the four J48 Classifiers, while the only TestSetMaker marks the incoming dataset partition as test set and sends it to all the four J48 Classifiers. The four J48 Classifiers are then connected with four ClassifierPerformanceEvaluators which are all connected with one ModelChooser component. The ModelChooser chooses the best model as explained earlier and sends such model to the TextViewer component for its visualization.

Figure 4.13 shows how to configure the DataSetPartitioner component to customize the size of the various partitions of the dataset.

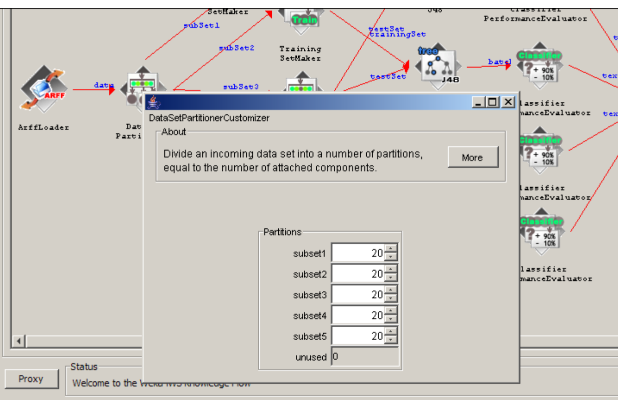

Fig. 4.13. Configuration panel of the DataSetPartitioner used in the workflow of Figure 4.12.

4.6 Conclusions

We presented a Grid-enabled version of Weka, called Weka4WS, which provides an extension of the KnowledgeFlow environment to support distributed execution of data mining workflows on a Grid. Weka4WS uses a service-oriented approach in which all the Weka data mining algorithms are wrapped as Web Services and deployed on Grid nodes; users can compose and invoke those services in a transparent way by defining data mining workflows as in the original Weka KnowledgeFlow. This approach allows to define simple parallel data mining applications in a easy and effective way.

The Weka4WS code is available [1] for research and application purposes, and can be used and modified under the terms of the GNU General Public License [2] as published by the Free Software Foundation [3].

[1] http://grid.deis.unical.it/weka4ws/
[2] http://www.gnu.org/licenses/gpl.txt
[3] http://www.fsf.org/

5

Knowledge Grid

The Knowledge Grid [10] supports the implementation of data mining applications on a Grid by providing mechanisms and services for publishing and searching the needed resources (data sources, data mining tools, etc.), creating and executing distributed data mining processes, and managing data mining results.

The Knowledge Grid services are organized in two hierarchical layers: the *core layer* and the *high-level layer*, as shown in Figure 5.1. The design idea is that client applications directly interact with high-level services that, in order to satisfy client requests, invoke suitable operations exported by the core-level services. In turn, core-level services perform their operations by invoking basic services provided by available Grid environments running on the specific host, as well as by interacting with other core-level services.

The high-level layer includes the following services:

- *Data Access Service* (*DAS*), which provides operations for publishing, searching and downloading data to be mined (`publishData`, `searchData`, and `downloadData` operations);
- *Tools and Algorithms Access Service* (*TAAS*), which is responsible for publishing, searching and downloading tools and algorithms for data extraction, pre-processing and mining (`publishTool`, `searchTool`, and `downloadTool` operations);
- *Execution Plan Management Service* (*EPMS*), which receives a conceptual model of the data mining task through the `submitKApplication` operation, translates it into an abstract execution plan, and passes it to the RAEMS service (see below).
- *Results Presentation Service* (*RPS*), which allows to retrieve the results (i.e., the inferred models) of previous data mining computations through the `getResults` operation.

The core-level layer includes two services:

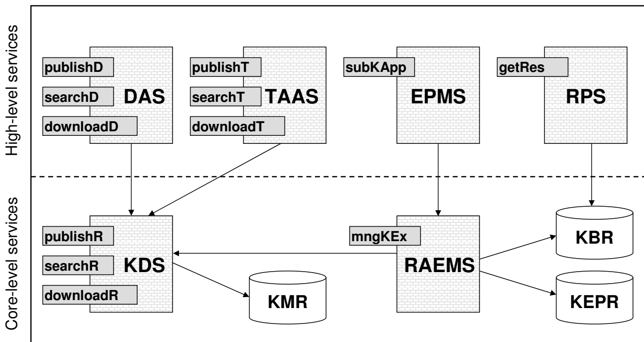

Fig. 5.1. Knowledge Grid layered services.

- *Knowledge Directory Service* (*KDS*), which is responsible for managing metadata about the Knowledge Grid resources (data, tools and algorithms). It provides three operations (`publishResource`, `searchResource`, and `downloadResource`) to publish, search and download resource metadata, which are stored in a *Knowledge Metadata Repository* (*KMR*) as XML documents (details about structure and use of metadata in the Knowledge Grid can be found in [61]).
- *Resource Allocation and Execution Management Service* (*RAEMS*), which starting from an abstract execution plan (received through the `manageKApplication` operation) generates a concrete execution plan and manages its execution. Generated execution plans are stored in a *Knowledge Execution Plan Repository* (*KEPR*), while the results are stored in a *Knowledge Base Repository* (*KBR*).

All the Knowledge Grid services have been implemented as Web Services that comply with the Web Services Resource Framework (WSRF) family of standards, as described in a previous work [15]. In particular, we used the WSRF library provided by Globus Toolkit 4 [27], as well as some basic Grid services (e.g., reliable file transfer, authentication and authorization) provided by the same toolkit.

Within the Knowledge Grid project, a visual software environment named DIS3GNO has recently been implemented to allow a user to: *i*) compose a distributed data mining workflow; *ii*) execute the workflow onto the Knowledge Grid; *iii*) visualize the results of the data mining task. DIS3GNO performs the mapping of the user-defined workflow to the conceptual model and submits it to the Knowledge Grid services, managing the overall computation in a way that is transparent to a user.

5.1 The DIS3GNO System

DIS3GNO represents the user interface for two main Knowledge Grid functionalities:

- *Metadata management*. DIS3GNO provides an interface to publish and search metadata about data and tools, through the interaction with the DAS and TAAS services.
- *Execution management*. DIS3GNO provides an environment to design and execute distributed data mining applications as service-oriented workflows, through the interaction with the EPMS service.

The DIS3GNO GUI, depicted in Figure 5.2, has been designed to reflect this two-fold functionality. In particular, it provides a panel (on the left) devoted to search resource metadata, and a panel (on the right) to compose and execute data mining workflows.

In the top-left corner of the window there is a menu used for opening, saving and creating new workflows, viewing and modifying some program settings and viewing the previously computed results present in the local file system. Under the menu bar there is a toolbar containing some buttons for the execution control (starting/stopping the execution and resetting the nodes statuses) and other for the workflow editing (creation of nodes representing datasets, tools or viewers, creation of edges, selection of multiple nodes and deletion of nodes or edges).

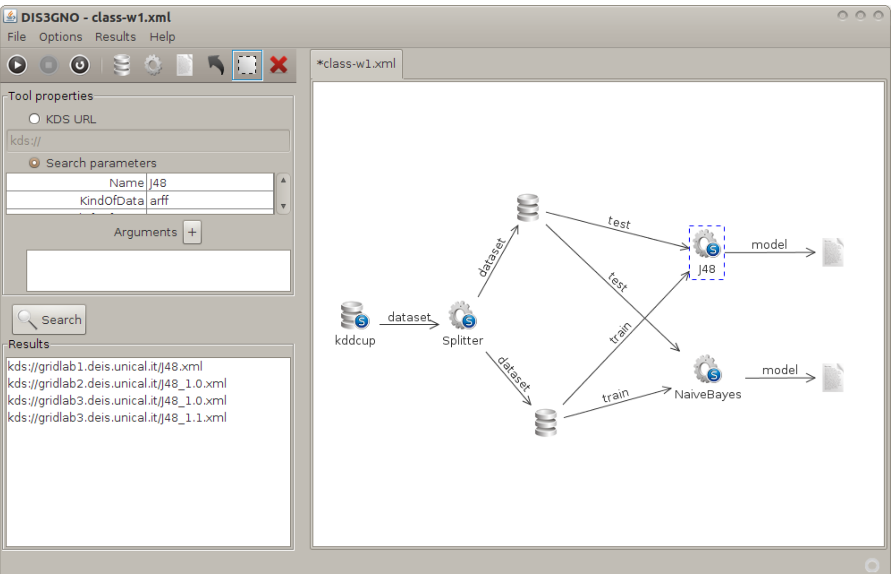

Fig. 5.2. A screenshot of the DIS3GNO GUI.

5.1.1 Workflow Representation

In DIS3GNO a workflow is represented as a directed acyclic graph whose nodes represent resources and whose edges represent the dependencies among the resources.

The types of resources that can be present in a data mining workflow (graphically depicted by the icons in Figure 5.3) are:

- *Dataset*, representing a dataset;
- *Tool*, representing a tool to perform any kind of operation which may be applied to a dataset (data mining, filtering, splitting, etc.) or to a model (e.g., voting operations);
- *Model*, represents a knowledge model (e.g., a decision tree, a set of association rules, etc.), that is the result produced by a data mining tool.

Fig. 5.3. Nodes types.

Each node contains a description of a resource as a set of properties which provide information about its features and actual use. This description may be full or partial: in other words, it is both possible to specify a particular resource and its location in the Grid, or just a few of its properties, leaving to the system the task to find a resource matching the specified characteristics and its location. In the former case we will refer to the resource as *concrete*, in the latter one as *abstract*.

For example, in the case of a data mining tool, one could be interested in any algorithm, located in any node of the Grid, provided it is a classification algorithm able to handle "arff" files, or could want specifically the algorithm named *NaiveBayes* located in a specified host. Once the workflow will be executed, the Knowledge Grid middleware will find a concrete resource matching the metadata, whether they are completely or partially specified. Clearly only dataset and tool nodes can be either concrete or abstract, the model node can't be abstract as it represents the result of a computation. The model node has only one property, the *location*, which if left empty will be implicitly set to the same location of the tool node in input.

When a particular resource property is entered, a label is attached below to the corresponding icon, as shown in the example in Figure 5.4. The property chosen as the label is the one considered most representative for the resource,

i.e. the *Name* for the dataset and tool nodes and the *Location* for the model node.

Fig. 5.4. Nodes labels.

In order to ease the workflow composition and to allow a user to monitor its execution, each resource icon bears a symbol representing the status in which the corresponding resource is at a given time. When the resource status changes, as consequence of the occurrence of certain events, its status symbol changes accordingly. The resource statuses can be divided in two categories: the *composition-time* and the *run-time* statuses.

The *composition-time* statuses (shown in Table 5.1), useful during the workflow composition phase, are:

1. *No information provided* = no parameter has been specified in the resource properties;
2. *Abstract resource* = the resource is defined through constraints about its features, but it is not known a priori; the *S* in the icon stands for *search*, meaning that the resource has to be searched in the Grid;
3. *Concrete resource* = the resource is specifically defined through its KDS URL; the *K* in the icon stands for *KDS URL*;
4. *Location set* = a location for the model has been specifically set (this status is pertinent to model nodes only);

The *run-time* statuses (shown in Table 5.2), useful during the workflow execution phase, are:

Symbol	Meaning
	No information provided
	Abstract resource
	Concrete resource
	Location set

Table 5.1. Nodes composition-time statuses.

Symbol	Meaning
	Matching resource found
	Running
	Resource not found
	Execution failed
	Task completed successfully

Table 5.2. Nodes run-time statuses.

1. *Matching resource found* = a concrete resource has been found matching the metadata;
2. *Running* = the resource is being executed/managed.
3. *Resource not found* = the system hasn't found a resource matching the metadata;
4. *Execution failed* = some error has occurred during the management of the corresponding resource;
5. *Task completed successfully* = the corresponding resource has successfully fulfilled its task;

Each resource may be in one of these run-time statuses only in a specific phase of the workflow execution: i.e. status 1 and 2 only during the execution, status 3 and 4 during or after the execution, status 5 only after the execution.

The nodes may be connected with each other through edges, establishing specific dependency relationships among them. All the possible connections are show in Table 5.3; those not present in Table 5.3 are not allowed and the graphical user interface ensures a user is prevented to create them.

First resource	Second resource	Label	Meaning	Graphical representation
dataset	dataset	`transfer`	*Explicit file transfer*	transfer
dataset	tool	`dataset, train, test`	*Type of input for a tool node*	?
tool	dataset	`dataset`	*Dataset produced by a tool*	dataset
tool	model	`model`	*Model produced by a DM algorithm*	model
model	tool	`model`	*Model received by a tool*	model
model	model	`transfer`	*Explicit transfer of a model*	transfer

Table 5.3. Nodes connections.

When an edge is being created between two nodes, a label is automatically attached to it representing the kind of relationship between the two nodes. In most of the cases this relationship is strict but in one case (dataset-tool connection) requires further input from a user to be specified.

The possible edge labels are:

- *dataset*: indicates that the input or output of a tool node is a dataset;
- *train*: indicates that the input of a tool node has to be considered a training set;

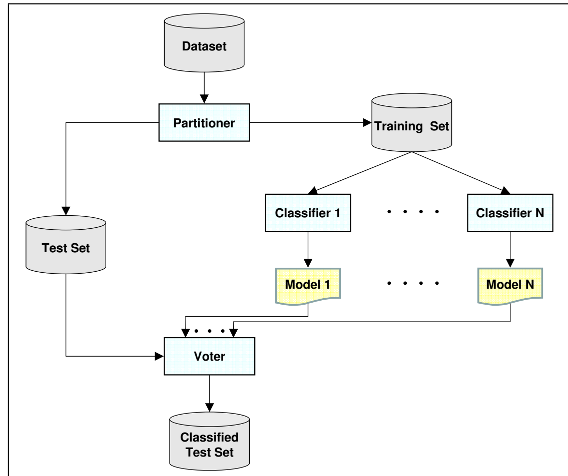

Fig. 5.5. Logical schema of an ensemble learning application.

