UNIVERSITÀ DELLA CALABRIA

Dipartimento di Elettronica,
Informatica e Sistemistica

Dottorato di Ricerca in
Ingegneria dei Sistemi e Informatica
XXVciclo

Tesi di Dottorato

Designing Cloud services for data processing and knowledge discovery

Fabrizio Marozzo

D.E.I.S., Novembre 2012
Settore Scientifico Disciplinare: ING-INF/05

UNIVERSITÀ DELLA CALABRIA

Dipartimento di Elettronica,
Informatica e Sistemistica

Dottorato di Ricerca in
Ingegneria dei Sistemi e Informatica
XXV ciclo

Tesi di Dottorato

Designing Cloud services for data processing and knowledge discovery

Fabrizio Marozzo

Coordinatore
Prof. Luigi Palopoli

Supervisori
Prof. Domenico Talia

Ing. Paolo Trunfio

DEIS

DEIS- DIPARTIMENTO DI ELETTRONICA, INFORMATICA E SISTEMISTICA
Novembre

Settore Scientifico Disciplinare: ING-INF/05

Preface

The past two decades have been characterized by an exponential growth of digital data in many field of human activities, from science to enterprise. For example in biological, medical, astronomic and earth science fields, very large data sets are produced daily from sensors, instruments and computers.

To find interesting information and extract meaning out of those big data repositories, it is necessary and helpful to work with data analysis environments allowing the effective and efficient access, management and mining of such repositories. Cloud systems can be effectively used to handle data mining processes since they provide scalable processing and storage services, together with software platforms for developing data analysis environment on top of such services.

The goal of this thesis is studying and exploiting the Cloud paradigm to support scalable knowledge discovery (KDD) applications in distributed scenarios. Three approaches have been investigated: the first one is the use of the Map/Reduce programming model for processing large data sets on dynamic Cloud environments; the second one is the use of the COMPSs paradigms and its sequential code approach for the execution of parallel data processing on hybrid Cloud infrastructures; finally, the design and implementation of a high-level visual environment to support a wide range of knowledge discovery applications on the Cloud.

The result of the first research activity is an extension of the architecture of current MapReduce implementations to make it more suitable for dynamic large-scale Cloud environments. The proposed system, called P2P-MapReduce, exploits a peer-to-peer model to manage intermittent node participation, master failures and job recovery in a decentralized but effective way, so as to provide a robust middleware MapReduce that can be effectively exploited in Internet-scale dynamic distributed environments.

The second research activity described in this thesis is an extension of COMPSs, a framework that provides a programming model and a runtime system that ease the development of distributed applications and their execution on a wide range of computational infrastructures. The goal of the

extension is enhancing the interoperability layer to support the execution of COMPSs applications into the Windows Azure Platform. The framework has been evaluated through the porting of a data mining workflow to COMPSs and its execution on a hybrid testbed.

Finally, we worked to design a Cloud framework for supporting the scalable execution of distributed knowledge discovery applications. The framework, called Data Mining Cloud Framework, has been implemented using Windows Azure and has been evaluated through a set of KDD applications executed on a Microsoft Cloud data center. Three classes of knowledge discovery applications are supported: single-task applications, in which a single data mining task is performed on a given dataset; parameter-sweeping applications, in which a dataset is analyzed by multiple instances of the same data mining algorithm with different parameters; workflow-based applications, in which knowledge discovery applications are specified as graphs linking together data sources, data mining tools, and data mining models.

Prefazione

Gli ultimi decenni sono stati caratterizzati da una crescita esponenziale dei dati in diversi settori, dalla scienza all'impresa. Ad esempio, nei settori della biologia, della medicina, dell'astronomia e della scienza della terra, ogni giorno enormi archivi di dati vengono generati da parte di sensori, strumenti e computer.

Al fine di estrarre conoscenza da tali archivi, è necessario e utile lavorare con ambienti di analisi che consentano l'accesso efficace ed efficiente ai dati, la loro gestione, e l'estrazione di conoscenza. I sistemi Cloud possono essere efficacemente usati per gestire i processi di estrazione della conoscenza in quanto forniscono potenza di calcolo e servizi di memorizzazione in modo altamente scalabile, nonché piattaforme software che permettono lo sviluppo di ambienti di analisi dei dati.

L'obiettivo di questa tesi è lo studio e l'uso del paradigma Cloud per l'implementazione di applicazioni scalabili di knowledge discovery in databases (KDD) in scenari distribuiti. Sono stati seguiti tre approcci diversi: il primo è l'uso del modello di programmazione MapReduce per l'elaborazione ed analisi di grandi quantità di dati in ambienti Cloud dinamici; il secondo è l'uso del paradigma COMPSs e il suo approccio a codice "sequenziale" per l'esecuzione parallela di applicazioni di analisi dati su infrastrutture Cloud ibride; infine, la progettazione e la realizzazione di un ambiente visuale di alto livello per l'implementazione di applicazioni di knowledge discovery su Cloud.

Il risultato della prima attività di ricerca è stato lo studio di un'estensione dell'architettura master-worker delle attuali implementazioni del modello MapReduce, per renderle più adeguate ad ambienti Cloud dinamici e di larga scala in termini di affidabilità e scalabilità. Il sistema proposto, chiamato P2P-MapReduce, fa uso di un modello peer-to-peer per definire un'architettura adattativa in grado di permettere la partecipazione intermittente dei nodi, i fallimenti dei master e il recupero dei job in modo decentralizzato ma efficiente, offrendo in tal modo un middleware MapReduce più affidabile che può essere effettivamente sfruttato su ambienti dinamici e distribuiti di larga scala.

La seconda attività di ricerca descritta in questa tesi è un'estensione di COMPSs, un framework che fornisce un modello di programmazione che semplifica lo sviluppo di applicazioni distribuite e la loro esecuzione su una vasta gamma di infrastrutture di calcolo (Cloud, griglie computazionali, cluster). Attraverso tale estensione abbiamo aumentato l'interoperabilità di tale framework così da permettere l'esecuzione di applicazioni COMPSs anche sulla piattaforma Cloud Windows Azure. Il framework è stato valutato attraverso applicazioni di data mining basate su workflow, che sono state eseguite su un ambiente Cloud ibrido (Windows Azure insieme ad una piattaforma Cloud privata).

Infine, abbiamo implementato un framework Cloud per l'esecuzione scalabile di applicazioni di knowledge discovery. Il framework, chiamato Data Mining Cloud framework, è stato realizzato utilizzando Windows Azure ed è stato valutato attraverso una serie di applicazioni KDD eseguite sui data center della Microsoft. Il framework supporta tre classi di applicazioni di knowledge discovery: applicazioni "single-task", in cui un task di data mining viene eseguito su un insieme di dati; applicazioni "parameter-sweeping", in cui un insieme di dati viene analizzato in parallelo da diverse istanze dello stesso algoritmo di data mining eseguite con diversi parametri; applicazioni "workflow-based", in cui l'applicazione di knowledge discovery viene definita come un workflow visuale formato da un grafo che collega insieme basi di dati, strumenti e modelli di data mining.

Rende, Cosenza, Italy *Fabrizio Marozzo*

November 2012

Contents

1 Introduction 1
1.1 Objectives of the Research 2
1.1.1 P2P-MapReduce 2
1.1.2 COMPSs with Azure 3
1.1.3 Data Mining Cloud Framework 3
1.2 Publications 4
1.2.1 Journals 4
1.2.2 Book Chapters 4
1.2.3 Papers in refereed conference proceedings 4
1.2.4 Other publications 5
1.3 Organization of the Thesis 5

2 Cloud computing 7
2.1 Service models 8
2.2 Deployment models 11
2.3 Development environments 12
2.3.1 Windows Azure 12
2.3.2 Amazon Web services 13
2.3.3 OpenNebula 14
2.4 Programming paradigms and frameworks 15
2.4.1 MapReduce 16
2.4.2 COMPSs 16
2.4.3 Sector/Sphere 17
2.4.4 All-Pairs 17
2.4.5 Dryad 18
2.4.6 Aneka 18

3 P2P-MapReduce 19
3.1 Background and Related Work 20
3.1.1 The MapReduce Programming Model 20
3.1.2 Related Work 22

3.2 System Model and Architecture 23
3.2.1 System Model 24
3.2.2 Architecture 26
3.3 System Mechanisms 29
3.3.1 Node Behavior 31
3.3.2 Job and Task Management 36
3.3.3 User Behavior 39
3.4 Implementation 40
3.5 System Evaluation 42
3.5.1 Experimental Setup and Methodology 42
3.5.2 Fault Tolerance 45
3.5.3 Network Traffic 46
3.5.4 Scalability 50
3.5.5 Remarks 51
3.6 Conclusion 52

4 COMPSs applications on the Cloud 53
4.1 The COMPSs framework 54
4.2 The Azure JavaGAT Adaptor 56
4.3 Data mining on COMPSs: a classifier-based workflow 58
4.3.1 The application workflow 58
4.3.2 The application implementation 59
4.3.3 Parallelization with COMPSs: the interface 60
4.4 Performance evaluation 62
4.5 Related work 64
4.6 Conclusions and future work 65