- *test*: indicates that the input of a tool node has to be considered a test set;
- *transfer*: indicates an explicit transfer of a dataset, or a result of a computation, from one Grid node to another;
- *model*: indicates a result of a computation of a data mining algorithm.

5.1.2 Workflow Composition

To outline the main functionalities of DIS3GNO, we briefly describe how it is used to compose and run a data mining workflow. By exploiting the DIS3GNO GUI, a user can compose a workflow by selecting from the toolbar the type of resource to be inserted in the workflow (a *dataset*, a *tool* or a *model* node), dragging it into the workflow composition panel. Such operation can be repeated as many times as needed to insert all the required application nodes. Then, she/he has to insert suitable edges by setting, for each one, the specific dependency relationship between the nodes (as described in Section 5.1.1 and summarized in Table 5.3). Typically, most nodes in a workflow represent abstract resources. In other terms, a user initially concentrates on the application logic, without focusing on the actual datasets or data mining tool to be used.

Let us suppose that a user wants to compose and execute the ensemble-learning application depicted in Figure 5.5. In contrast to ordinary machine learning approaches which try to learn one model from training data, ensemble methods build a set of models and combine them to obtain the final model [92]. In a classification scenario, an ensemble method constructs a set of *base classifiers* from training data and performs classification by taking a vote on the predictions made by each classifier.

As shown in Figure 5.5, the input dataset is split, using a partitioner tool, into two parts, namely a *training set* and a *test set*. The training set is given in input to N classifiers which run in parallel to build N independent classification models from it. Then, a voter tool performs an ensemble classification by assigning to each instance of the test set the class predicted by the majority of the N models generated at the previous step.

Using DIS3GNO, the ensemble learning application can be designed as follows. First, a user chooses the input dataset (Figure 5.6). To do that, she/he selects from the toolbar the *dataset* icon and drags it into the workflow composition panel. In order to associate such an icon to a concrete resource, the user specifies name and format of the desired dataset into the *Search parameters* panel. The search is started by pressing the *Search* button; on completion, the system lists the set of datasets matching the search criteria (left-bottom part of the GUI).

After having selected one of the datasets listed, the URL of such dataset is associated with the dataset icon (Figure 5.7a). This also changes the dataset icon mark from S (resource still to be selected) to K (resource identified by a KDS URL). In the same way, the user inserts a *tool* node, which is associated with the KDS URL of the desired partitioner (Figure 5.7b). Then, the user must specify the relationship between the two nodes. To do that, an edge linking the *dataset* and the *tool* icons is created and labelled appropriately (Figure 5.7c).

According to the ensemble-learning scenario, two dataset icons representing the output of the partitioner are added to the workflow (Figure 5.8a).

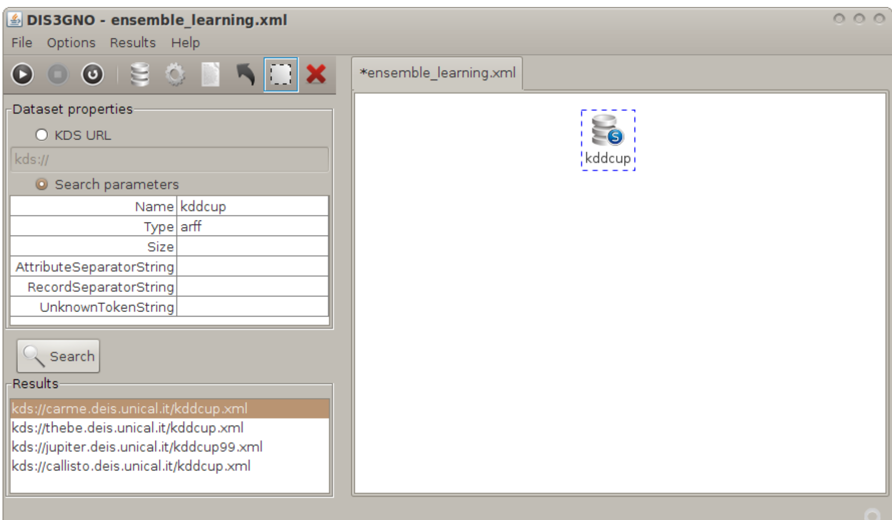

Fig. 5.6. Insertion of the input dataset icon with specification of its properties and search for matching resources.

Then, the user proceeds by adding the classifiers that are required to build the base models. We assume that the user wants to use four classifiers (ConjunctiveRule, NaiveBayes, RandomForest and J48) specified as abstract resources (see Section 5.1.1). For example, Figure 5.8b shows the insertion of the first classifier (ConjunctiveRule) and the specification of its properties (name of software and type of data supported). The classifier icon is marked with an S to remind that the corresponding resource will be searched and made concrete at runtime. Similarly, the other three classifiers are added, and an edge between the training set and the four classifiers is created (Figure 5.8c).

Figure 5.9 shows the complete workflow. It includes: (i) a *model* node connected to each classifier; (ii) a *tool* node representing a voter that takes in input the test set and the four base models; (iii) the output dataset obtained as output of the voter tool.

The workflow can be submitted to the EPMS service by pressing the *Run* button in the toolbar. As a first action, if a user credentials are not available or have expired, a Grid Proxy Initialization window is loaded. After that,

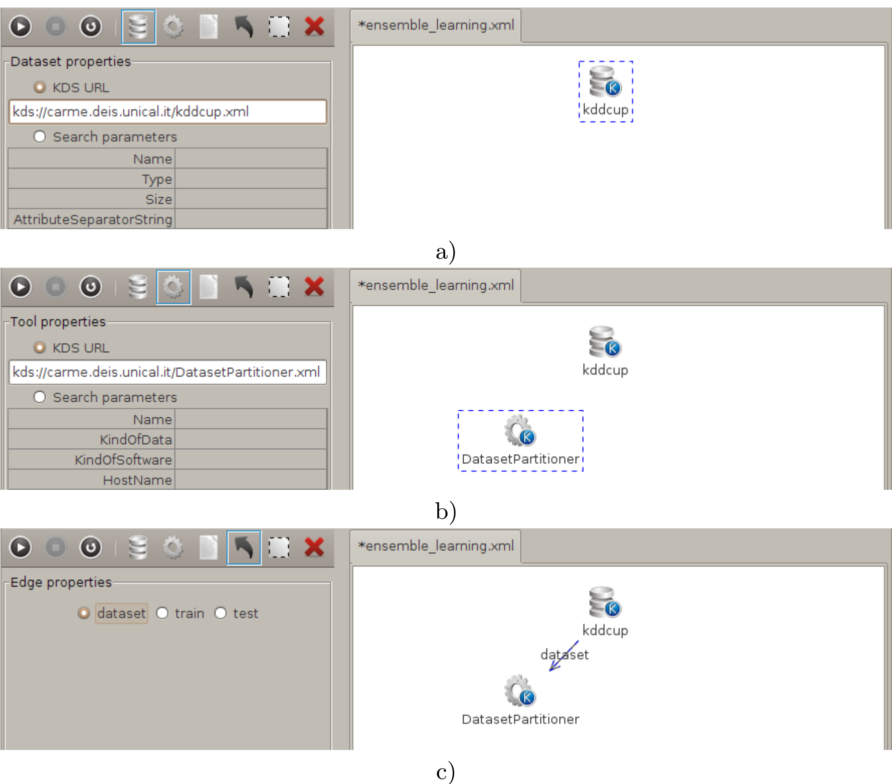

Fig. 5.7. a) Selection of the input dataset; b) Insertion and selection of a partitioner tool; c) Insertion of a labelled edge between dataset and partitioner.

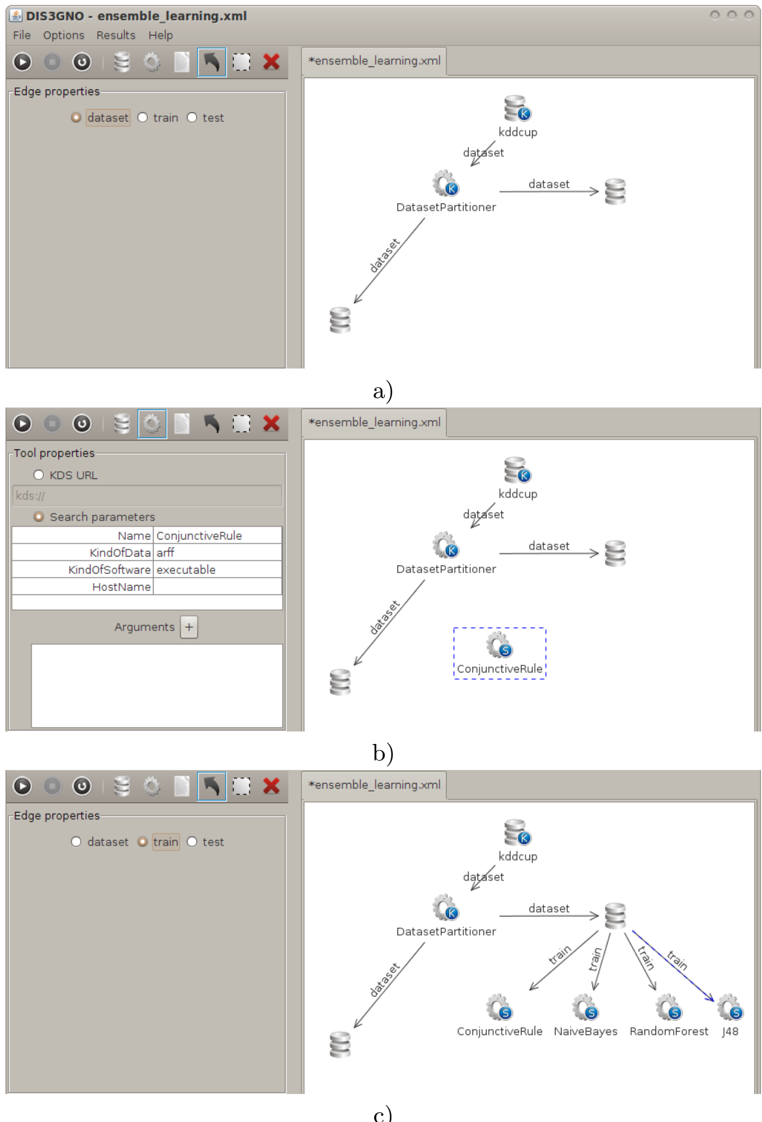

Fig. 5.8. a) Insertion of two dataset icons representing the partitioner output; b) Insertion and specification of an abstract tool resource; c) Workflow after insertion and specification of all the classifiers and associated input edges.

the workflow execution actually starts and proceeds as detailed in the next section.

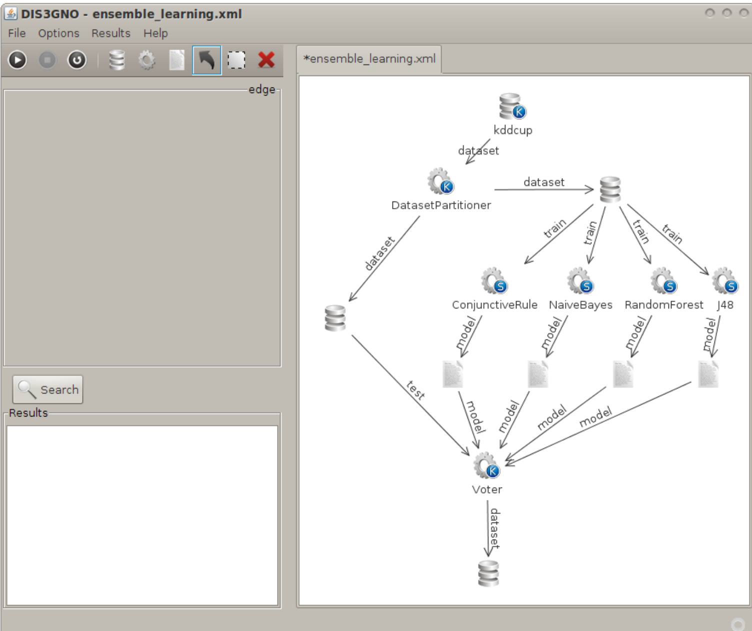

Fig. 5.9. The complete workflow including the base models, a voter tool, and the output dataset.

5.1.3 Execution Management

Starting from the data mining workflow designed by a user, DIS3GNO generates an XML representation of the data mining application referred to as *conceptual model*. DIS3GNO passes the *conceptual model* to a given EPMS, which is in charge of transforming it into an *abstract execution plan* for subsequent processing by the RAEMS. The RAEMS receives the abstract execution plan and creates a *concrete execution plan*. In order to carry out such a task, the RAEMS needs to evaluate and resolve a set of resources and services, by contacting the KDS and choosing the most appropriate ones. As soon as the RAEMS has built the concrete execution plan, it is in charge of coordinating its execution by invoking the coordinated execution of services corresponding to the nodes of the concrete execution plan. The status of the computation is notified to the EPMS, which in turn forwards the notifications to the DIS3GNO system for visualization.

Figure 5.10 describes the interactions that occur when an invocation of the EPMS is performed. In particular, the figure outlines the sequence of invocations of others services, and the interchanges with them when a data mining workflow is submitted for allocation and execution. To this purpose, the EPMS exposes the `submitKApplication` operation, through which it receives a conceptual model of the application to be executed (step 1).

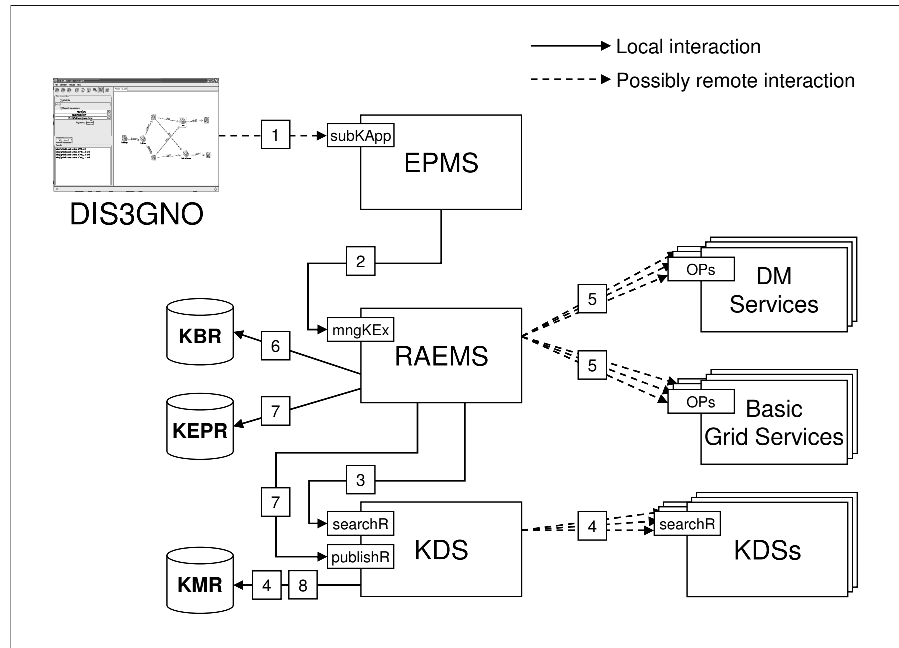

Fig. 5.10. Execution management. The sequence of operation invocations is showed for all services involved in the mapping and execution of a data mining workflow.

The basic role of the EPMS is to transform the conceptual model into an abstract execution plan for subsequent processing by the RAEMS. An abstract execution plan is a more formal representation of the structure of the application. Generally, it does not contain information on the physical Grid resources and services to be used, but rather constraints about them.

The RAEMS exports the `manageKExecution` operation, which is invoked by the EPMS and receives the abstract execution plan (step 2). First of all, the RAEMS queries the local KDS (through the `searchResource` operation) to obtain information about the resources needed to instantiate the abstract execution plan (step 3). Note that the KDS performs the searching both accessing the local Knowledge Metadata Repository (KMR) and querying all the reachable remote KDSs (step 4). To reach as many remote KDSs as needed, an unstructured peer-to-peer overlay is built among Knowledge Grid nodes. To this end, each node possesses a configurable set of neighboring nodes to which forward a query.

After the instantiated execution plan is obtained, the RAEMS coordinates the actual execution of the overall computation. To this purpose, the RAEMS invokes the appropriate data mining services (DM Services) and basic Grid services (e.g., file transfer services), as specified by the instantiated execution plan (step 5). The RAEMS stores the results of the computation into

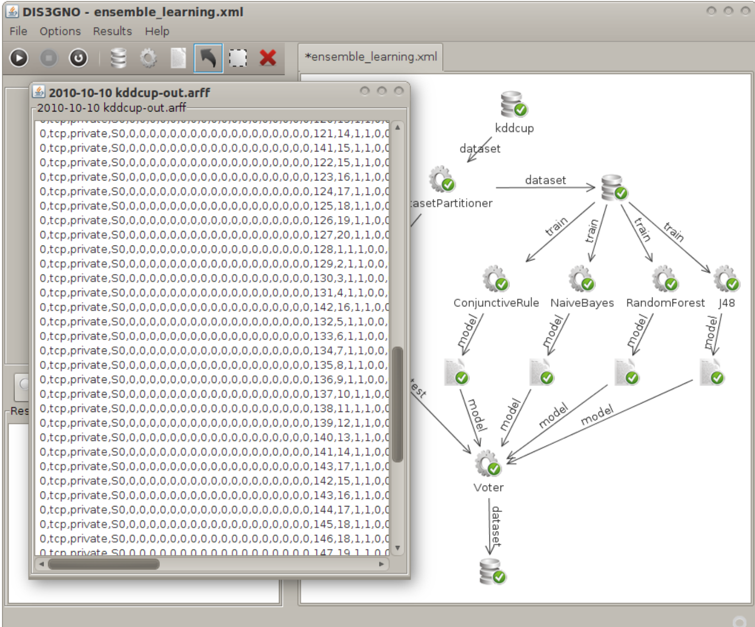

Fig. 5.11. The final visualization of DIS3GNO after the completion of the application running.

the Knowledge Base Repository (KBR) (step 6), while the execution plan is stored into the Knowledge Execution Plan Repository (KEPR) (step 7). To make available the results stored in the KBR, it is necessary to publish results metadata into the KMR. To this end, the RAEMS invokes the `publishResource` operation of the local KDS (steps 7 and 8).

Figure 5.11 shows the final screenshot of the DIS3GNO interface when the execution of the data mining workflow has been completed and the final classification result has been created and showed in an ad hoc window.

5.2 Conclusions

The DIS3GNO system described in this chapter provides a set of visual programming facilities to design and execute distributed data mining workflows in Grids according to the service-oriented model. The DIS3GNO GUI operates as an intermediary between an end user and the Knowledge Grid allowing the high-level development of service-oriented high-performance distributed KDD applications. All the Knowledge Grid services for metadata access and execution management are accessed transparently by DIS3GNO, thus allowing the domain experts to compose and run complex data mining applications without worrying about the underlying infrastructure details.

6

Workflow level fault tolerance

Due to the heterogeneous and distributed nature of Grid systems, faults inevitably happen. Unfortunately up to now, most of the existing Grid workflow systems still cannot deliver the quality, robustness and reliability that is needed for widespread acceptance as tools used on a day-to-day basis for scientists from a multitude of scientific fields [70]. The scientists typically want to use the grid to compute solutions for complex problems, potentially utilizing numerous resources for workflows that can run for an extended period of time. With a system that has a low tolerance for faults, the users will regularly be confronted with a situation that makes them lose days or even weeks of valuable computation time because the system could not recover from a fault that happened before the successful completion of their workflow applications. This is, of course, intolerable for anyone trying to effectively use the Grid, and makes scientists accept a slower solution that only uses their own computing resources because of a higher reliability and controllability of these systems.

The enactment of a scientific workflow typically requires the mapping of tasks onto third-party resources which are geographically distributed. This execution of tasks across multiple autonomous administrative domains makes the mapping process highly unreliable: faults may arise due to hardware and network failures, and software or application errors. A workflow environment can generally consist of multiple tiers – such as resource management, middleware-supported data distribution and workflow enactors, and user front ends (portals). Each of these tiers may support their own fault tolerance capabilities. For instance, a resource manager may detect a local fault and invoke an action without exposing the fault to the user portal. Our focus in this work is on fault tolerance capabilities that must be supported at the workflow level, as a user executing a workflow often has limited control of an external resource management system.

The recovery technique proposed in this chapter consists in systematically analyzing known faults, and subsequently applying the most suitable recovery action to them. A fault taxonomy for scientific workflows has been proposed and a methodology for the analysis has been developed: fault detection, fault

identification and finally fault correction. We identified fault detection mechanisms in the literature, and proposed the identification mechanisms (monitoring mechanisms adhered to the proposed taxonomy) as well as a set of possible recovery actions at workflow level.

6.1 Methodology

We consider the fault handling activity from an event-condition-action perspective. Consider f_i being a single fault (hardware/software), and $\{f\}$ a set of faults, leading to a *known* event/error e_i. The event causes a single action a_i or a set of actions (executed in some sequence) $\{a\}$ to be invoked to overcome the effect of the fault (undertaken using an automated system or by a human user). This can be expressed as:

$$(f_i|\{f\}) \rightarrow^{d_1} e_i \rightarrow^{d_2} \{m\} \rightarrow^{d_3} (f_i|\{a\}) \rightarrow^{d_4}$$

where d_i represents a time duration, with (d_1) representing the time after which a fault leads to an error message, (d_2) the time to monitor the error message using one or more monitoring tools m, (d_3) the time to invoke a corrective (recovery) action, and (d_4) the time over which the action must execute for the system to recover from the fault.

Thus, we can divide the process into three sub-activities: fault detection, fault identification and fault correction (recovery). The fault correction subactivity may utilize either backward or forward recovery [33] as undertaken in distributed systems generally, we do not advocate a new approach for undertaking this process, and primarily identify ways to achieve system recovery. In this work, we focus on faults that can be detected through error messages (propagated to a workflow enactor) that can be monitored (although we also identify the notion of an "unknown" fault for completeness).

We propose a taxonomy for conducting a systematic analysis of faults in scientific workflows. Our starting point is a general fault taxonomy developed in [42] for Grids, and our experience in the use of scientific workflow systems and applications. We demonstrate use of the taxonomy via a workflow within Weka4WS. We differ from [42] by: (i) focusing on the workflow enactment process, and not just the middleware or resource management systems; and (ii) considering an analysis (detection and recovery actions) of faults at the client side and not at the resource manager, which we believe represents a more realistic scenario for existing workflow systems.

6.2 Weka4WS extended architecture

An extended version of the Weka4WS architecture (see Figure 6.1) that is fault-aware was designed and implemented. The implementation is based on the faults taxonomy identified in this chapter, and demonstrates the use of the

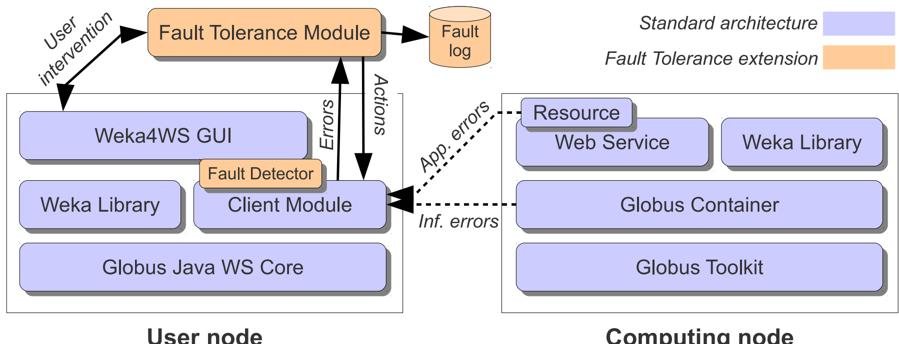

Fig. 6.1. Weka4WS extended architecture

taxonomy in the context of a particular system. In Weka4WS, errors consist of text messages extracted from Java exceptions and can be due to infrastructure or application faults. The infrastructure related errors are detected during the *task submission* phase by the Client Module at the user node, or by the Globus Container at the computing node and then propagated back to the Client Module. The application related errors are detected during the *task execution* phase by the Web Service at the computing node and stored into the Web Service Resource (as part of the WSRF implementation); the errors are then retrieved by the Client Module using either the push-style or the *pull-style* mode of the NotificationMessage delivery mechanism (Web Services Base Notification 1.3 OASIS Standard [88]), depending on the network scenario (see Section 4.2), where the Client Module is the NotificationConsumer and the Web Service is the NotificationProducer. When an error is detected the execution of the whole workflow is interrupted and the error message is displayed to the user.

In our proposed extension of the architecture, shown in Figure 6.1, an internal *Fault Detector* intercepts errors in the Client Module and redirects them to an external Fault Tolerance Module. Presently, we focus on fail-stop faults, however in order to detect more complex behaviors such as Byzantine faults, the Fault Detector Module would need to be extended with the consensus models described in [43]. The *Fault Tolerance Module* determines the fault types by parsing the returned error messages, interacts with Weka4WS if user intervention is needed, and finally invokes a set of pre-defined actions. The activity of the Fault Tolerance Module is stored in a *Fault log*, so that faults and the selected actions can be subsequently examined by the user.

6.3 Faults Taxonomy

The proposed faults taxonomy is illustrated in Figure 6.2. Two types of relationships are presented in it: (i) fault detection & identification; (ii) fault recovery. Fault detection & identification in our approach is an array containing elements from the fault taxonomy, which can be measured via monitoring tools (error detectors) to some degree of accuracy. Associated with each such array is one or more actions (referred to as fault recovery) that must be invoked to overcome the effects of the fault. An example on the array and the associated actions can be found in Section 7.3. Additionally, we identify a number of potential faults that may be propagated to the workflow-level, and they are shown on the right side of Figure 6.2 and include: insufficient memory, malformed datasets, incompatible datasets, etc. The circled numbers cross reference elements of the taxonomy with the identified faults, indicating that a fault can be a member of a number of different elements.

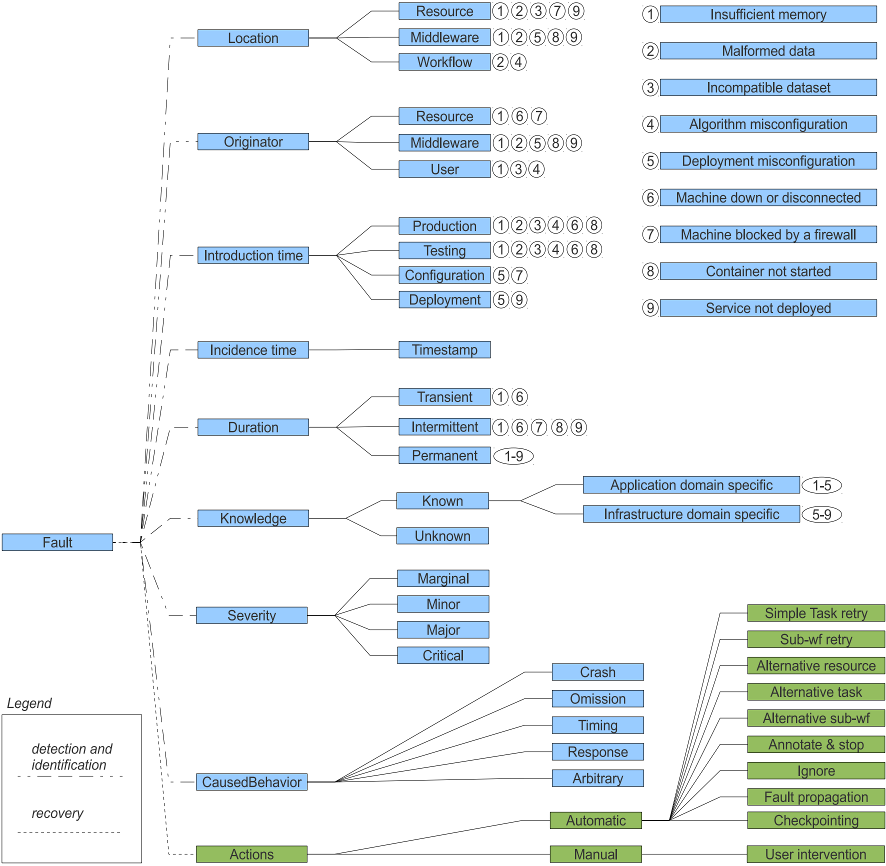

Fig. 6.2. Our Fault Taxonomy for Scientific workflows

6.4 Fault identification

Elements that are part of fault detection & identification are presented in this section. These elements must be measurable directly or capable of being inferred from the measured data.