5 Data Mining Cloud Framework 67
5.1 Cloud-based Data Mining 67
5.1.1 Functional requirements 68
5.1.2 Non-functional requirements 69
5.1.3 Cloud for distributed KDD 71
5.2 Data Mining Cloud Framework 73
5.2.1 System Model 73
5.2.2 General architecture 75
5.2.3 Execution mechanisms 76
5.2.4 User Interface 79
5.3 Implementing the Data Mining Cloud Framework 79
5.3.1 Fulfilling the functional requirements with Azure 80
5.3.2 Implementing the system components on Azure 80
5.4 Parameter-sweeping data mining applications 81
5.5 Workflow-based data mining applications 83
5.5.1 Workflow formalism 84
5.5.2 Workflow composition 88
5.5.3 Workflow execution 92

5.6 Experimental results 94
5.6.1 Parameter sweeping data mining applications 94
5.6.2 Workflows-based data mining applications 98
5.7 Conclusions 102

6 Conclusions 103

References 107

1

Introduction

The past two decades have been characterized by an exponential growth of digital data production in many fields of human activities, from science to enterprise. Very large data sets are produced daily from sensors, instruments, and computers, and are often stored in distributed data repositories. For example, astronomers have to analyze terabytes of image data that every day comes from telescopes and artificial satellites; physicists must study the huge amount of data generated by particle accelerators to understand the laws of Universe; medical doctors and biologists collect huge amount of information about patients to search and try to understand the causes of diseases. Such examples demonstrate how the exploration and automated analysis of massive datasets powered by computing capabilities are fundamental to advance our knowledge in many fields.

Unfortunately, massive datasets are hard to understand, and in particular models and patterns hidden in them cannot be comprehended neither by humans directly, nor by traditional analysis methodologies. To cope with big data repositories, parallel and distributed knowledge discovery and data mining techniques must be used. It is also necessary and helpful to work with data analysis environments allowing the effective and efficient access, management and mining of such repositories. For example a scientist can use data analysis environments to execute complex simulations, validate models, compare and share results with colleagues located world-wide.

To face the challenge of extracting knowledge from big data repositories in efficient way, researchers and companies have turned to parallel computing in the Cloud. As a concept, the Cloud is an abstraction for remote, infinitely scalable provisioning of computation and storage resources [1]. In reality, Cloud infrastructures are based on large sets of computing resources, located somewhere "in the Cloud", which are allocated to applications on demand. Thus Cloud computing can be defined as a distributed computing paradigm in which all the resources, dynamically scalable and often virtualized, are provided as a service through Internet. Cloud systems can be effectively used to handle data mining processes since they provide scalable processing and storage services,

together with software platforms for developing data analysis environment on top of such services.

The goal of this thesis is studying and exploiting the Cloud paradigm to support scalable knowledge discovery applications in distributed scenarios. Three approaches have been investigated: the first one is the use of the Map/Reduce programming model for processing large data sets on dynamic Cloud environments; the second one is the use of the COMPSs paradigms and its sequential code approach for the execution of parallel data processing on hybrid Cloud infrastructures; finally, the design and implementation of a high-level visual environment to support a wide range of knowledge discovery applications on the Cloud.

The result of the first research activity is an extension of the architecture of current MapReduce implementations to make it more suitable for dynamic large-scale Cloud environments. The proposed system, called P2P-MapReduce, exploits a peer-to-peer model to manage intermittent node participation, master failures and job recovery in a decentralized but effective way, so as to provide a robust middleware MapReduce that can be effectively exploited in Internet-scale dynamic distributed environments.

The second research activity described in this thesis is an extension of COMPSs, a framework that provides a programming model and a runtime system that ease the development of distributed applications and their execution on a wide range of computational infrastructures. The goal of the extension is enhancing the interoperability layer to support the execution of COMPSs applications into the Windows Azure Platform. The framework has been evaluated through the porting of a data mining workflow to COMPSs and its execution on a hybrid testbed.

Finally, we worked to design a Cloud framework for supporting the scalable execution of distributed knowledge discovery applications. The framework has been implemented using Windows Azure and has been evaluated through a set of KDD applications executed on a Microsoft Cloud data center. Three classes of knowledge discovery applications are supported: single-task applications, in which a single data mining task is performed on a given dataset; parameter-sweeping applications, in which a dataset is analyzed by multiple instances of the same data mining algorithm with different parameters; workflow-based applications, in which knowledge discovery applications are specified as graphs linking together data sources, data mining tools, and data mining models.

1.1 Objectives of the Research

1.1.1 P2P-MapReduce

MapReduce is a programming model for parallel data processing widely used in Cloud computing environments. Current MapReduce implementations are

based on centralized master-slave architectures that do not cope well with dynamic Cloud infrastructures, like a Cloud of Clouds, in which nodes may join and leave the network at high rates. We have designed an adaptive MapReduce framework, called *P2P-MapReduce*, which exploits a peer-to-peer model to manage node churn, master failures, and job recovery in a decentralized but effective way, so as to provide a more reliable MapReduce middleware that can be effectively exploited in dynamic Cloud infrastructures. This paper describes the P2P-MapReduce system providing a detailed description of its basic mechanisms, a prototype implementation, and an extensive performance evaluation in different network scenarios. The performance results confirm the good fault tolerance level provided by the P2P-MapReduce framework compared to a centralized implementation of MapReduce, as well as its limited impact in terms of network overhead.

1.1.2 COMPSs with Azure

The advent of Cloud computing has given to researchers the ability to access resources that satisfy their growing needs, which could not be satisfied by traditional computing resources such as PCs and locally managed clusters. On the other side, such ability, has opened new challenges for the execution of their computational work and the managing of massive amounts of data into resources provided by different private and public infrastructures.

COMP Superscalar (COMPSs) is a programming framework that provides a programming model and a runtime that ease the development of applications for distributed environments and their execution on a wide range of computational infrastructures. COMPSs has been recently extended in order to be interoperable with several Cloud technologies like Amazon, OpenNebula, Emotive and other OCCI compliant offerings.

This paper presents the extensions of this interoperability layer to support the execution of COMPSs applications into the Windows Azure Platform. The framework has been evaluated through the porting of a data mining workflow to COMPSs and the execution on an hybrid testbed.

1.1.3 Data Mining Cloud Framework

Data mining techniques are used in many application areas to extract useful knowledge from large datasets. Very often, parameter sweeping is used in data mining applications to explore the effects produced on the data analysis result by different values of the algorithm parameters. Parameter sweeping applications can be highly computing demanding, since the number of single tasks to be executed increases with the number of swept parameters and the range of their values. Cloud technologies can be effectively exploited to provide end-users with the computing and storage resources, and the execution mechanisms needed to efficiently run this class of applications. In this paper, we present a *Data Mining Cloud Framework* that supports the execution of

parameter sweeping data mining applications on a Cloud. The framework has been implemented using the Windows Azure platform, and evaluated through a set of parameter sweeping clustering and classification applications. The experimental results demonstrate the effectiveness of the proposed framework, as well as the scalability that can be achieved through the parallel execution of parameter sweeping applications on a pool of virtual servers.

1.2 Publications

The following publications have been produced while accomplishing this thesis.

1.2.1 Journals

- F. Marozzo, D. Talia, P. Trunfio, "*P2P-MapReduce: Parallel data processing in dynamic Cloud environments*". Journal of Computer and System Sciences, vol. 78, n. 5, pp. 1382–1402, Elsevier Science, September 2012.

1.2.2 Book Chapters

- F. Marozzo, D. Talia, P. Trunfio, "*A Peer-to-Peer Framework for Supporting MapReduce Applications in Dynamic Cloud Environments*". In: Cloud Computing: Principles, Systems and Applications, N. Antonopoulos, L. Gillam (Editors), Springer, chapt. 7, pp. 113–125, 2010.

1.2.3 Papers in refereed conference proceedings

- F. Marozzo, D. Talia, P. Trunfio, "*A Cloud Framework for Big Data Analytics Workflows on Azure*" Workshop in High Performance Computing, Grids and Clouds (HPC 2012), Cetraro, Italy, June 2012.
- F. Marozzo, F. Lordan, R. Rafanell, D. Lezzi, D. Talia, R. M. Badia, "*Enabling Cloud Interoperability with COMPSs*". Proc. of the 18th International Conference on Parallel and Distributed Computing (Euro-Par 2012), Rhodes Island, Greece, pp. 16–27, Lecture Notes in Computer Science, August 2012.
- F. Marozzo, D. Talia, P. Trunfio, "*Using Clouds for Scalable Knowledge Discovery Applications*" 3rd International Workshop on High Performance Bioinformatics and Biomedicine (Euro-Par 2012 Workshops - HiBB), Rhodes Island, Greece, Lecture Notes in Computer Science, August 2012.
- F. Marozzo, D. Talia, P. Trunfio, "*A Cloud Framework for Parameter Sweeping Data Mining Applications*". Proc. of the 3rd IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2011), Athens, Greece, pp. 367–374, IEEE Computer Society Press, December 2011.

- F. Marozzo, D. Talia, P. Trunfio, "*A Framework for Managing MapReduce Applications in Dynamic Distributed Environments*". Proc. of the 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2011), Ayia Napa, Cyprus, pp. 149–158, IEEE Computer Society Press, February 2011.
- F. Marozzo, D. Talia, P. Trunfio, "*Adapting MapReduce for Dynamic Environments Using a Peer-to-Peer Model*". Proc. of the First Workshop on Cloud Computing and its Applications (CCA 2008), Chicago, USA, October 2008.

1.2.4 Other publications

- F. Marozzo, D. Talia, P. Trunfio, "*Large-Scale Data Analysis on Cloud Systems*". ERCIM News, n. 89, pp. 26–27, April 2012.
- F. Marozzo, F. Lordan, R. Rafanell, D. Lezzi, D. Talia, R. M. Badia, "*Enabling cloud interoperability with COMPSs*". Cloud Futures 2012, Berkeley, California, United States, May 2012.
- F. Marozzo, D. Talia, P. Trunfio, "*Enabling Reliable MapReduce Applications in Dynamic Cloud Infrastructures*". ERCIM News, n. 83, pp. 44-45, October 2010.

1.3 Organization of the Thesis

The remainder of this thesis is organized as follows: Section 2 introduces Cloud computing architecture, service models (SaaS, PaaS, IaaS), systems and applications, in Section 3 presents the P2P-MapReduce system, Section 4 describes the COMPSs for Azure platform, in Section 5 the Data Mining Cloud Framework and some data mining applications executed on it. Finally, conclusions and future work are discussed.

2

Cloud computing

An increasing number of everyday desktop applications are now provided in the form of Web-based applications, or applications available on Internet and accessed through a client like a Web browser or mobile application.

This model is included in the Software as a Service (SaaS) paradigm, in which applications can be accessed through Internet interfaces in the form of "on demand" services, for example through Web Services. SaaS applications, hosted and executed on remote servers, are used to manage business and data activities without end-users having to install, update or maintain them, or own license for the use. Examples of Web-based applications include Web-mails, calendars, document management, image manipulation, and customer relationship management.

Close to the SaaS paradigm are the Platform as a Service (Paas) and Infrastructure as a Service (IaaS) models. PaaS allows the implementation of software applications and Web services by exploiting virtual platforms that offer storage, backup, replication, data protection, security and distributed computing. IaaS allows customers renting resources like CPUs, disks, or more complex like virtualized servers or operating systems to support their operations. All that without bearing the costs of purchasing, maintenance, supporting and updating of IT infrastructure. These "service" models are included in the concept of Cloud computing, a distributed computing paradigm in which resources, dynamically scalable and often virtualized, are provided as a service over Internet. In such model, end-users do not need to have the knowledge or control of the technological infrastructures that support their applications.

Cloud services are based on "Clouds" of computers that act as if they were a single entity and assign computing resources to each application in on-demand mode. In addition, resources are provided in a highly scalable way, by depending on the use of them that is done by users. As a result, what you pay for is what you consume. Cloud systems that support the execution of applications are elastic, i.e., they expand and contract according with the needs of users and, in principle, have no precise boundaries.

Cloud providers can offer their services following three deployment models: public, private or hybrid. A public Cloud provider delivers services available through Internet, with little or no control of end-users over the underlying technology infrastructure. On the other hand, private Cloud providers offer activities and functions "as a service" deployed over an intranet or hosted in remote data centers. Finally, a hybrid Cloud is the composition of two or more Clouds (private or public) that remain different entities but are linked together by offering the benefits of the two deployment models. In this way the private infrastructure can be supplemented with additional computing and storage resources from public Clouds to meet peak demands, better serve user requests and implement high availability strategies.

The remainder of this chapter is organized as follows. Section 2.1 describes the Cloud computing service models (i.e., SaaS, PaaS and IaaS), while Section 2.2 describes Cloud computing deployments models (i.e., public, private and hybrid Cloud). Finally Sections 2.3 and 2.4 introduce Cloud development environments and programming paradigms/frameworks that can be used to implement applications for data processing and knowledge discovery.

2.1 Service models

As mentioned above, Cloud computing vendors provide their services according to three main models: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS).

Software as a Service defines a delivery model in which software and data are provided through Internet to customers as ready-to-use services. Specifically, software and associated data are hosted by providers, and customers access them without needing to use any additional hardware or software. Moreover, customers normally pay a monthly/yearly fee, with no additional purchase of infrastructure or software licenses. Examples of common SaaS applications are Webmails (e.g., Gmail), calendars (Yahoo Calendar), document management (Microsoft Office 365), image manipulation (Photoshop Express), customer relationship management (Salesforce), and others.

In *Platform as a Service* model, Cloud vendors deliver a computing platform typically including databases, application servers, development environment for building, testing and running custom applications. Developers can just focus on deploying of applications since Cloud providers are in charge of maintenance and optimization of the environment and underlying infrastructure. Hence, customers are helped in application development as they use a set of "environment" services that are modular and easily integrable. Normally, the applications are developed as ready-to-use SaaS. Google Apps Engine, Windows Azure, Force.com are some examples of PaaS Cloud environments.

Finally, *Infrastructure as a Service* is an outsourcing model under which customers rent resources like CPUs, disks, or more complex resources like virtualized servers or operating systems to support their operations (e.g., Ama-

zon EC2, RackSpace Cloud). Users of a IaaS have normally skills on system and network administration as they must deal with configuration, operation and maintenance tasks. Compared to the PaaS approach, the IaaS model has a higher system administration costs for the user; on the other hand, IaaS allows a full customization of the execution environment. Developers can scale up or down its services adding or removing virtual machines, easily instantiable from a virtual machine images.

Table 2.1 describes how the three models satisfy the final user requirements, expressed in terms of flexibility, scalability and elasticity, portability, security, maintenance, and costs.

Requirement	*SaaS*	*PaaS*	*IaaS*
Flexibility	Users can customize the application interface and control its behavior, but can not decide which software and hardware components are used to support its execution.	Developers write, customize, test their application using libraries and supporting tools compatible with the platform. Additionally, users can choose what kind of virtual storage and compute resources are used for executing their application.	Developers have to build the servers that will host their applications, and configure operating system and software modules on top of such servers.
Scalability and elasticity	The underlying computing and storage resources normally scale automatically to match application demand, so that users do not have to allocate resources manually. The result depends only on the level of elasticity provided by the Cloud system.	Like the SaaS model, the underlying computing and storage resources normally scale automatically.	Developers can use new storage and compute resources, but their applications must be scalable and allow the dynamic addition of new resources.
Portability	There can be some issues to move applications to other providers, since some software and tools could not work on different systems. For example, application data may not be in a format that can be easily transported to another provider.	Applications can be moved to another provider only if the the new provider share with the old one the required platform tools and services	If a provider allows to download a virtual machine in a standard format, it may be moved to a different provider.
Security	Users can control only some security settings of their applications (e.g., using https instead of http to access some Web pages). Additional security layers (e.g., data replication) are hidden to the user and managed directly by the system.	The security of code and additional libraries used to build application is responsibility of the developer.	Developers must take care security issues from the operating system layer to the user applications.
Maintenance	Users must not carry maintenance tasks.	Developers are in charge of maintaining only their application; other software and the hardware are maintained by the provider.	Developers are in charge of all software, including the operating system; hardware is maintained by the provider.
Cost	Users pay a monthly/yearly fee for using the software, with no additional fee for the infrastructure.	Developers pay for the compute and storage resources, and for the licenses of libraries and tools used by their applications.	Developers pay for all the software and hardware resources used.

Table 2.1: How SaaS, PaaS and IaaS satisfy the user requirements.

2.2 Deployment models

Cloud computing services are delivered according to three deployment models: public, private or hybrid. A *public Cloud* provider delivers services to the general public through the Internet. The users of a public Cloud have little or no control over the underlying technology infrastructure. In this model, services can be offered for free, or provided according to a pay-per-use policy. The main public providers, such as Google, Microsoft, Amazon, own and manage their proprietary data centers delivering services built on top of them. A *private Cloud* provider offers operations and functionalities "as a service", which are deployed over a company intranet or hosted in a remote data center. Often, small and medium-sized IT companies prefer this deployment model as it offers advance security and data control solutions that are not available in the public Cloud model. Finally, a *hybrid Cloud* is the composition of two or more Clouds (private or public) that remain different entities but are linked together. Companies can extend their private Clouds using other private Clouds from partner companies, or public Clouds. In particular, by extending the private infrastructure with public Cloud resources, it is possible to satisfy peaks of requests, better serve user requests, and implement high availability strategies.