- **Knowledge**: refers to the degree of accuracy to which a fault can be detected. Faults may be *known* or *unknown*; a *known* fault is one which can be detected correctly, whose effects are monitorable, and which may or not be corrected by the execution of an available action. Among the known faults, there are application domain specific faults, which are due to the specific algorithms utilized by a particular application, and infrastructure specific faults, which are due to the infrastructure(s) being used by the workflow system for the enactment process. Insufficient memory faults when executing data mining algorithms, malformed datasets, data mining algorithm misconfiguration, and dataset-algorithm incompatibilities belong to the application domain specific class; whereas a machine blocked by a firewall, disconnected machines, containers not started, services not deployed or other infrastructure configuration faults belong to the infrastructure specific class. In general terms, an *insufficient memory fault* occurs when a machine's memory is not sufficient to perform the execution of a remote request. A *malformed data* fault happens when the dataset in input presents some error in its header, attribute declaration or in its instances. An *algorithm misconfiguration* fault occurs when a wrong argument is passed to the algorithm, or a required argument is missing or when the argument values are out of range. Finally, an incompatibility between a correct dataset and an algorithm can happen when the algorithm is not suitable for the structure of the dataset.
  An *unknown* fault occurs when: i) suitable monitoring tools are not available to detect the outcome of the fault – i.e. no error conditions are generated when the fault occurs; ii) the error conditions cannot be monitored to any degree of confidence (accuracy).
- **Location**: refers to the part of the system where a fault manifests itself. We distinguish three main locations: resource, middleware, and workflow (or application). Examples of these types of faults can be: at *resource level*, an insufficient memory fault can occur due to insufficient memory at the computing node. Malformed dataset is produced due to data corruption during the data down/uploading process. Additionally, an incompatibility between a dataset and a particular data mining algorithm may occur when loading and checking the data at this stage. At *middleware level*, the transfer of datasets (accomplished by the middleware data movement services) can lead to dataset corruptions (i.e. a network failure). At *workflow level*, an algorithm misconfiguration fault can occur at development time and is generally presented to the user.
- **Originator**: refers to the component within the system responsible for having caused the fault – which can include resource, middleware, work-

flow, and user. An insufficient memory fault may be caused by a *resource*, a malformed dataset may be due to the *middleware*, for instance when a data movement service transmitted the data unsuccessfully. The middleware can also cause an insufficient memory fault, for instance when the (meta-) scheduling service binds a workflow task to a resource with insufficient memory for the execution. A user may be responsible for the misconfiguration of the algorithm or for malformed datasets (i.e. the structure of the datasets is wrong). Identifying the originator is an important part of the fault detection process.

- **Introduction time**: refers to the workflow lifecycle (design, testing or production) during which a fault was introduced in the system. Insufficient memory faults, malformed datasets, algorithm misconfigurations, and dataset-algorithm incompatibility can happen both at production and at testing. In contrast, faults depending on the infrastructure, such as service not deployed belong to other classes like deployment or configuration.
- **Incidence time**: The incidence time, via a timestamp, identifies when a fault happened during execution.
- **Duration**: Duration refers to the frequency of a fault, and is related to the incidence time, and include: transient, intermittent, and permanent faults. *Transient* faults occur once and then disappear completely. If the operation is repeated after a time t, no errors will be observed. An insufficient memory fault at a resource can belong to this class – as for shared resources, not enough memory may be available to execute a particular algorithm, at a particular time. However, due to completion of other processes at the resource, additional memory may be released non-deterministically. *Intermittent* faults can recur and there may be variable times between their occurrence. Insufficient memory faults can also be part of this class, as, in general terms, we do not have control of the available memory at a resource. Finally, a fault can be *permanent*, i.e. it requires recover action for the system to continue operation. Malformed data, algorithm misconfigurations, and dataset–algorithm incompatibilities are permanent faults.
- **Severity**: identifies the impact of the fault on system operation, and the difficulty of taking any associated corrective action. *Marginal* faults may have very limited effect, for instance some datasets can have attributes missing whose absence may not be significant. We can also recognise, in increasing degree of severity *minor*, *major*, and *critical* (the greatest impact) faults. However, in general terms, the severity of a fault is subject to ambiguous and subjective criteria. For instance, severity may also be related to whether the fault occurs during the testing or production phase. Hence, some aspects of "severity" can be inferred/assumed' (for instance, production phase faults are more severe than testing phase faults), whereas others need to be annotated by a user.
- **Caused Behavior**: Traditionally, faults have been classified according to their behavior and/or their severity (i.e. difficulty of detection). A *crash* (or fail-stop) fault occurs when a component prematurely halts, but was

working correctly until it stopped. Once the component crashed, it did not show any new activity or functionality. Insufficient memory faults, some malformed dataset faults (those that cannot be ignored) and incorrect configuration belong to this class. An *omission* fault occurs when a component fails to respond to a request. This may be due to a message reception problem, for instance when a connection was correctly established but there was no thread listening to incoming requests. Another class, the component may also fail to respond when something is wrong with the delivery functionality of the response, even if the request was successfully processed. Another class is a *timing* fault, whereby a response lies outside the expected real-time interval. A *response* fault occurs when the server's response is incorrect. Two kinds of response fault may happen: a *value* fault when the server provides a incorrect reply to a request and a *state transmission* fault when the server reacts unexpectedly to an incoming request. An *arbitrary* fault (or *Byzantine*) occurs when a server is producing inconsistent output but which cannot be detected as being incorrect.

6.5 Corrective Actions

As a consequence of a fault, an action or a set of actions may be undertaken at client side – which include:

- **Simple task retry**: on error, a task is retried on the same resource that failed.
- **Alternative resource**: on error, the execution of the task is retried with an alternative resource with equivalent functionality (i.e. due to an intermittent or permanent fault).
- **Alternative simple task**: on error, an alternative task that has equivalent functionality is selected and enacted.
- **Sub-workflow retry**: an intermediate node enacting a sub-workflow fails, and the execution of the sub-workflow is retried.
- **Alternative sub-workflow**: an intermediate node enacting a sub-workflow fails, and is replaced by an alternative one with equivalent functionality. This is achieved by modifying the workflow structure at runtime. This action may be taken because there was evidence of having a persistently failing descendant node which will not be able to complete successfully. Thus, the only possibility is to find an alternative set of descendants (alternative sub-workflow).
- **Fault propagation**: on detection, the error is propagated to higher layers of the workflow hierarchy – i.e. from an error at a resource level to the workflow enactor. This occurs because: i) the error cannot be corrected at the level at which it is detected (i.e. it is a permanent fault, and there are no alternative resources, tasks or sub-workflow available) or ii) although some actions for recovery may be available, it may be

better to accomplish it at another level (i.e. choosing an alternative dataset configuration to overcome an insufficient memory fault).

- **User intervention request**: faults may also be due to user error, such as malformed datasets or algorithm misconfiguration; and their recovery is very difficult if not impossible to be achieved in an automated way. To provide a balance between automatic recovery and user intervention, we propose the use of a "threshold value" which indicates, for instance, the maximum number of times one or more automated recovery techniques are attempted without leading to system recovery. When such a threshold is reached, the system asks for user intervention.
- **Fault ignore**: this corresponds to a marginal fault whose effect either can be ignored or can be repaired in an automated way (i.e. some missing attributes in a malformed dataset can be easily inferred from other parts of the dataset).
- **Fault annotation**: on error, in case there is no recovery action available, the fault can be annotated and tracked so that the user can examine it afterwards.
- **Checkpointing**: for intermittent faults, or when error rates exceed a user defined threshold, a checkpointing strategy may be initiated. In order to reduce the overheads arising out of a checkpoint strategy, the system may also modify the checkpoint frequency based on user defined criteria.

6.6 Fault detection & monitoring

In order to detect faults, we propose two mechanisms. The inherent mechanism provided by the workflow system and a fault detector that is external to the system. Workflow enactment systems can typically detect errors arising from faults at remote nodes. In all of these cases, typically, a programming language exception is generated at the distributed node which populates the corresponding attributes in the Web Service. On the other hand, different fault detection algorithms have been proposed in the literature, some example of them customised for Grid systems can be found in [43]. In general terms, their degree of sophistication, understood in terms of variety of faults they can detect (ranging from fail-stop faults to Byzantine faults), is related to the number of replicas that need to be provided.

On fault detection, the system attempts to monitor the different elements of our fault taxonomy, forming what we call the *fault array*. This can be performed by: (i) obtaining values directly via observation from monitoring tools, and (ii) inferring values from historical data or from user entered variables and / or conditions (i.e. the user provides an annotation to identify that whenever an insufficient memory fault occurs, then the fault severity is critical).

In particular, we identify how the taxonomy can be used in the Weka4WS system. For instance, when a Java exception occurs, a timestamp is included

to set the *incidence time*, with the exception information and the timestamp being sent to the workflow engine. The workflow engine can infer additional values to populate the fault array and identify the fault, such as:

- **Obtaining the Location**: whenever a Java exception is generated by the Weka library in the server node, this value is set to *resource*. In case the fault manifests itself within the application container (i.e. WSRF container) or picked up by the fault detector, the value is set to *middleware*.
- **Inferring the Originator**: this value is specific to the application domain. In Weka4WS, potential faults include: insufficient memory, malformed data and algorithm misconfiguration. For the insufficient memory fault, the user is the originator in case he / she bound a task in the workflow with a resource in the environment without enough memory for the execution. The originator can also be a failed scheduling (middleware) action or can be due to a resource (identified by the monitoring process). In case it is not the resource, the actual originator can only be determined at workflow level, in case the task was scheduled, it will be a scheduling problem. For the malformed data, it can be due to a corrupted transfer of the dataset, a corrupted hard disk storage, or a user error. An MD5 or CRC check on the dataset can be used to identify a network or hard-disk problem.
- **Obtaining the Introduction time**: a user annotation is used to determine whether the fault occurred during the testing or production phase.
- **Obtaining the Incidence time**: as described before, this is set when the fault was detected, and can be obtained from the exception information.
- **Inferring the Duration**: this is inferred from the incidence type, fault type and the frequency of occurrence. Currently, we only focus on fail-stop faults.
- **Obtaining the Knowledge**: this can be obtained from monitoring, either by the information captured from the node at which the exception occurred or by a fault detection algorithm.
- **Inferring the Severity**: users must annotate conditions in order to allow the Fault Tolerance module from Figure 6.1 to infer the severity of a fault. The conditions can make use of values obtained from the monitoring process. Such data can be utilized by the Fault Tolerance Module to prioritise faults that occur within a predefined period of time.

In some scenarios, it is possible for a fault to be detectable, but not fully identifiable, thereby leading to an *unknown fault*. Hence, there are no previous conditions in the log/monitored data that match the obtained *fault array*: this could occur because either some values from monitoring could not be obtained or in spite of having obtained them, no match can be found. In such circumstances, identifying a suitable action to execute to overcome the fault remains an open research challenge.

6.7 Conclusions

The distribution of tasks to third-party, inter-organizational resources makes fault tolerance an important subject for scientific workflows. Some faults may be managed directly by the underlying resource management systems. However, this cannot always be guaranteed and depends on the type of resource manager being used. Consequently, the development of fault tolerance mechanisms at workflow (user) level become important. Because scientific workflows have no control over external resources, the set of available actions to overcome faults directly, on external systems, is limited. For that reason, and in order to allow a workflow system to select the most suitable action, a taxonomy for conducting a systematic analysis on faults within scientific workflows has been proposed.

Our fault taxonomy, identified in Figure 6.2, considers both a means to identify faults and suitable corrective actions that need to be invoked to overcome them. We believe the relationship between these two characteristics is important to support in a workflow enactment system, particularly when identifying which action can be automated to overcome the effect of a fault. The taxonomy we propose is general purpose, and can be applied across a variety of different workflow systems. However, it is important to note that some monitoring information is, by necessity, system specific. For instance, in our taxonomy, we include Globus specific information (for use with Weka4WS), which does not translate to other workflow systems. However, the general category under which such information is captured, such as *middleware* or *configuration*, could have alternative counterparts for other systems.

7

Validation

7.1 Weka4WS use cases and performance

In this section we present two examples of distributed data mining workflows designed and executed on a Grid using the Knowledge Flow component of Weka4WS. The first workflow defines a classification application while the second one defines a clustering application. Both of these workflows have been executed on a Grid environment composed of five machines, to evaluate performance and scalability of the system. In addition, we present the execution of a workflow on a multi-core machine to show how Weka4WS can obtain lower execution times compared to Weka even when executed on a single computer.

7.1.1 Classification workflow

Data mining applications that easily exploit the Weka4WS approach are those where a dataset is analyzed in parallel on multiple Grid nodes using different data mining algorithms. For example, a given dataset can be concurrently classified using different classification algorithms with the aim of finding the "best" classifier on the basis of some evaluation criteria (e.g., error rate, confusion matrix, etc.). We used the Knowledge Flow component of Weka4WS to build such kind of application, in which a dataset is analyzed in parallel using four different classification algorithms: *Decision Stump*, *Naive Bayes*, *J48* and *Random Forest*.