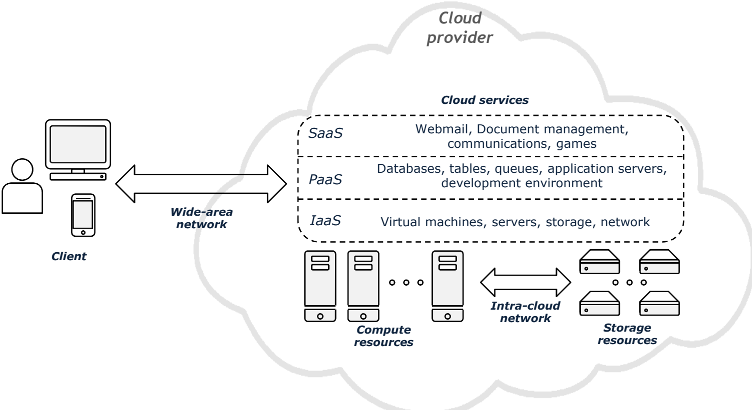


Fig. 2.1: Cloud computing architecture and different delivery models.

Figure 2.1 depicts the general architecture of a public Cloud and its main components, as outlined in [2]. Users access Cloud computing services using *client* devices, such as desktop computers, laptops, tablets and smartphones. Through these devices, users access and interact with Cloud-based services using a Web browser or desktop/mobile app. The business software and user's data are executed and stored on servers hosted in Cloud data centers, that provide *storage and computing resources*. Resources include thousands of servers

and storage devices connected each other through an *intra-Cloud network*. The transfer of data between data center and users takes place on *wide-area network*.

A lot of technologies and standards are used by the different components of the architecture. For example, users can interact with Cloud services through *SOAP* [3] or *REST* [4] Web service standards. *Ajax* [5] and *HTML5* [6] technologies allow Web interfaces to Cloud services to have look and interactivity equivalent to those of desktop applications. *Open Cloud Computing Interface* (OCCI) [7] specifies how Cloud providers can deliver their compute, data, and network resources through a standardized interface. Another example is *Open Virtualization Format* (OVF) [8] for packaging and distributing virtual devices or software (e.g. virtual operating system) to be run on virtual machines.

2.3 Development environments

This section introduces three representative examples of Cloud development environments: Windows Azure as an example of public PaaS, Amazon Web Services as the most popular public IaaS, and OpenNebula as an example of private IaaS. These environment can be used to implement applications and frameworks for data processing and knowledge discovery.

2.3.1 Windows Azure

Azure [9] is an environment and a set of Cloud services that can be used to develop Cloud-oriented applications, or to enhance existing applications with Cloud-based capabilities. The platform provides on-demand compute and storage resources exploiting the computational and storage power of the Microsoft data centers. Azure is designed for supporting high availability and dynamic scaling services that match user needs with a pay-per-use pricing model. The Azure platform can be used to perform the storage of large datasets, execute large volumes of batch computations, and develop SaaS applications targeted towards end-users.

Windows Azure includes three basic components/services as shown in Figure 2.2:

- *Compute* is the computational environment to execute Cloud applications. Each application is structured into roles: *Web role*, for Web-based applications; *Worker role*, for batch applications; *VM role*, for virtual-machine images.
- *Storage* provides scalable storage to manage: binary and text data (*Blobs*), non-relational tables (*Tables*), queues for asynchronous communication between components (*Queues*), and NTFS volume (*Drives*).
- *Fabric controller* whose aim is to build a network of interconnected nodes from the physical machines of a single data center. The Compute and Storage services are built on top of this component.

The Windows Azure platform provides standard interfaces that allow developers to interact with its services. Moreover, developers can use IDEs like Microsoft Visual Studio and Eclipse to easily design and publish Azure applications.

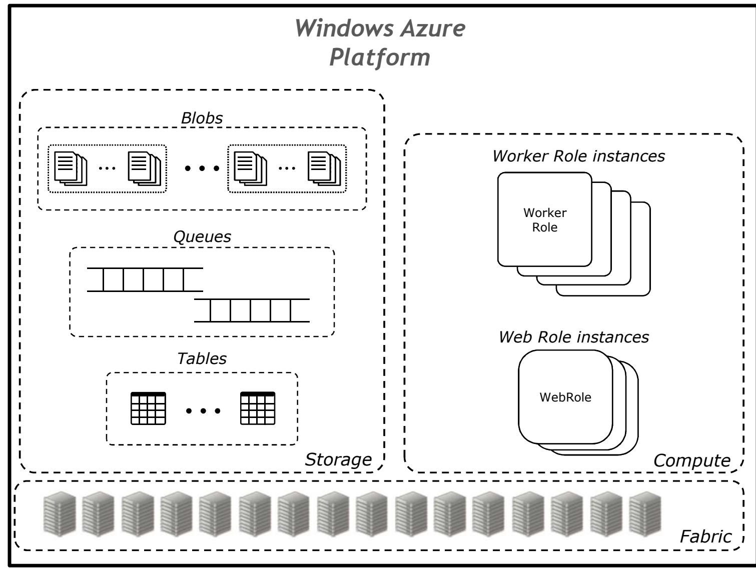


Fig. 2.2: Windows Azure platform.

2.3.2 Amazon Web services

Amazon offers compute and storage resources of its IT infrastructure to developers in the form of Web services. The Amazon Web Services (AWS) [10] is a large set of Cloud services that can be composed by users to build their SaaS applications or integrate traditional software with Cloud capabilities (see Figure 2.3). It is simple to interact with these service since Amazon provides SDKs for the main programming languages and platforms (e.g. Java, .Net, PHP, Android).

AWS includes the following main services:

- Compute: *Elastic Compute Cloud* (EC2) allows creating and running virtual servers; *Amazon Elastic MapReduce* for building and executing MapReduce applications (see 2.4.1).
- Storage: *Simple Storage Service* (S3), which allows storing and retrieving data via the Internet.
- Database: *Relational Database Service* (RDS) for relational tables; *DynamoDB* for non-relational tables; *SimpleDB* for managing small datasets; *ElasticCache* for caching data.

- Networking: *Route 53*, a DNS Web service; *Virtual Private Cloud* for implementing a virtual network.
- Deployment and Management: *CloudFormation* for creating a collection of ready-to-use virtual machines with pre-installed software (e.g., Web applications);
  CloudWatch for monitoring AWS resources; *Elastic Beanstalk* to deploy and execute custom applications written in Java, PHP and other languages; *Identity and Access Management* to securely control access to AWS services and resources.
- Content delivery: *Amazon CloudFront* makes easy to distribute content via a global network of edge locations.
- App services: *Simple Email Service* providing a basic email-sending service; *Simple Notification Service* to notify users; *Simple Queue Service* that implement a message queue; *Simple Workflow Service* to implement workflow-based applications.

Even if Amazon is best known to be the first IaaS provider (based on its EC2 and S3 services), it now also a PaaS provider, with services like Elastic Beans.

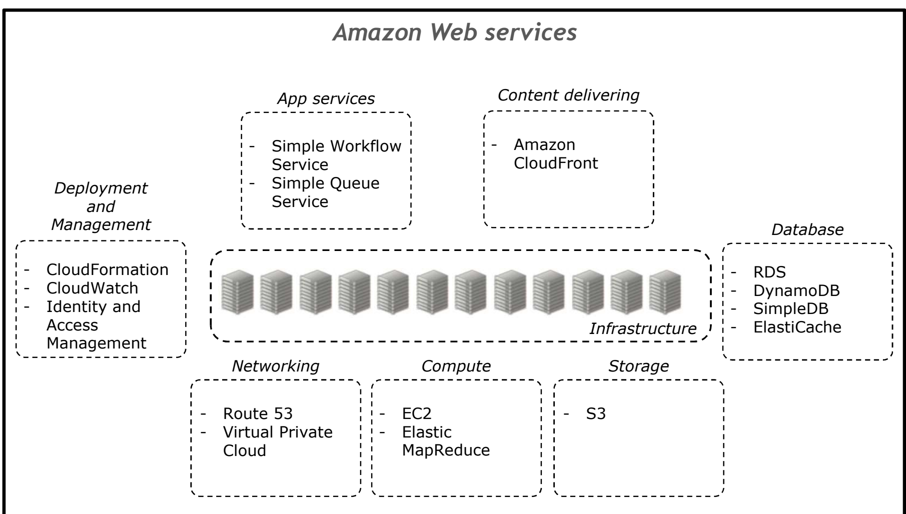


Fig. 2.3: Amazon Web services.

2.3.3 OpenNebula

OpenNebula [11] is an open-source framework mainly used to build private and hybrid Clouds.

The main component of the OpenNebula architecture is the *Core* (see Figure 2.4), which creates and controls virtual machines (i.e., VM) by interconnecting them with a virtual network environment. Moreover, the Core interacts with specific storage, network and virtualization operations through pluggable components called *Drivers*. In this way, OpenNebula is independent from the underlying infrastructure and offers a uniform management environment. The Core also supports the deployment of *Services*, which are a set of linked components (e.g., Web server, database) executed on several VMs. Another component is the *Scheduler*, which is responsible for allocating the VMs on the physical servers. To this end, the Scheduler interacts with the Core component through appropriate deployment commands.

OpenNebula can implement a hybrid Cloud using specific Cloud Drivers that allow to interact with external Clouds. In this way, the local infrastructure can be supplemented with computing and storage resources from public Clouds. Currently, OpenNebula includes drivers for using resources from Amazon EC2 and Eucalyptus [12], another open source Cloud framework.

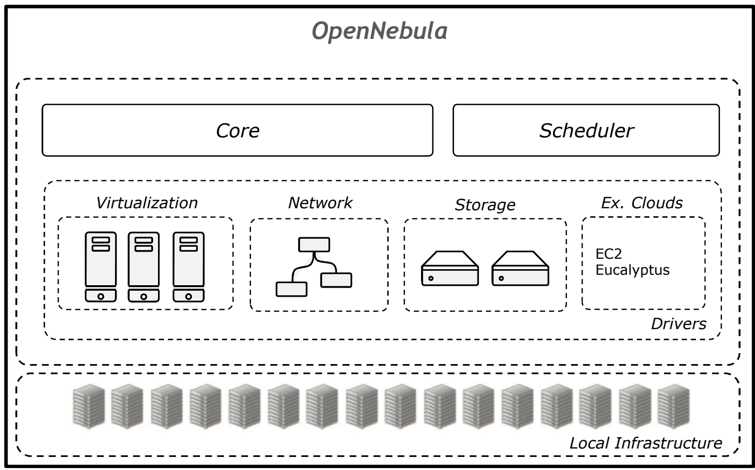


Fig. 2.4: OpenNebula architecture.

2.4 Programming paradigms and frameworks

This section introduces the most known programming paradigms and frameworks used to implement Cloud-based applications for processing and analysis data. Such applications can be categorized into five groups, as outlined in [13]:

- *Map applications*: A "map" operation is performed to each entry of an input dataset. This is often called "embarrassingly parallel" computation.

Some examples are parameter sweeping, BLAST [14] and other protein applications, multi-gene analyses.

- *MapReduce applications*: A MapReduce application is defined in terms of a map function that processes a (key,value) pair to generate a list of intermediate (key, value) pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Common examples are algorithms for distributed searching and sorting on large textual data sources.
- *Iterative MapReduce applications*: Some parallel algorithms are structured as executions of MapReduce applications inside an iterative loop. Examples are the expectation maximization clustering algorithm, linear algebra algorithms, or Page Rank.
- *Task-based applications*: Applications are defined as a set of tasks executed by a set of workers.
- *MPI applications*: Message Passing Interface standard is used to execute parallel applications, e.g. for solving differential equations or for simulating particle dynamics.

In the following some examples of programming paradigms and frameworks are discussed.

2.4.1 MapReduce

MapReduce [15] is a programming model inspired by the *map* and *reduce* primitives present in Lisp and other functional languages. A user defines a MapReduce application in terms of a map function that processes a (key, value) pair to generate a list of intermediate (key, value) pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Most MapReduce implementations, like Hadoop [16], are based on a master-slave architecture. A job is submitted by a user node to a master node that selects idle workers and assigns a map or reduce task to each one. When all the tasks have been completed, the master node returns the result to the user node. The MapReduce paradigm is appropriate to implement data mining tasks in parallel. An example is Disco [17], a framework built on top of Hadoop for data pre-processing and co-clustering. Other relevant examples are the use of MapReduce for K-Means clustering [18], and to run a semi-supervised classification on large scale graphs [19].

2.4.2 COMPSs

COMPSs [20] is a programming framework, composed of a programming model and an execution runtime which supports it, whose main objective is to ease the development of applications for distributed environments.

On the one hand, the programming model aims to keep the programmers unaware of the execution environment and parallelization details. They are

only required to create a sequential application and specify which methods of the application code will be executed remotely. This selection is done by providing an annotated interface where these methods are declared with some metadata about them and their parameters. On the other hand, the runtime is in charge of optimizing the performance of the application by exploiting its inherent concurrency. The runtime intercepts any call to a selected method creating a representative task and finding the data dependencies with all the previous ones that must be considered along the application run. The task is added to a task dependency graph as a new node and such dependencies are represented by edges of the graph. Tasks with no dependencies enter the scheduling step and are assigned to available resources. This decision is made according to a scheduling algorithm that takes into account data locality, task constraints and the workload of each node. According to this decision the input data for the scheduled task are transferred to the selected host and the task is remotely submitted. Once a task finishes, the task dependency graph is updated, possibly resulting in new dependency-free tasks that can be scheduled.

2.4.3 Sector/Sphere

Sector/Sphere [21] is a framework designed to run data analysis applications on large distributed datasets. It consists of two complementary components: a storage component (*Sector*) and a compute component (*Sphere*). Sector provides a long term archival storage to access and index large distributed datasets. It is designed to support different types of network protocols, and to safely archive data through replication mechanisms. Sphere enables the parallel execution of user-defined functions on data stored in Sector. It splits the input files into data segments that are processed in parallel by servers called Sphere processing elements. A data segment can be a single entry record, a collection of data records or a file. An evaluation of Sector/Sphere for executing data analysis applications on a wide area is reported in [22].

2.4.4 All-Pairs

All-Pairs [23] is a programming model and a framework to implement data intensive scientific applications on a cluster or a Cloud. The framework can be used for applications that have to compare the elements of two datasets on the basis of a user-defined comparison function. The user defines the problem to be solved through a specification, called *abstraction*; the framework includes an *engine* that chooses how to implement the specification using the available resources. The engine partitions and transfer data to a set of disks in the cluster, and then dispatches batch jobs to execute locally on each data split.

2.4.5 Dryad

Dryad [24] is a Microsoft framework to run data-parallel applications on a cluster or a data center. A Dryad application combines computational *vertices* with communication *channels* to form a dataflow graph. The application runs by executing the vertices of the graph on a set of available computers and communicating as appropriate through files, TCP pipes, and shared-memory. The users define sequential programs for each vertex, and the framework schedules and executes them in parallel on multiple CPU cores or computers. Dryad has been used in combination with LINQ (a language for adding data querying capabilities to .NET languages) to implement a set of data analysis applications, including a data clustering application based on the K-Means algorithm [25].

2.4.6 Aneka

Aneka [26] platform provides a framework for the development of application supporting not only the MapReduce programming model but also a Task Programming and Thread Programming ones. The applications can be deployed on private or public Clouds such as Windows Azure, Amazon EC2, and GoGrid Cloud Service. The user has to use a specific .NET SDK for the porting of the code also to enact legacy code.

3

P2P-MapReduce

MapReduce is a system and method for efficient large-scale data processing presented by Google in 2004 [15] to cope with the challenge of processing very large input data generated by Internet-based applications. Since its introduction, MapReduce has proven to be applicable to a wide range of domains, including machine learning and data mining, log file analysis, financial analysis, scientific simulation, image retrieval and processing, blog crawling, machine translation, language modelling, and bioinformatics. Today, MapReduce is widely recognized as one of the most important programming models for Cloud computing environments, being it supported by leading Cloud providers such as Amazon, with its Elastic MapReduce service [27], and Google itself, which recently released a Mapper API for its App Engine [28].

The MapReduce abstraction is inspired by the *map* and *reduce* primitives present in Lisp and other functional languages [29]. A user defines a MapReduce application in terms of a map function that processes a (key, value) pair to generate a list of intermediate (key, value) pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Current MapReduce implementations, like Google's MapReduce [30] and Hadoop [16], are based on a master-slave architecture. A job is submitted by a user node to a master node that selects idle workers and assigns a map or reduce task to each one. When all the tasks have been completed, the master node returns the result to the user node. The failure of one worker is managed by re-executing its task on another worker, while master failures are not explicitly managed as designers consider failures unlikely in reliable computing environments, such as a data center or a dedicated Cloud.

On the contrary, node churn and failures - including master failures - are likely in large dynamic Cloud environments, like a Cloud of Clouds, which can be formed by a large number of computing nodes that join and leave the network at very high rates. Therefore, providing effective mechanisms to manage such problems is fundamental to enable reliable MapReduce applications in dynamic Cloud infrastructures, where current MapReduce middleware could be unreliable. We have designed an adaptive MapReduce framework, called

P2P-MapReduce, which exploits a peer-to-peer model to manage node churn, master failures, and job recovery in a decentralized but effective way, so as to provide a more reliable MapReduce middleware that can be effectively exploited in dynamic Cloud infrastructures.

This chapter describes the P2P-MapReduce system providing a detailed description of its basic mechanisms, a prototype implementation, and an extensive performance evaluation in different network scenarios. The experimental results show that, differently from centralized master-server implementations, the P2P-MapReduce framework does not suffer from job failures even in presence of very high churn rates, thus enabling the execution of reliable MapReduce applications in very dynamic Cloud infrastructures. In an early version of this work [31] we presented a preliminary architecture of the P2P-MapReduce framework, while in a more recent paper [32] we introduced its main software modules and a preliminary evaluation. This chapter significantly extends our previous work by providing a detailed description of the mechanisms at the base of the P2P-MapReduce system, as well as an extensive evaluation of its performance in different scenarios.