The dataset analyzed is *kddcup99*, publicly available at the UCI KDD archive [41]. This dataset contains a wide set of data produced during seven weeks of monitoring in a military network environment subject to simulated intrusions. From the original dataset we removed all but 9 attributes and the class attribute, using a selection filter provided by the Preprocess panel of the Explorer component. Then, from the resulting dataset, we used another filter to extract three datasets with a number of instances of 215000, 430000 and 860000, and a file size of about 7.5 MB, 15 MB and 30 MB respectively. The same classification workflow has been executed for each of those datasets.

Figure 7.1 shows the workflow designed to build the application. The workflow begins (on the left side) with an *ArffLoader* node, used to load the *kdd-cup99* dataset from file, which is connected to a *CrossValidation FoldMaker* node (set to 5 folds), used for splitting the dataset into training and test sets according to a cross validation. The *CrossValidation FoldMaker* node is connected to four nodes, each one performing the four algorithms mentioned earlier. These are in turn connected to a *Classifier PerformanceEvaluator* node for the model validation, and then to a *TextViewer* node for results visualization.

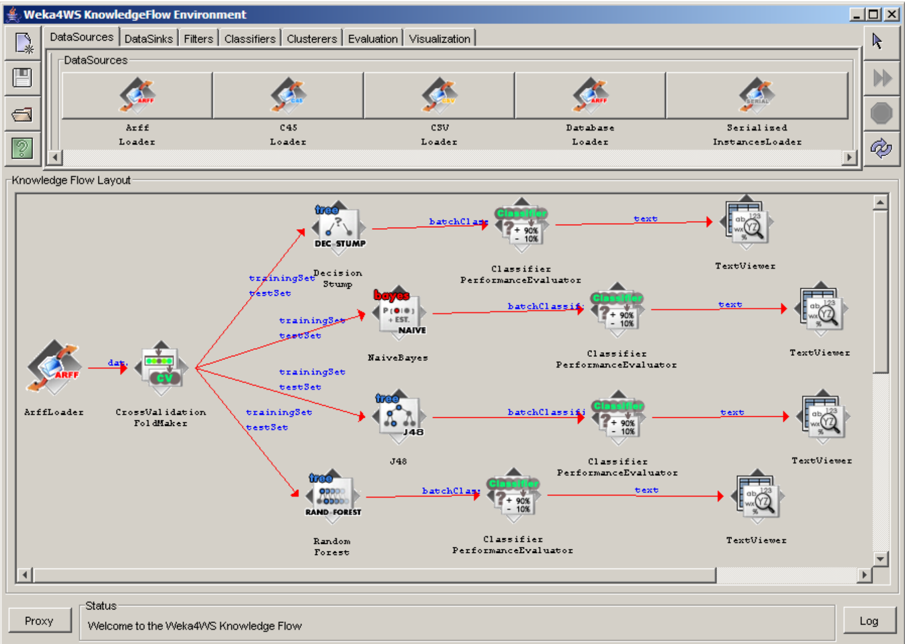

Fig. 7.1. Workflow of the classification application

When the application is started by the user, the four branches of the workflow are executed in parallel. For each of the three dataset sizes (215, 430 and 860 thousands instances), the workflow has been executed using 1 to 4 Grid nodes in order to evaluate the speedup of the system. The machines used for the experiments had Intel Pentium processors ranging from 2.8 GHz to 3.2 GHz, RAM ranging from 1 GB to 2 GB, and belong to two local area networks. The results of the experiments are shown in Figure 7.2.

For the largest dataset (860k instances) the total execution time decreases from 2456 sec (about 41 min) using 1 node, to 1132 sec (about 19 min) using 4 nodes, achieving a speedup of 2.17. For the smallest dataset (215k instances) the total execution time passes from 117 seconds with 1 node, to 55 seconds

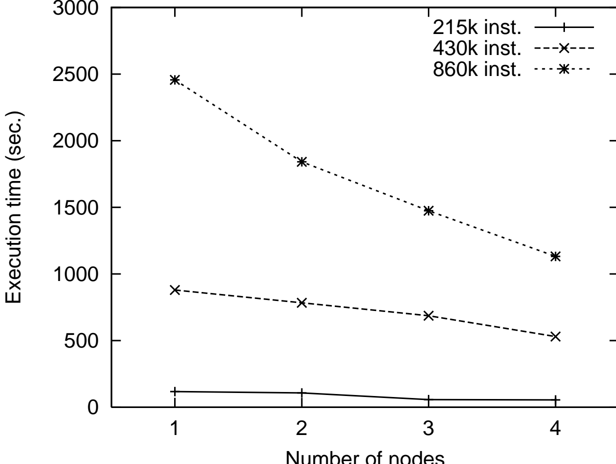

Fig. 7.2. Execution times of the classification workflow using 1 to 4 Grid nodes for datasets with 215, 430 and 860 thousands instances

using 4 nodes, with a speedup of 2.13. We can observe that the speedup value is not so high as one might expect. The reason of that is the diversity of the classification algorithms used in the example, coupled with the heterogeneity of computing nodes.

Since to execute the model evaluation all the four classifiers must have completed their execution, the total classification time is given by the slower classifier. However this experiment shows that the workflow execution time decreases significantly when large datasets are analyzed on Grids.

It is worth noticing that the total execution time is made by the sum of three contributions: file transfer time, WSRF overhead, and data mining time. In all cases, the file transfer took just a few seconds to complete (about 6 seconds in the worst case). Also the WSRF overhead, that is the overall time needed to invoke the service, subscribe to notification, and receive the results as explained in Section 4.4, took a small time (less than 4 seconds). The sum of file transfer time and WSRF overhead is therefore negligible compared to the data mining time which is the most relevant part of the total execution time, as already detailed in [78].

7.1.2 Clustering workflow

The data mining workflow described in this section implements a parameter sweeping application in which a given dataset is analyzed using multiple instances of the same algorithm with different parameters. In particular, we used the Knowledge Flow to compose an application in which a dataset is analyzed by running multiple instances of the same clustering algorithm, with the goal of obtaining multiple clustering models from the same data source.

The dataset *covertype*, from the UCI KDD archive [41], has been used as data source. The dataset contains information about forest cover type for 581012 sites in the United States. Each dataset instance, corresponding to a site observation, is described by 54 attributes that give information about the main features of a site (e.g., elevation, aspect, slope, etc.). The 55th attribute contains the cover type, represented as an integer in the range 1 to 7. From this dataset we extracted three datasets with about 72, 145 and 290 thousands instances and a file size of about 9 MB, 18 MB and 36 MB respectively. Then we used Knowledge Flow to perform a clustering analysis on each of those datasets. The workflow corresponding to the application is shown in Figure 7.3.

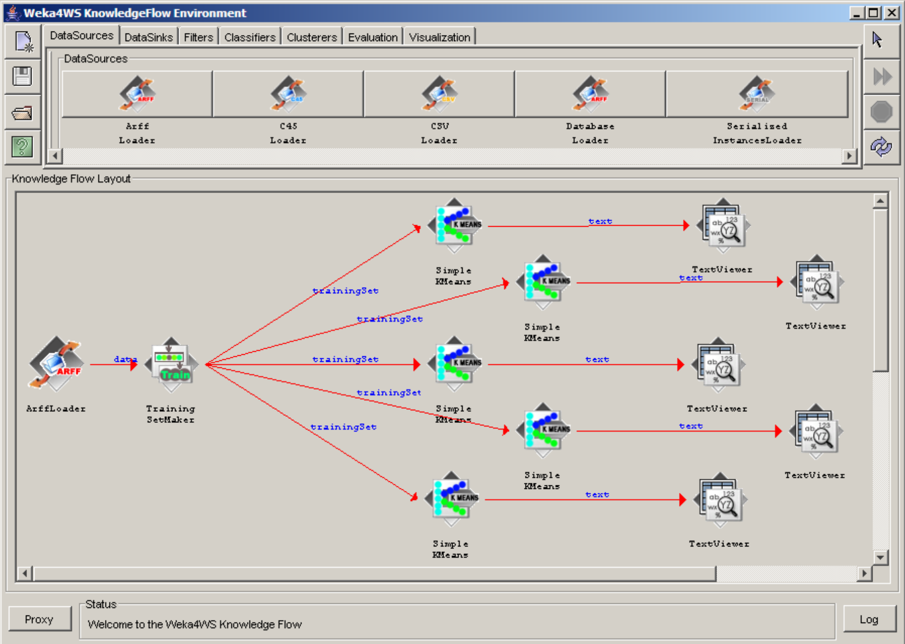

Fig. 7.3. Workflow of the clustering application

This workflow is similar to the one shown in Figure 7.1: it includes an *ArffLoader* node connected to a *Training SetMaker* node, used for accepting a dataset and producing a training set. The *Training SetMaker* node is connected to 5 nodes, each one performing the *KMeans* clustering algorithm, and each one set to group data into a different number of clusters (3 to 7), based on all the attributes but the last one (the cover type). These nodes are in turn connected to a *TextViewer* node for results visualization.

The workflow has been executed using a number of computing nodes ranging from 1 to 5 for each of the three datasets (72k, 145k and 290k instances). The execution times are shown in Figure 7.4.

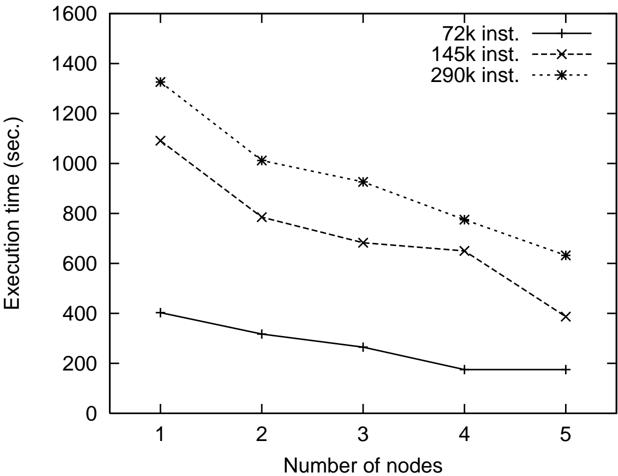

Fig. 7.4. Execution times of the clustering workflow using 1 to 5 Grid nodes for datasets with 72, 145 and 290 thousands instances

With the largest dataset as input, the total execution time decreases from 1326 seconds obtained using 1 computing node, to 632 seconds obtained on 5 nodes. For the smallest dataset, the execution time passed from 403 to 175 seconds using 1 to 5 nodes. The execution speedup with 5 nodes ranged between 2.10 to 2.82. In this case the speedup is mainly limited by the different amounts of time taken by the various clustering tasks included in the workflow: the slower algorithm determines the total execution time. However, the speedup values for large datasets are good.

7.1.3 Execution on a multi-core machine

In this section we present the execution times of a classification workflow when it is executed locally on a multi-core machine. Since Weka4WS executes an independent thread for each branch of the workflow, we obtain lower execution times compared to Weka even when the workflow is run on a single multiprocessor and/or multi-core machine.

The workflow considered here is a variant of the parameter sweeping workflow presented in the previous section. In this case, a data source is analyzed in parallel using 4 instances of the J48 classification algorithm, configured to use a confidence factor of 0.20, 0.30, 0.40 and 0.50.

As data source, we used 6 datasets extracted from the *covertype* dataset introduced earlier. Those datasets have a number of instances ranging from 39 to 237 thousands, with a size ranging from 5 MB to 30 MB. For each of those 6 datasets as input, we executed the same workflow with Weka and Weka4WS. The machine used for this experiment has two Intel Xeon dual-core processors with a clock frequency of 3 GHz and 2 GB of RAM. The execution times are reported in Figure 7.4

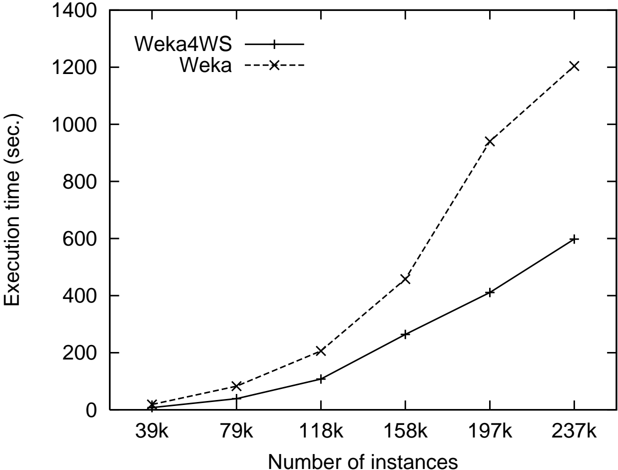

Fig. 7.5. Execution times of the classification workflow using Weka and Weka4WS on a two-processor dual-core machine, for a dataset having 39k to 237k instances

With Weka, the execution time ranges from 19 seconds for the dataset with 39k instances, to 1204 seconds for the dataset with 237k instances. Using Weka4WS the execution time passes from 8 to 598 seconds, saving an amount of time ranging from a minimum of 42% to a maximum of 58%. These results confirm that the multi-threaded approach of Weka4WS is well suited also to fully exploit the computational power of multi-processor and/or multi-core machines.

7.2 Knowledge Grid use cases and performance

In this section we present two examples of distributed data mining workflows designed and executed on a Grid using the DIS3GNO system. The first workflow is a parameter sweeping application in which a dataset is processed using multiple instances of the same classification algorithm with different parameters, with the goal of finding the best classifier based on some accuracy

parameters. In the second workflow, a dataset is analyzed using different classification algorithms. The resulting classification models are combined through voting to derive a global model that is more accurate than the single ones. Both of these workflows have been executed on a Grid environment composed of several machines to evaluated the effectiveness of the systems as well as its performance in terms of scalability.

7.2.1 Parameter sweeping workflow

We used DIS3GNO to compose an application in which a given dataset is analyzed by running multiple instances of the same classification algorithm, with the goal of obtaining multiple classification models from the same data source.