The remainder of this chapter is organized as follows. Section 3.1 provides a background on the MapReduce programming model and discusses related work. Section 3.2 introduces the system model and presents the general architecture of the P2P-MapReduce framework. System 3.3 describes the system mechanisms, while Section 3.4 discusses its implementation. Section 3.5 evaluates the performance of P2P-MapReduce compared to a centralized implementation of MapReduce. Finally, Section 3.6 concludes the chapter.

3.1 Background and Related Work

This section provides a background on the MapReduce programming model and discusses related work.

3.1.1 The MapReduce Programming Model

As mentioned before, MapReduce applications are based on a master-slave model. This section briefly describes the various operations that are performed by a generic application to transform input data into output data according to that model.

Users define a *map* and a *reduce* function [29]. The *map* function processes a (key, value) pair and returns a list of intermediate (key, value) pairs:

map (k1,v1) $\rightarrow$ list(k2,v2).

The *reduce* function merges all intermediate values having the same intermediate key:

reduce (k2, list(v2)) $\rightarrow$ list(v3).

As an example, let's consider the creation of an inverted index for a large set of Web documents [15]. In its basic form, an inverted index contains a set of words (index terms), and for each word it specifies the IDs of all the documents which contain that word. Using a MapReduce approach, the *map* function parses each document and emits a sequence of (word, documentID) pairs. The *reduce* function takes all pairs for a given word, sorts the corresponding document IDs, and emits a (word, list(documentID)) pair. The set of all output pairs generated by the *reduce* function forms the inverted index for the input documents.

In general, the whole transformation process performed in a MapReduce application can be described through the following steps (see Figure 3.1):

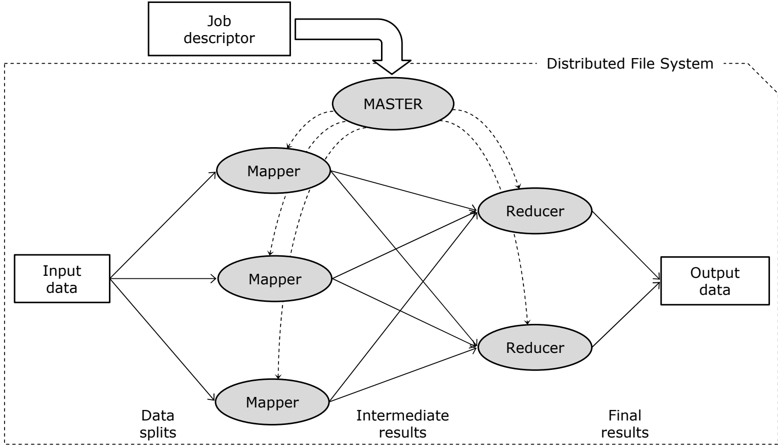


Fig. 3.1: Execution phases of a generic MapReduce application.

1. A master process receives a job descriptor which specifies the MapReduce job to be executed. The job descriptor contains, among other information, the location of the input data, which may be accessed using a distributed file system or an HTTP/FTP server.
2. According to the job descriptor, the master starts a number of mapper and reducer processes on different machines. At the same time, it starts a process that reads the input data from its location, partitions that data into a set of splits, and distributes those splits to the various mappers.
3. After receiving its data partition, each mapper process executes the *map* function (provided as part of the job descriptor) to generate a list of intermediate key/value pairs. Those pairs are then grouped on the basis of their keys.
4. All pairs with the same keys are assigned to the same reducer process. Hence, each reducer process executes the *reduce* function (defined by the

job descriptor) which merges all the values associated to the same key to generate a possibly smaller set of values.
5. The results generated by each reducer process are then collected and delivered to a location specified by the job descriptor, so as to form the final output data.

Distributed file systems are the most popular solution for accessing input/output data in MapReduce systems, particularly for standard computing environments like a data center or a cluster of computers. On the other hand, distributed file systems may be ineffective in large-scale dynamic Cloud environments characterized by high levels of churn. Therefore, we assume that data are moved across nodes using a file transfer protocol like FTP or HTTP as done, for example, by the MISCO MapReduce Framework [33].

3.1.2 Related Work

Besides the original MapReduce implementation by Google [30], several other MapReduce implementations have been realized within other systems, including Hadoop, GridGain [34], Skynet [35], MapSharp [36], Twister [37], and Disco [38]. Another system sharing most of the design principles of MapReduce is Sector/Sphere [21], which has been designed to support distributed data storage and processing over large Cloud systems. Sector is a high-performance distributed file system; Sphere is a parallel data processing engine used to process Sector data files.

Some other works focused on providing more efficient implementations of MapReduce components, such as the scheduler [39] and the I/O system [40], while others focused on adapting the MapReduce model to specific computing environments, like shared-memory systems [41], volunteer computing environments [42], desktop Grids [43], and mobile environments [33].

Zaharia et al. [39] studied how to improve the Hadoop's scheduler in heterogeneous environments, by designing a new scheduling algorithm, called LATE, which significantly improves response times in heterogeneous settings. The LATE algorithm uses estimated finish times to efficiently schedule speculative copies of tasks (also called "backup" tasks in MapReduce terminology) to finish the computation faster. The main policy adopted by LATE is to speculatively execute the task that is thought to finish farthest into the future.

The Hadoop Online Prototype (HOP) [40] modifies the Hadoop MapReduce framework to supports online aggregation, allowing users to see early returns from a job as it is being computed. HOP also supports continuous queries, which enable MapReduce programs to be written for applications such as event monitoring and stream processing. HOP extends the applicability of the MapReduce model to pipelining behaviors, which is useful for batch processing. In fact, by pipelining both within and across jobs, HOP can reduce the time to job completion.

Phoenix [41] is an implementation of MapReduce for shared-memory systems that includes a programming API and a runtime system. Phoenix uses

threads to spawn parallel map or reduce tasks. It also uses shared-memory buffers to facilitate communication without excessive data copying. The runtime schedules tasks dynamically across the available processors in order to achieve load balance and maximize task throughput. Overall, Phoenix proves that MapReduce is a useful programming and concurrency management approach also for multi-core and multi-processor systems.

MOON [42] is a system designed to support MapReduce jobs on opportunistic environments. MOON, extends Hadoop with adaptive task and data scheduling algorithms to offer reliable MapReduce services on a hybrid resource architecture, where volunteer computing systems are supplemented by a small set of dedicated nodes. The adaptive task and data scheduling algorithms in MOON distinguish between different types of MapReduce data and different types of node outages in order to place tasks and data on both volatile and dedicated nodes.

Another system that shares some of the key ideas behind MOON is that proposed by Tang et al. [43]. The system is specifically designed to support MapReduce applications in Desktop Grids, and exploits the BitDew middleware [44] which is a programmable environment for automatic and transparent data management on Desktop Grids. BitDew relies on a specific set of metadata to drive key data management operations, namely life cycle, distribution, placement, replication and fault-tolerance with a high level of abstraction.

Finally, Misco [33] is a framework for supporting MapReduce applications on mobile systems. Although Misco follows the general design of MapReduce, it does vary in two main aspects: task assignment and data transfer. The first aspect is managed through the use of a polling strategy. Each slave polls the master each time it becomes available. If there are no tasks to execute, the slave will idle for a period of time before requesting a task again. For data transfer, instead of a distributed file system that is not practical in a mobile scenario, Misco uses HTTP to communicate requests, task information and transfer data.

Even though P2P-MapReduce shares some basic ideas with some of the systems discussed above (in particular, [42, 43, 33]), it also differs from all of them for its use of a peer-to-peer approach both for job and system management. Indeed, the peer-to-peer mechanisms described in Section 3.3 allows nodes to dynamically join and leave the network, change state over time, manage nodes and job failures in a way that is completely transparent both to users and applications.

3.2 System Model and Architecture

As mentioned before, the goal of P2P-MapReduce is to enable a reliable execution of MapReduce applications in Cloud environments characterized by high levels of churn. To achieve this goal, P2P-MapReduce adopts a peer-to-peer model in which a wide set of autonomous nodes (peers) can act either

as a master or a slave. At each time, a limited set of nodes is assigned the master role, while the others are assigned the slave role. The role assigned to a given node can change dynamically over time, so as to ensure the presence of the desired master/slave ratio for reliability and load balancing purposes.

To prevent loss of work in the case of a master failure, each master can act as a backup for other masters. The master responsible for a job J, referred to as the *primary master* for J, dynamically updates the job state (e.g., the assignments of tasks to nodes, the status of each task, etc.) on its backup nodes, which are referred to as the *backup masters* for J. To prevent excessive overhead, the update does not contain whole job information, but only that part of information that has updated. If a primary master fails (or, equivalently, it abruptly leaves the network), its place is taken by one of its backup masters in a way that is transparent both to the user who submitted the job, and to the job itself.

The overall system behavior, as well as its features (resilience to failures, load balancing), are the result of the behavior of each single node in the system. The node behavior will be described in detail in Section 3.3 as a state diagram that defines the different states that a node can assume, and all the events that determine the transitions from state to state. The remainder of this section describes the system model and the general architecture of the P2P-MapReduce framework.