The dataset *covertype*[1] from the UCI KDD archive, has been used as data source. The dataset contains information about forest cover type for a large number of sites in the United States. Each dataset instance, corresponding to a site observation, is described by 54 attributes that give information about the main features of a site (e.g., elevation, aspect, slope, etc.). The 55th attribute contains the cover type, represented as an integer in the range 1 to 7. The original dataset is made of 581,012 instances and is stored in a file having a size of 72MB. From this dataset we extracted three datasets with 72500, 145000 and 290000 instances and a file size of 9 MB, 18 MB and 36 MB respectively. Then we used DIS3GNO to perform a classification analysis on each of those datasets.

DIS3GNO has been used to run an application in which 8 independent instances of the J48 algorithm perform a different classification task on the covertype data set. In particular, each J48 instance has been asked to classify data using a different value of confidence, ranging from 0.15 to 0.50. The same application has been executed using a number of computing nodes ranging from 1 to 8 to evaluate the speedup of the system.

The workflow corresponding to the application is shown in Fig. 7.6. It includes a Dataset node (representing the covertype dataset) connected to 8 Tool nodes, each one associated to an instance of the J48 classification algorithm with a different value of confidence (ranging from 0.15 to 0.50). These nodes are in turn connected to another Tool node, associated to a model chooser which selects the best classification model among those learnt by the J48 instances. Finally, the node associated to the model chooser is connected to a Viewer node having the location set to localhost; this enforces the model to be transferred to the client host for its visualization.

The workflow has been executed using a number of computing nodes ranging from 1 to 8 for each of the three datasets (9 MB, 18 MB and 36 MB) in order to evaluate the speedup of the system. Table 7.1 reports the execution times of the application when 1, 2, 4 and 8 computing nodes are used.

[1] http://kdd.ics.uci.edu/databases/covertype/covertype.html

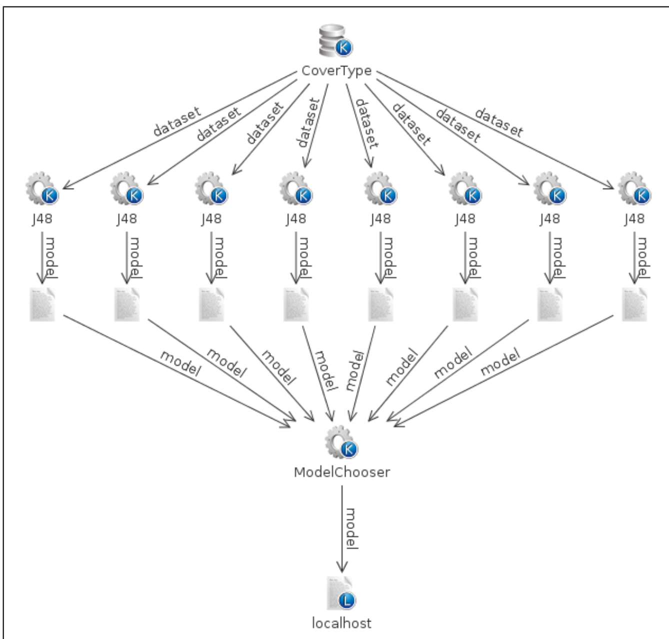

Fig. 7.6. Parameter sweeping workflow.

The 8 classification tasks that constitute the overall application are indicated as $DM_1..DM_8$, corresponding to the tasks of running J48 with a confidence value of 0.15, 0.20, 0.25, 0.30, 0.35, 0.40, 0.45, and 0.50, respectively. The table shows how the classification tasks are assigned to the computing nodes (denoted as $N_1..N_8$), as well as the execution times for each dataset size.

Table 7.1. Task assignments and execution times for the parameter sweeping workflow (times expressed as hh:mm:ss)

No of nodes	**Task assignments (Node ← Tasks)**	**Exec. time 9 MB**	**Exec. time 18 MB**	**Exec. time 36 MB**
1	$N_1 \leftarrow DM_1, ..., DM_8$	2:43:47	7:03:46	20:36:23
2	$N_1 \leftarrow DM_1, DM_3, DM_5, DM_7$ $N_2 \leftarrow DM_2, DM_4, DM_6, DM_8$	1:55:19	4:51:24	14:14:40
4	$N_1 \leftarrow DM_1, DM_5$ $N_2 \leftarrow DM_2, DM_6$ $N_3 \leftarrow DM_3, DM_7$ $N_4 \leftarrow DM_4, DM_8$	58:30	2:26:48	7:08:16
8	$N_i \leftarrow DM_i$ for $1 \leq i \leq 8$	32:35	1:21:32	3:52:32

When the workflow is executed on more than one node, the execution time includes the overhead due to file transfers. For example, in our network scenario, the transfer of a 36 MB dataset from the user node to a computing node takes on average 15 seconds. This value is small as compared to the amount of time required to run a classification algorithm on the same dataset, which takes between 2.5 and 3.9 hours depending on the computing node. The overall execution time also includes the amount of time needed to invoke all the involved services (i.e., EPMS, RAEMS, KDS) as required by the workflow. However, such an amount of time (approximatively 2 minutes) is negligible as compared to the total execution time.

For the 36 MB dataset, the total execution time decreases from more than 20 hours obtained using 1 computing node, to less than 4 hours obtained with 8 nodes. The achieved execution speedup ranged from 1.45 using 2 nodes, to 5.32 using 8 nodes. Similar trends have been registered with the other two datasets. The execution times and speedup values for different number of nodes and dataset sizes are shown in Fig. 7.7.

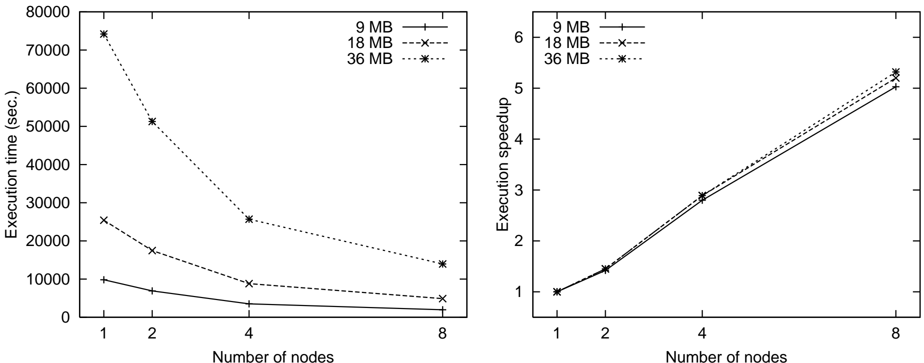

Fig. 7.7. Execution times and speedup values for different numbers of nodes and dataset sizes, for the parameter sweeping workflow

7.2.2 Ensemble learning workflow

Ensemble learning is a machine learning paradigm where multiple learners are trained to solve the same problem. In contrast to ordinary machine learning approaches which try to learn one model from training data, ensemble methods build a set of models and combine them to obtain the final model [92]. In a classification scenario, an ensemble method constructs a set of *base classifiers* from training data and performs classification by taking a vote on the predictions made by each classifier. As proven by mathematical analysis, ensemble classifiers tend to perform better (in terms of error rate) than any single classifier [79].

The DIS3GNO system has been exploited to design a workflow implementing an ensemble learning application which analyzes a given dataset using different classifiers and performs a voting on the models inferred by them.

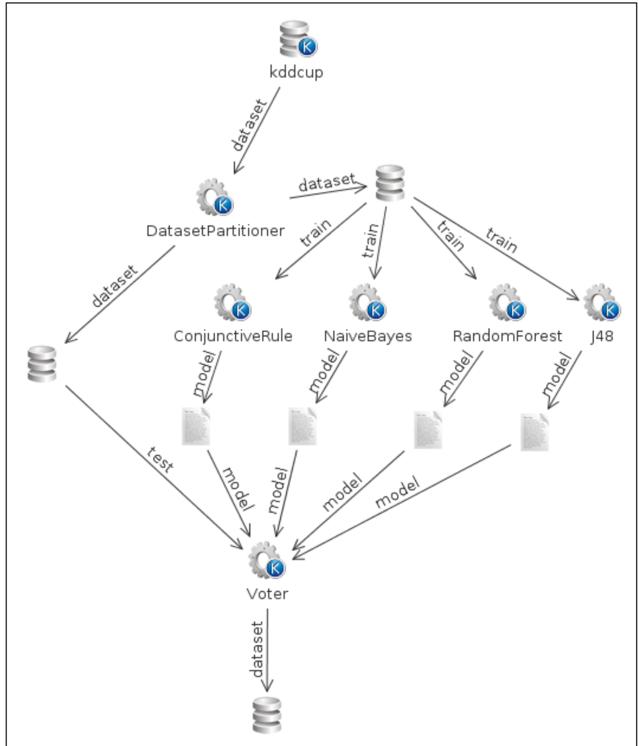

Fig. 7.8. Ensemble learning workflow.

As input dataset we used *kddcup99*[2]. This data set, used for the KDD'99 Competition, contains a wide set of data produced during seven weeks of monitoring in a military network environment subject to simulated intrusions. We extracted three data sets from it, with 940000, 1315000 and 1692000 instances and a size of 100 MB, 140 MB and 180 MB.

DIS3GNO has been used to split the dataset into two parts: a test set (1/3 of the original dataset) and a training set (2/3 of the original dataset). The latter has been processed using four classifiers: ConjuctiveRule, NaiveBayes, RandomForest and J48. The models generated by the four classifiers are then collected to a node where they are given to a voter component; the classification is performed and evaluated on the test set by taking a vote, for each instance, on the predictions made by each classifier. The same workflow has

[2] http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

been executed, for each of the three datasets, using a number of computing nodes ranging from 1 to 4 (excluding the node where we performed the voting operation) to evaluate the speedup of the system.

The workflow corresponding to the application is shown in Fig. 7.8. It includes a Dataset node representing the kddcup dataset, connected to a Tool node associated to a dataset partitioner, which from which a test set and a training set are obtained, as detailed above. The training set is connected to four Tool nodes, associated to the classification algorithms mentioned earlier. The four models generated by such algorithms are connected to a Tool node associated to a voter which assigns to each istance of the test set a class obtained through a voting operation.

Table 7.2 reports the execution times of the application when 1, 2 and 4 computing nodes are used. The four tasks are indicated as $DM_1..DM_4$, corresponding ConjuctiveRule, NaiveBayes, RandomForest and J48 respectively. The table shows how the tasks are assigned to the computing nodes, as well as the execution times for each dataset size.

Table 7.2. Task assignments and execution times for the ensemble learning workflow

No of nodes	**Task assignments (Node ← Tasks)**	**Exec. time 100 MB**	**Exec. time 140 MB**	**Exec. time 180 MB**
1	$N_1 \leftarrow DM_1, ..., DM_4$	1:30:50	2:31:14	3:34:27
2	$N_1 \leftarrow DM_1, DM_3$ $N_2 \leftarrow DM_2, DM_4$	1:03:47	1:37:05	2:07:05
4	$N_1 \leftarrow DM_i$ for $1 \leq i \leq 4$	46:16	1:13:47	1:37:23

The execution times and speedup values for different number of nodes and dataset sizes are represented in Fig. 7.9. In this case, the speedup is lower than that obtained with the parameter sweeping workflow. This is due to the fact that the four algorithms used require very different amounts of time to complete their execution on a given dataset. In fact, the overall execution time is bound to the execution time of the slowest algorithm, thus limiting the speedup. However, the absolute amount of time saved by running the application on a distributed environment is still significant, particularly for the largest dataset when four computing nodes are used.

7.3 Analysis of potential faults

Let us take the example of data mining workflow shown Figure 7.10 built using the Knowledge Flow component of Weka4WS. The workflow consists of two classification algorithms, Naïve Bayes and J48, running in parallel and bound by the user to two different computing nodes at workflow design time. At execution time, each computing node receives a copy of a dataset, and

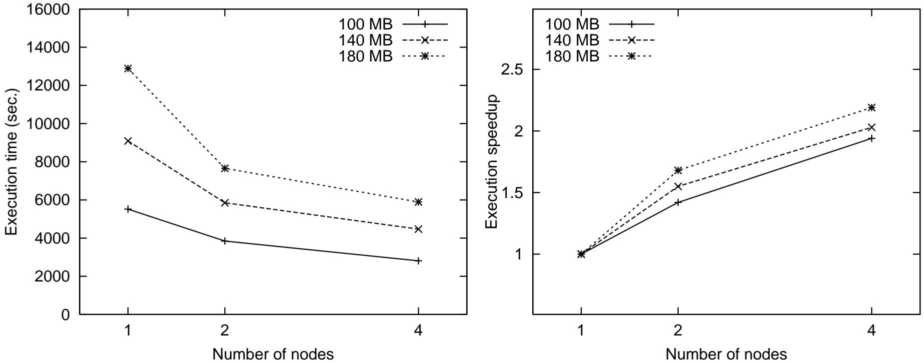

Fig. 7.9. Execution times and speedup values for different numbers of nodes and dataset sizes, for the ensemble learning workflow

after the execution at the computing nodes, the results are sent back to the user node and displayed through the Text Viewer component.

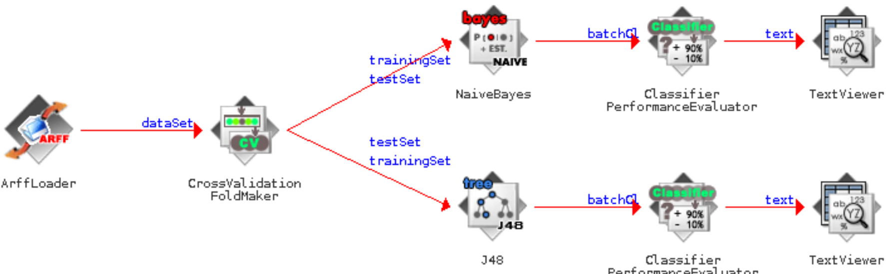

Fig. 7.10. A data mining workflow example

Table 7.3. Fault Identification Array and Action Recovery.

Knowledge	insuff. mem	insuff. mem	malf. data	malf. data
Location	resource	resource	resource	workfow
Originator	user / sched.	res.	res.	user
Introd. time	production	production	production	production
Incid. time	t_1	t_2	t_3	t_4
Duration	perm.	trans./ interm./perm.	perm.	perm.
Severity	critical	critical	marginal	major
Caused Behavior	crash	crash	crash	crash
Action	user	alternative resource	ignore	user

We assume that the user has some apriori knowledge about the overall amount of memory required for each algorithm – this may be the size of the binary (executable) file associated with each algorithm, or based on previous executions of the algorithm by the same (or other) users. We consider two different execution scenarios for the workflow in Figure 7.10, both leading to an insufficient memory fault. In one scenario, the algorithm is enacted on a machine without enough memory, and in the other the amount of available memory varies due to other, concurrently sent, external requests. In both scenarios, the Weka4WS system can detect insufficient memory faults, as an insufficient memory fault raises a Java exception in the computing node which is propagated to the workflow enactor (user node).

In the workflow enactor, the fault is intercepted by the fault detector and redirected to the fault tolerance module (see Figure 6.1), which will try to extract the fault type from this data and generate a fault identification array. The fault identification array exploits the viewpoints from the taxonomy introduced in Section 6.3. In both scenarios, the array consists of the following values: knowledge (known fault, application domain specific, insufficient memory fault), location (network address of a computing node), and incidence time (a timestamp of the occurrence). Once these values are populated, the fault manager will try to find user's annotations or infer the rest.

At design time, the user will annotate the workflow as a production or testing workflow, so that the introduction time can be obtained from this. As the severity depends on a user's perception, let us suppose that the severity for an insufficient memory fault type has been set as "critical". Caused behaviour is a crash (fail-stop) fault because it was detected by internal mechanisms provided by Weka4WS. The main difference between these two execution scenarios, however, lies in the originator, as in the first case it was the user's or the meta-scheduler's responsibility and in the second it was due to the resource manager at the computing node. The error message received by the fault manager identifies the memory available at the computing node and the total memory required to execute the algorithm. By comparing these two values, it can be determined whether the fault was due to the user / meta-scheduler (not enough memory) or due to other external requests. Finally, the duration can also be inferred to some extent: in case the fault was due to the user, it is clearly a permanent fault (as it is not going to be fixed on its own after a period of time); whereas in case the fault was due to multiple external requests leading to the insufficient memory fault, the fault can be either transient, intermittent or permanent.

Once the fault has been identified, and the array generated, the fault manager can try to find a matching in its records (as for instance Table 7.3 shows), so that the most suitable action (or a set of actions) can be executed for the recovery. According to Table 7.3, in the first execution scenario, the fault manager can report the problem to the user (though, alternatively, the fault manager could have been set to attempt to find an alternative computing node with enough memory). For the second scenario, the fault manager is set

to find an alternative resource (it could also have been set to attempt to retry the execution on the same node). Other policies could also apply, for instance by introducing multiple alternative actions for a fault and introducing priorities for each action, indicating which action should be attempted first.

7.4 Conclusions

The experimental evaluation conducted by executing some typical data mining patterns has demonstrated the effectiveness of both the Weka4WS and Knowledge Grid systems to support data mining workflows design and execution in distributed Grid environments. The use of the fault taxonomy has been demonstrated through an example of workflow within Weka4WS, considering two different execution scenarios leading to a fault.

8

Conclusions and Future Work

This thesis originated from our research aimed at providing data mining services and workflow systems for analyzing scientific data in a high performance distributed environment such as the Grid.

We described the design and implementation of Weka4WS, an extension of Weka which, adopting both WSRF technologies and the services offered by Globus Toolkit, provides support to the designing of distributed applications which coordinate the execution of multiple data mining tasks on a set of Grid nodes. Our framework extends, in particular, the functionalities of the Knowledge Flow component of Weka which allow users to compose knowledge discovery workflows made by several algorithms and data analysis processes. Thanks to the extensions implemented, the Knowledge Flow of Weka4WS allows the parallel execution of the data mining algorithms which are part of the workflow on several Grid nodes, hence allowing to reduce the execution time, as ascertained by the performance tests results presented above. Weka4WS can help reducing the execution time of multiple data mining algorithms also when used on a single multi-processor and/or multi-core machine.

We described the Knowledge Grid, a software system that we developed for providing services to execute distributed data mining tasks in Grid environments. In particular, we designed and developed the DIS3GNO component to provide a set of visual programming facilities to design and execute distributed data mining workflows and to manage the resources of the Knowledge Grid. All the Knowledge Grid services for metadata access and execution management are accessed transparently by DIS3GNO, thus allowing the domain experts to compose and run complex data mining applications without worrying about the underlying infrastructure details. In addition, DIS3GNO allows the composition of abstract workflows, that is workflows whose nodes may be not completely specified. In this way, a user can concentrate on the application logic, without focusing on the actual datasets or data mining tool to be used. The Knowledge Grid services will take care of finding the resources that fit user specifications.

The main contributions of this thesis to the field are:

- the design and implementation of Weka4WS, an extension of a well established data mining environment, the Weka toolkit, to allow the execution its data mining workflows in a Grid: in this way, domain experts can exploits the computing power and storage capability of a Grid infrastructure but can focus on designing their data mining applications, without worrying about learning complex tools or languages for Grid submission and management;

- the definition a workflow formalism and the design and implementation of a visual environment, called DIS3GNO, for the composition and execution of data mining tasks on the Knowledge Grid. DIS3GNO provides mechanisms for publishing and searching the needed resources, creating and executing data mining workflows whose nodes may also be not fully specified, thus leaving the Knowledge Grid services the task of finding the resources that fit user specifications;

- the definition of a fault taxonomy for scientific workflows that may help in conducting a systematic analysis of faults, so that the potential faults that may arise at execution time can be corrected.

Experiments have been performed to assess the efficiency of both workflow systems as well as to test effective design of real data mining applications. The results of the tests performed have proven their good scalability and that the overhead introduced by the remote invocations of the algorithms does not affect significantly the performance of the systems. As for the fault taxonomy it has been demonstrated through an example of workflow within Weka4WS, considering two different execution scenarios leading to a fault.

The approaches and the models used have demonstrated the feasibility of the workflow systems, which are available [1] and are used in the scientific community.

Future Work

Future developments of this research work are:

- improve the usability of the workflow systems by: (i) giving the user full control over the use of their data, allowing for example to choose whether a dataset they own can be maintained on the node where it is processed or has to be deleted, (ii) increasing the current expressive power of workflow by adding constructs for flow control, by adding support for hierarchical workflows and by adding support for workflow templates. The use of workflow templates allows the parameters of the workflow nodes to assume a range of values, and a step of increment, in order to avoid duplication of

[1] http://grid.deis.unical.it/

branches of the workflow which are very similar thus providing more flexibility in performing various data mining algorithms in parallel;

- improve the efficiency of the workflow systems by: (i) making an efficient scheduling of the tasks on the grid nodes by using dynamic information about the resources, (ii) reusing results from past workflows computations, or portions of workflows, in order to save time in the execution of new workflows, in case the new ones would contain portions identical to some executed in the past (this will imply finding efficient ways for the workflow to automatically generate the metadata about the way the work was done), (iii) adding support for the adaptive data parallelism, in which the optimal number of partitions is determined automatically as a function of the number of available resources on the Grid.

Acknowledgements

During the time of writing of this PhD thesis I received support and help from many people. In particular, I am thankful to my supervisor, Prof. Domenico Talia, who was very generous with his time and knowledge and assisted me in each step to complete the thesis.

I would like to thank also the people of my research group, especially Paolo Trunfio and Eugenio Cesario, who helped closely during my study period, giving me motivation and assistance whenever I needed it. Thanks are also due to Gianluigi Folino and Carlo Mastroianni for their intellectual support, ideas and feedback.

I am grateful to Prof. Omer F. Rana, who has given me the privilege to work with him in Cardiff, as well as to Rafael Tolosana for his encouragement and friendship. To them and to the other friends I met in Cardiff, Ioan Petri, Sathish Periyasamy and Yaser Alosefer, goes my deepest gratitude for the intellectual and cultural growth they have provided me.

And finally, but not least, thanks go to my whole family, who have been an important and indispensable source of moral support, especially my girlfriend, Claudia Rotella, whose precious presence has been vital to the achievement of this goal.

References

1. Al Sairafi S, Emmanouil F S, Ghanem M, Giannadakis N, Guo Y, Kalaitzopoulos D, Osmond M, Rowe A, Syed J, Wendel P, "*The Design of Discovery Net: Towards Open Grid Services for Knowledge Discovery*". Int. Journal of High Performance Computing Applications, 17(3): 297-315, 2003.
2. Allcock W, Bresnahan J, Kettimuthu R, Link M, Dumitrescu C, Raicu I, Foster I, "*The Globus striped GridFTP framework and server*". Supercomputing Conference, 2005.
3. Altintas I, Berkley C, Jaeger E, Jones M, Ludascher B, Mock S, "*Kepler: an extensible system for design and execution of scientific workflows*". 16th International Conference on Scientific and Statistical Database Management, 2004.
4. Avizienis A, Laprie J C, Randell B, Landwehr C, "*Basic Concepts and Taxonomy of Dependable and Secure Computing*", IEEE T. on Dep. and Sec. Comp.1(1): pp. 11-32, 2004.
5. Berman F, "*Viewpoint: From TeraGrid to Knowledge Grid*". Communications of the ACM 44 (11) (2001)27-28.
6. "*Business Process Execution Language for Web Services*" (BPEL4WS). http://www.ibm.com/developerworks/library/specification/ws-bpel/ [Visited: 25 November 2010]
7. Bresnahan J, Link M, Khanna G, Imani Z, Kettimuthu R, Foster I, "*Globus GridFTP: What's New in 2007*". (Invited Paper). Proceedings of the First International Conference on Networks for Grid Applications (GridNets 2007), Oct, 2007.
8. Brezany P, Hofer J, Min Tjoa A, Woehrer A, "*GridMiner: An Infrastructure for Data Mining on Computational Grids*". Proc. APAC Conference and Exhibition on Advanced Computing, Grid Applications and eResearch, Queensland, Australia, 2003.
9. Caeiro-Rodriguez M, Priol T, Nmeth Z,"*Dynamicity in scientific workflows*". Technical report, CoreGRID, 2008.
10. Cannataro M, Talia D, "*The Knowledge Grid*". Communitations of the ACM, vol. 46, n. 1, pp. 89-93, 2003.
11. Chan P, Stolfo S J, "*A Comparative Evaluation of Voting and Meta-learning on Partitioned Data*". 12th International Conference on Machine Learning, 1995.
12. Chapman P, Clinton J, Kerber R, Khabaza T, Reinart T, Shearer C, Wirth R, "*CRISP-DM Step-by-Step Data Mining Guide*", 2000.

http://www.crisp-dm.org/CRISPWP-0800.pdf
[Visited: 25 November 2010]

13. Churches D, Gombas G, Harrison A, Maassen J, Robinson C, Shields M, Taylor I, Wang I, "*Programming scientific and distributed workflow with Triana services: Research articles*". Concurrency and Computation: Practice & Experience, 18(10):1021-1037, 2006.
14. Cristian F, "*Understanding fault-tolerant distributed systems*". Commun. ACM 34, 2, 56-78, February, 1991.
15. Congiusta A, Talia D, Trunfio P, "*Distributed data mining services leveraging WSRF*". Future Generation Computer Systems, vol. 23, n. 1. Elsevier Science: 34-41, 2007.
16. Congiusta A, Talia D, Trunfio P, "*Using Grids for Distributed Knowledge Discovery*". In: Mathematical Methods for Knowledge Discovery and Data Mining, (eds) Felici G, Vercellis. IGI Global, 284-298, 2007.
17. Czajkowski K, et al. (2006) The WS-Resource Framework Version 1.0. http://www-106.ibm.com/developerworks/library/ws-resource/ws-wsrf.pdf [Visited: 25 November 2010]
18. Czajkowski K, Ferguson D F, Foster I, Frey J, Graham S, Sedukhin I, Snelling D, Tuecke S, Vambenepe W, "*The WS-Resource Framework*". March 5, 2004.
19. Deelman E, Blythe J, Gil Y, Kesselman C, Mehta G, Patil S, Su M H, Vahi K, Livny M, "*Pegasus: Mapping Scientific Workflows onto the Grid*". Across Grids Conference, 2004.
20. Deelman E, Mehta G, Singh G, Su M, Vahi K (2007). "*Mapping Large-Scale Workflows to Distributed Resources*", in Workflows for eScience, pp 376-394. Springer, 2006.
21. Djorgovski S G, "*Virtual Astronomy, Information Technology, and the New Scientific Methodology*". Proc. 7th Int. Workshop on Computer Architectures for Machine Perception, 2005.
22. Duan R, Prodan R, Fahringer T,"*Dee: A distributed fault tolerant workflow enactment engine for grid computing*". In (eds)Yang LT, Rana O F, Martino B D, Dongarra J, High Performance Computing and Communications, First International Conference, HPCC 2005, Sorrento, Italy, September 21-23, 2005, Proceedings, volume 3726 of Lecture Notes in Computer Science, pp 704-716. Springer, 2005.
23. Fahringer T, Jugravu A, Pllana S, Prodan R, Seragiotto Junior C, Truong H L, "*ASKALON: A Tool Set for Cluster and Grid Computing*". Concurrency and Computation: Practice & Experience, vol. 17, n. 2-4, 2005.
24. Fayyad U M, et al "*From data mining to knowledge discovery: an overview*". In Fayyad, (eds) U. M.et al, Advances in knowledge discovery and data mining. AAAI Press / The MIT Press, 1996.
25. Felix C. Gärtner."*Fundamentals of fault-tolerant distributed computing in asynchronous environments*". ACM Comput. Surv. 31, 1 (March 1999), 1-26.
26. Feller M, Foster I, Martin S, "*GT4 GRAM: A Functionality and Performance Study*".
27. Foster I. "*Globus Toolkit Version 4: Software for service-oriented systems*". Conference on Network and Parallel Computing, LNCS 3779, 2005.
28. Foster I,"*What is the grid? a three point checklist*". GRIDToday, July, 2002.
29. Foster I, Kesselman C, Nick J, Tuecke S, "*The Physiology of the Grid*". In: (eds) Berman F, Fox G, Hey A, Grid Computing: Making the Global Infrastructure a Reality, Wiley: 217-249, 2003.