3.2.1 System Model

The model introduced here provides abstractions for describing the characteristics of jobs, tasks, users, and nodes. For the reader's convenience, Figure 3.2 illustrates the system model entities and their interrelationships using the UML Class Diagram formalism.

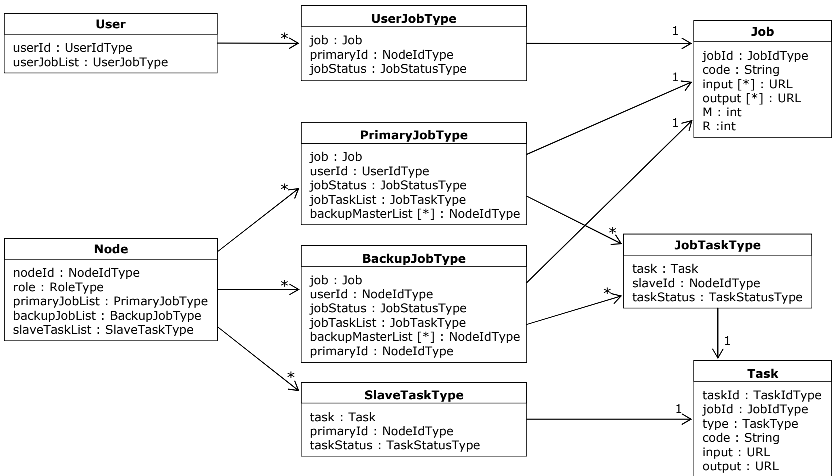


Fig. 3.2: System model described through the UML Class Diagram formalism.

Jobs and tasks

A *job* is modelled as a tuple of the form:

$$job = \langle jobId, code, input, output, M, R \rangle$$

where *jobId* is a job identifier, *code* includes the map and reduce functions, *input* (resp., *output*) is the location of the input (output) data of the job, M is the number of map tasks, and R is the number of reduce tasks.

A *task* is modelled as a tuple:

$$task = \langle taskId, jobId, type, code, input, output \rangle$$

where *taskId* is a task identifier, *jobId* is the identifier of the job the task belongs to, *type* can be either `MAP` or `REDUCE`, *code* includes the map or reduce function (depending on the task type), and *input* (*output*) is the location of the input (output) data of the task.

Users and nodes

A *user* is modelled as a pair of the form:

$$user = \langle userId, userJobList \rangle$$

which contains the user identifier (*userId*) and the list of jobs submitted by the user (*userJobList*). The *userJobList* contains tuples of a *userJobType* defined as:

$$userJobType = \langle job, primaryId, jobStatus \rangle$$

where *job* is a job descriptor, *primaryId* is the identifier of the node that is managing the job as the primary master, and *jobStatus* represents the current status of the job.

A *node* is modelled as a tuple:

$$node = \langle nodeId,\ role,\ primaryJobList,\ backupJobList,\ slaveTaskList \rangle$$

which contains the node identifier (*nodeId*), its *role* (`MASTER` or `SLAVE`), the list of jobs managed by this node as the primary master (*primaryJobList*), the list of jobs managed by this node as a backup master (*backupJobList*), and the list of tasks managed by this node as a slave (*slaveTaskList*). Note that *primaryJobList* and *backupJobList* are empty if the node is currently a slave, while *slaveTaskList* is empty if the node is acting as a master.

The *primaryJobList* contains tuples of a *primaryJobType* defined as:

$$primaryJobType = \langle job,\ userId,\ jobStatus,\ jobTaskList,\ backupMasterList \rangle$$

where *job* is a job descriptor, *userId* is the identifier of the user that has submitted the job, *jobStatus* is the current status of the job, *jobTaskList* is a list containing dynamic information about the tasks that compose the job, and *backupMasterList* is a list containing the identifiers (*backupId*) of the backup masters assigned to the job. The *jobTaskList* contains tuples of a *jobTaskType*, which is defined as follows:

$$jobTaskType = \langle task,\ slaveId,\ taskStatus \rangle$$

where *task* is a task descriptor, *slaveId* is the identifier of the slave responsible for the task, and *taskStatus* represents the current status of the task.

The *backupJobList* contains tuples of a *backupJobType* defined as:

$$backupJobType = \langle job,\ userId,\ jobStatus,\ jobTaskList,\ backupMasterList,\ primaryId \rangle$$

that differs from *primaryJobType* for the presence of an additional field, *primaryId*, which represents the identifier of the primary master associated to the job.

Finally, the *slaveTaskList* contains tuples of a *slaveTaskType*, which is defined as:

$$slaveTaskType = \langle task,\ primaryId,\ taskStatus \rangle$$

where *task* is a task descriptor, *primaryId* is the identifier of the primary master associated to the task, and *taskStatus* contains its status.

3.2.2 Architecture

The P2P-MapReduce architecture includes three types of nodes, as shown in Figure 3.3: *user*, *master* and *slave*. Master nodes and slave nodes form two logical peer-to-peer networks referred to as *M-net* and *S-net*, respectively. As mentioned earlier in this section, computing nodes are dynamically assigned

the master or the slave role, thus *M-net* and *S-Net* change their composition over time. The mechanisms used for maintaining the infrastructure will be described in Section 3.3.

User nodes submit their MapReduce jobs to the system through one of the available masters. The choice of the master to which submit the job may be done on the basis of the current workload of the available masters, i.e., the user may choose the master that is managing the lowest number of jobs.

Master nodes are at the core of the system. They perform three types of operations: management, recovery and coordination. Management operations are those performed by masters that are acting as the *primary master* for one or more jobs. Recovery operations are executed by masters that are acting as *backup master* for one or more jobs. Coordination operations are performed by the master that is acting as the network *coordinator*. The coordinator has the power of changing slaves into masters, and viceversa, so as to keep the desired master/slave ratio.

Each slave executes the tasks that are assigned to it by one or more primary masters. Task assignment may follow various policies, based on current workload, highest reliability, and so on. In our implementation tasks are assigned to the slaves with the lowest workload, i.e., with the lowest number of assigned tasks.

Jobs and tasks are managed by processes called *Job Managers* and *Task Managers*, respectively. Each primary master runs one Job Manager thread per managed job, while each slave runs one Task Manager thread per managed task. Moreover, masters use a *Backup Job Manager* for each job they are responsible for as backup masters.

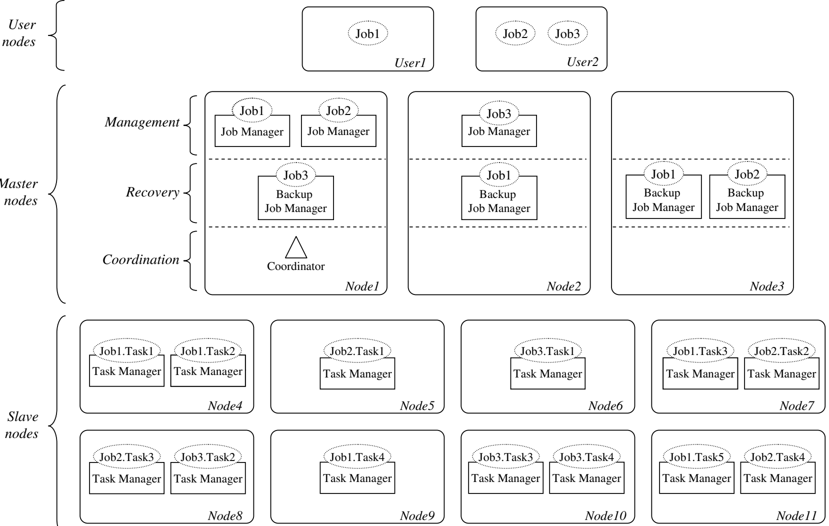


Fig. 3.3: General architecture of P2P-MapReduce.

Figure 3.3 shows an example scenario in which three jobs have been submitted: one job by *User1* (*Job1*) and two jobs by *User2* (*Job2* and *Job3*). Focusing on *Job1*, *Node1* is the primary master, and two backup masters are used (*Node2* and *Node3*). *Job1* is composed by five tasks: two of them are assigned to *Node4*, and one each to *Node7*, *Node9* and *Node11*.

If the primary master *Node1* fails before the completion of *Job1*, the following recovery procedure takes place:

- Backup masters *Node2* and *Node3* detect the failure of *Node1* and start a distributed procedure to elect the new primary master among them.
- Assuming that *Node3* is elected as the new primary master, *Node2* continues to play the backup function and, to keep the desired number of backup masters active (two, in this example), another backup node is chosen by *Node3*. Then, *Node3* binds to the connections that were previously associated to *Node1*, and proceeds to manage the job using its local replica of the job state.

As soon as the job is completed, the (new) primary master notifies the result to the user node that submitted the managed job.

The system mechanisms sketched above are described in detail in Section 3.3, while Section 3.4 will provide a description of the system implementation.

3.3 System Mechanisms

The behavior of a generic node is modelled as a state diagram that defines the different states that a node can assume, and all the events that determine the transitions from a state to another state. Figure 3.4 shows such state diagram modelled using the UML State Diagram formalism.