30. Foster I, Kishimoto H, Savva A, Berry D, Djaoui A, Grimshaw A, Horn B, Maciel F, Siebenlist F, Subramaniam R, Treadwell J, Von Reich J, "*The Open Grid Services Architecture, Version 1.0*". Informational Document, Global Grid Forum (GGF), January 29, 2005.
31. Foster I, Vöckler JS, Wilde M, Zhao Y (2002), "*Chimera: A virtual data system for representing, querying, and automating data derivation*". In Proceedings of the 14th International Conference on Scientific and Statistical Database Management, July 24-26, 2002, Edinburgh, Scotland, UK, pp 37-46. IEEE Computer Society.
32. Fox G C, Gannon D, "*Special Issue: Workflow in Grid Systems*". Concurrency and Computation: Practice and Experience, 18: 1009-1019. doi: 10.1002/cpe.1019, 2006.
33. Gärtner F C, "*Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments*". *ACM Computing Surveys*, 31(1), 1-26, 1999.
34. Gelernter D, "*Generative communication in linda*". ACM Trans. Program. Lang. Syst., 7(1):80-112, 1985.
35. Gil Y, Ratnakar V, Deelman E, Mehta G, Kim J,"*Wings for Pegasus: Creating large-scale scientific applications using semantic representations of computational workflows*". In Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada, pp 1767-1774. AAAI Press, 2007b.
36. Graham S, et al. (2004) Publish-Subscribe Notification for Web services. http://www.oasis-open.org/committees/download.php/6661/WSNpubsub-1-0.pdf
[Visited: 25 November 2010]
37. Grossman R L, Kamath C, Kegelmeyer P, Kumar V, Namburu R R, "*Data mining for scientific and engineering applications*". Kluwer Academic Publishers, 2001.
38. Guan Z, Hernandez F, Bangalore P, Gray J, Skjellum A, Velusamy V, Liu Y, "*Grid-Flow: a Grid-enabled scientific workflow system with a Petri-net-based interface: Research Articles*". Concurr. Comput. : Pract. Exper. 18, 10 (August 2006), 1115-1140, 2006.
39. Guan Z, Liu Y, Velusamy V, Bangalore P V, "*WebRun: A unified platform supporting Grid computing environment*". Technical Report UABCIS-TR-2004-1404-1, Department of Computer and Information Sciences, University of Alabama at Birmingham, 2004.
40. Hand D, Mannila H, Smyth P,"*Principles of Data Mining*". MIT Press, 2001.
41. Hettich S, Bay S D, The UCI KDD Archive, University of California, Department of Information and Computer Science.
http://kdd.ics.uci.edu
[Visited: 25 November 2010]
42. Hofer J, Fahringer T, "*A multi-perspective taxonomy for systematic classification of grid faults*". In *6th Conf. PDP*, pp 126-130. IEEE Comp. Soc., 2008.
43. Hofer J, Fahringer T, "*Synthesizing Byzantine Fault-Tolerant Grid Application Wrapper Services*". In *Int Conf on CCGRID 2008*, pp 467-474, IEEE Computer Society, 2008
44. Hoheisel A, Alt M, "*Petri Nets*" in Workflows for eScience, pp 190-207. Springer-Verlag, 2006.
45. Hoheisel A,"*User tools and languages for graph-based grid workflows: Research articles*". Concurr. Comput. : Pract. Exper., 18(10):1101-1113, 2006.

46. Hull D, Wolstencroft K, Stevens R, Goble C, Pocock M, Li P, Oinn T, "*Taverna: a tool for building and running workflows of services*". Nucleic Acids Research, vol. 34, Web Server issue, pp. 729-732, 2006.
47. Jensen K, Rozenberg G, (eds), "*High-level Petri nets: theory and application*". Springer-Verlag, London, UK, 1991.
48. Johnston W E, "*Computational and Data Grids in Large Scale Science and Engineering*". Future Generation Computer Systems 18 (8) (2002), 1085-1100.
49. Kamath C, "*Scientific Data Mining: A Practical Perspective*". SIAM, 2009.
50. Kargupta H, Park B, Hershberger D, Johnson E, "*A New Perspective toward Distributed Data Mining*". In: Advances in Distributed and Parallel Knowledge Discovery, (eds) Kargupta H, Chan P. AAAI/MIT Press, 133-184, 2000.
51. Kesselmann C,Foster I, Tuecke S, "*The anatomy of the grid: Enabling scalable virtual organizations*". International J. Supercomputer Applications, 2001.
52. Khalaf R, Keller A, Leymann F, "*Business processes for Web services: principles and applications*". IBM Syst. J., 45(2):425-446, 2006.
53. Kosiedowski M, Kurowski K, Mazurek C, Nabrzyski J, Pukacki J, "*Workflow applications in GridLab and PROGRESS projects*". In Concurrency and Computation: Practice and Experience, 18, 10 (August 2006), 1141-1154.
54. Lackovic M, Talia D, Trunfio P, "*A Framework for Composing Knowledge Discovery Workflows in Grids*". In: Foundations of Computational Intelligence Vol 6: Data Mining Theoretical Foundations and Applications, Studies in Computational Intelligence, Abraham A, Hassanien A, Carvalho A, Snášel V (Editors), Springer, 2009.
55. Lackovic M, Talia D, Trunfio P, "*Service Oriented KDD: A Framework for Grid Data Mining Workflows*". 10th International Workshop on High Performance Data Mining, 2008.
56. Lamport L, "*Proving the correctness of multiprocess programs*". IEEE Trans. Softw. Eng. 3, 2, 125-143, Mar 1977.
57. Lee E A, Neuendorffer S, "*Actor-oriented models for codesign: Balancing re-use and performance*". In Formal Methods and Models for System Design. Kluwer, 2004.
58. Lee K, Paton N, Sakellariou R, Deelman E, Fernandes A, Mehta G, "*Adaptive workflow processing and execution in Pegasus*". In Third International Workshop on Workflow Management and Applications in Grid Environments (WaGe08), May 25-28, 2008, Kunming, China, pp 99-106.
59. Leymann F, Roller D,"*Modeling business processes with BPEL4WS*". Inf. Syst. E-Business Management, 4(3):265-284, 2006.
60. Ludscher B, Altintas I, Berkley C, Higgins D, Jaeger E, Jones M, Lee E A, Tao J, Zhao Y. "*Scientific workflow management and the Kepler system*". In Concurrency and Computation: Practice & Experience, 2005.
61. Mastroianni C, Talia D, Trunfio P, "*Metadata for Managing Grid Resources in Data Mining Applications*". Journal of Grid Computing 2 (1) (2004), 85-102.
62. McCann K M, Yarrow M, DeVivo A, Mehrotra P, "*ScyFlow: An Environment for the Visual Specification and Execution of Scientific Workflows*". Concurrency and Computation: Practice and Experience, 18: 1155-1167, 2006.
63. McGough A S, Lee W, Cohen J, Katsiri E, Darlington J,"*ICENI*" in Workflows for eScience, pp 395-415. Springer-Verlag, 2006.
64. Meyer F, "*Genome Sequencing vs. Moore's Law: Cyber Challenges for the Next Decade*". CTWatch Quarterly 2 3, 2006.

65. Moore R, "*Knowledge-based Grids*". Proc. 18th IEEE Symposium on Mass Storage Systems and 9th Goddard Conference on Mass Storage Systems and Technologies, 2001.
66. Ngu A, Bowers S, Haasch N, McPhillips T, Critchlow T, "*Flexible scientific workflow modeling using frames, templates, and dynamic embedding*". In Scientific and Statistical Database Management, pp 566-572, 2008.
67. Oinn T, Greenwood M, Addis M, Alpdemir MN, Ferris J, Glover K, Goble C, Goderis A, Hull D, Marvin D, Li P, Lord P, Pocock MR, Senger M, Stevens R, Wipat A, Wroe C. "*Taverna: lessons in creating a workflow environment for the life sciences: Research articles*". Concurrency and Computation: Practice and Experience, 18(10):1067-1100, 2006.
68. Pautasso C, Alonso G, "*Parallel Computing Patterns for Grid Workflows*". Workshop on Workflows in Support of Large-Scale Science, 2006.
69. Pease M, Shostak R, Lamport L, "*Reaching Agreement in the Presence of Faults*". J. ACM 27, 2 (April 1980), 228-234, 1980.
70. Plankensteiner K, Prodan R, Fahringer T, Kertesz A, "*Kacsuk: Fault-tolerant behavior in state-of-the-art Grid Workflow Management Systems*". TR-0091, Core-GRID, 2007.
71. Prodromidis A L, Chan P K, Stolfo S J, "*Meta-learning in Distributed Data Mining Systems: Issues and Approaches*". In: Advances in Distributed and Parallel Knowledge Discovery, (eds) Kargupta H, Chan P. AAAI/MIT Press, 81-87, 2000.
72. Shields M, Taylor I, "*Programming Scientific and Distributed Workflow with Triana Services*". Workflow in Grid Systems Workshop in GGF10, 2004.
73. Schuller B, Demuth B, Mix H, Rasch K, Romberg M, Sild S, Maran U, Bala P, del Grosso E, Casalegno M, Piclin N, Pintore M, Sudholt W, Baldridge K K, "*Chemomentum - UNICORE 6 based infrastructure for complex applications in science and technology*". Proceedings of 3rd UNICORE Summit 2007 in Springer LNCS 4854, Euro-Par 2007 Workshops: Parallel Processing, pp. 82-93.
74. Siddiqui M, Fahringer T, "*Grid Resource Management: On-demand Provisioning, Advance Reservation, and Capacity Planning of Grid Resources*". Springer, 2010.
75. Stankovskia V, Swainb M, Kravtsovc V, Niessend T, Wegenerd D et al., "*Grid-enabling data mining applications with DataMiningGrid: An architectural perspective*". Future Generation Computer Systems 24 (4) (2008), 259-279.
76. Streit A, Bala P, Beck-Ratzka A, Benedyczak K, Bergmann S, et al. "*UNICORE 6 Recent and Future Advancements*". In Annals of Telecommunications, Springer Paris, 2010.
77. Talia D, Trunfio P, "*How Distributed Data Mining Tasks can Thrive as Knowledge Services*". Communications of the ACM 53 (7) (2010), 132-137.
78. Talia D, Trunfio P, Verta O, "*The Weka4WS framework for distributed data mining in service-oriented Grids*". Concurrency and Computation: Practice and Experience, 20(16) (2008), 1933-1951.
79. Tan P N, Steinbach M, Kumar V, "*Introduction to Data Mining*". Addison-Wesley, 2006.
80. Tanenbaum A S, van Steen M, "*Distributed Systems: Principles and Paradigms*". Prentice Hall International, 2008.

81. Taylor I, Shields M, Wang I, Harrison A, "*The Triana Workflow Environment: Architecture and Applications*". In: (eds) Taylor I, Deelman E, Gannon D, Shields M. Workflows for e-Science, Springer: 320-339, 2007.
82. Thompson H S, "*What's a URI and why does it matter?*". School of Informatics, University of Edinburgh, Markup Systems, 26 August 2010 http://www.ltg.ed.ac.uk/∼ht/WhatAreURIs/ [Visited: 25 November 2010]
83. Tolosana-Calasanz R, Bañares J A, Rana O F, Álvarez P, Ezpeleta J, Hoheisel A,"*Adaptive exception handling for scientific workflows*". Concurr. Comput. : Pract. Exper. 22, 5 (April 2010), 617-642, 2010.
84. Tolosana-Calasanz R, Bañares J A, Álvarez P, Ezpeleta J, Rana O F, "*An uncoordinated asynchronous checkpointing model for hierarchical scientific workflows*". J. Comput. Syst. Sci. 76, 6 (September 2010), 403-415, 2010.
85. Valk R, "*Petri nets as token objects: An introduction to elementary object nets*". In (eds) Desel J, Silva M, Application and Theory of Petri Nets 1998, 19th International Conference, ICATPN 98, Lisbon, Portugal, June 22-26, 1998, Proceedings, volume 1420 of Lecture Notes in Computer Science, pp 1-25. Springer, 1998.
86. Von Laszewski G, Amin K, Hampton S, Nijsure S, "*GridAnt - White Paper*". Technical report, Argonne National Laboratory, Argonne, IL., July 2002.
87. Von Laszewski G, Hategan M, Kodeboyina D,"*Java CoG Kit Workflow*", in Workflows for eScience, pp 340-356. Springer-Verlag, 2006.
88. Web Services Base Notification 1.3 (2006) OASIS Standard http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf [Visited: 25 November 2010]
89. Witten H, Frank E, "*Data Mining: Practical machine learning tools with Java implementations*". Morgan Kaufmann, 2000.
90. Yu J, Buyya R, "*A Novel Architecture for Realizing Grid Workflow using Tuple Spaces*". In Fifth IEEE/ACM International Workshop on Grid Computing, pages 119-128. IEEE Computer Society Press: Los Alamitos, CA, 2004.
91. Zhang X, Schopf J. "*Performance Analysis of the Globus Toolkit Monitoring and Discovery Service, MDS2*". Proceedings of the International Workshop on Middleware Performance (MP 2004), part of the 23rd International Performance Computing and Communications Workshop (IPCCC), April 2004.
92. Zhou Z H,"*Semi-supervised learning by disagreement*". 4th IEEE International Conference on Granular Computing, pp. 93, 2008.