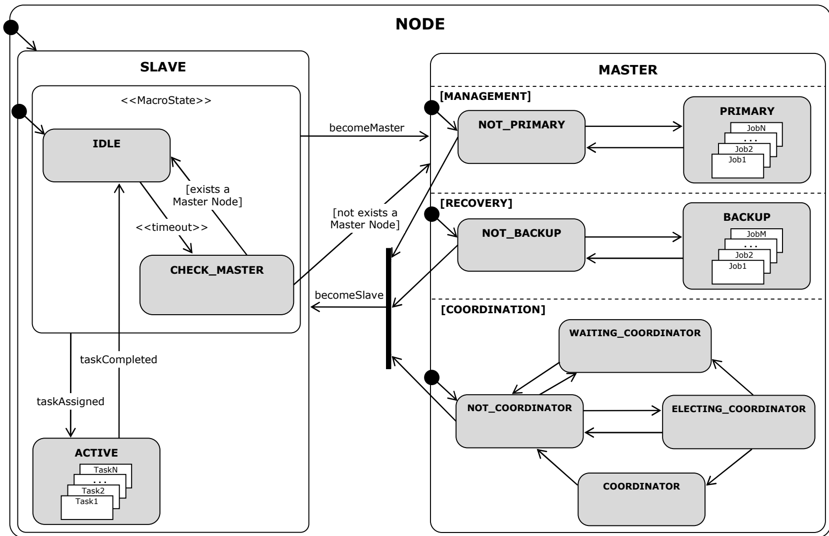


Fig. 3.4: Behavior of a generic node described by an UML State Diagram.

The state diagram includes two macro-states, `SLAVE` and `MASTER`, which describe the two roles that can be assumed by each node. The `SLAVE` macro-state has three states, `IDLE`, `CHECK_MASTER` and `ACTIVE`, which represent respectively: a slave waiting for task assignment; a slave checking the existence of at least one master in the network; a slave executing one or more tasks. The `MASTER` macro-state is modelled with three parallel macro-states, which represent the different roles a master can perform concurrently: possibly acting as the primary master for one or more jobs (`MANAGEMENT`); possibly acting as a backup master for one or more jobs (`RECOVERY`); coordinating the network for maintenance purposes (`COORDINATION`).

The `MANAGEMENT` macro-state contains two states: `NOT_PRIMARY`, which represents a master node currently not acting as the primary master for any job, and `PRIMARY`, which, in contrast, represents a master node currently managing at least one job as the primary master. Similarly, the `RECOVERY` macro-state includes two states: `NOT_BACKUP` (the node is not managing any job as backup master) and `BACKUP` (at least one job is currently being backed up on this node). Finally, the `COORDINATION` macro-state includes

four states: `NOT_COORDINATOR` (the node is not acting as the coordinator), `COORDINATOR` (the node is acting as the coordinator), `WAITING_COORDINATOR` and `ELECTING_COORDINATOR` for nodes currently participating to the election of the new coordinator, as specified later.

The combination of the concurrent states [`NOT_PRIMARY`, `NOT_BACKUP`, `NOT_COORDINATOR`] represents the abstract state `MASTER.IDLE`. The transition from master to slave role is allowed only to masters in the `MASTER.IDLE` state. Similarly, the transition from slave to master role is allowed to slaves that are not in `ACTIVE` state.

The events that determine state transitions are shown in Table 3.1. For each event message the table shows: the event parameters; whether it is an inner event; the sender's state; the receiver's state.

In the following we describe in detail the algorithmic behavior of each node, using Table 3.1 as a reference for the events that are exchanged among system entities. We proceed as follows. Section 3.3.1 describes data structures and high-level behaviors of slave and master nodes. Section 3.3.2 focuses on job and tasks management, by describing the algorithms that steer the behavior of Job Managers, Tasks Managers, and Backup Job Managers. Finally, Section 3.3.3 describes the behavior of user nodes.

Event	Parameters	Inner event	Sender's state	Receiver's state
becomeMaster		no	`COORDINATION`	`SLAVE`
becomeSlave		no	`COORDINATION`	`MANAGEMENT`
jobAssigned	Job, UserIdType	no	`USER`	`MANAGEMENT`
jobReassigned	BackupJobType	yes	`BACKUP_JOB_MANAGER`	`MANAGEMENT`
jobCompleted	JobIdType, JobStatusType	yes no	`JOB_MANAGER` `MANAGEMENT`	`MANAGEMENT` `USER`
backupJobAssigned	PrimaryJobType, NodeIdType	no	`JOB_MANAGER`	`RECOVERY`
jobUpdate	JobIdType, List<NodeIdType>, List<JobTaskType>	no	`JOB_MANAGER`	`RECOVERY`
backupJobCompleted	JobIdType	yes no	`BACKUP_JOB_MANAGER` `MANAGEMENT`	`RECOVERY` `RECOVERY`
taskAssigned	List<Task>, NodeIdType	no	`JOB_MANAGER`	`SLAVE`
taskCompleted	TaskIdType, TaskStatusType	yes no	`TASK_MANAGER` `SLAVE`	`SLAVE` `MANAGEMENT`

primaryUpdate	JobIdType,
	NodeIdType
		no
		no
		yes
		no
		yes
			`MANAGEMENT`
			`MANAGEMENT`
			`RECOVERY`
			`MANAGEMENT`
			`SLAVE`
				`USER`
				`RECOVERY`
				`BACKUP_JOB_MANAGER`
				`SLAVE`
				`TASK_MANAGER`

Table 3.1: Description of the event messages that can be exchanged by P2P-MapReduce nodes

3.3.1 Node Behavior

Figure 3.5 describes the behavior of a generic node, by specifying the algorithmic details behind the finite state machine depicted in Figure 3.4. Each node includes five fields, already introduced in Section 3.2.1: *nodeId*, *role*, *primaryJobList*, *backupJobList* and *slaveTaskList*.

As soon as a node joins the network, it transits to the `SLAVE` macro-state (*line 8*). Then, it sets its role accordingly (*line 12*) and passes to the `IDLE` state (*line 13*). When a slave is in `IDLE` state, three events can happen:

- An internal timeout elapses, causing a transition to the `CHECK_MASTER` state (*line 17*).
- A *taskAssigned* message is received from the Job Manager of a primary master (*line 19*). The message includes a list with the tasks to be executed (*assignedTaskList*), and the identifier of the primary master (*primaryId*). For each assigned task, a new *slaveTask* is created, it is added to the *slaveTaskList*, and associated with a *TaskManager* (lines 20-24). Then, the slave passes to the `ACTIVE` state (*line 25*).
- A *becomeMaster* message is received from the coordinator, causing a transition to the `MASTER` state (*lines 27-28*).

When a slave is in the `CHECK_MASTER` state, it queries the discovery service to check the existence of at least one master in the network. In case no masters are found, the slave promotes itself to the master role; otherwise, it returns to the `IDLE` state (*lines 33-36*). This state allows the first node joining (and the last node remaining into) the network to assume the master role. A node in `CHECK_MASTER` state can also receive *taskAssigned* and *becomeMaster* events, which are managed as discussed earlier.

```
1  StateDiagram NODE {
2   NodeIdType nodeId;
3   RoleType role;
4   List<PrimaryJobType> primaryJobList;
5   List<BackupJobType> backupJobList;
6   List<SlaveTaskType> slaveTaskList;
7   entry() {
8    transition to SLAVE;
9   }
10  Sequential Macro-State SLAVE {
11   entry() {
12    role = SLAVE;
13    transition to IDLE;
14   }
15   State IDLE {
16    do() {
17     transition to CHECK_MASTER after a timeout;
18    }
19    on event taskAssigned(List<Task>
          assignedTaskList, NodeIdType primaryId) {
20     foreach(Task t in assignedTaskList){
21      SlaveTaskType st(task, primaryId);
22      add st to slaveTaskList;
23      start TaskManager for st;
24     }
25     transition to ACTIVE;
26    }
27    on event becomeMaster() {
28     transition to MASTER;
29    }
30   }
31   State CHECK_MASTER {
32    do() {
33     if(exists a master)
34      transition to IDLE;
35     else
36      transition to MASTER;
37    }
38    on event taskAssigned(List<Task>
          assignedTaskList, NodeIdType primaryId) {
39     //same as lines 20 to 25
40    }
41    on event becomeMaster() {
42     transition to MASTER;
43    }
44   }
45   State ACTIVE {
46    on event taskAssigned(List<Task>
          assignedTaskList, NodeIdType primaryId) {
47     //same as lines 20 to 24
48    }
49    on event primaryUpdate(JobIdType
          updatedJobId, NodeIdType
          updatedPrimaryId) {
50     foreach(SlaveTaskType t in slaveTaskList) {
51      if(t.task.jobId = updatedJobId){
52       t.primaryId = updatedPrimaryId;
53       propagate event to TaskManager for t;
54      }
55     }
56    }
57    on inner event taskCompleted(TaskIdType
          completedTaskId, TaskStatusType
          completedTaskStatus) {
58     SlaveTaskType t = tuple in slaveTaskList
           with task.taskId=completedTaskId;
59     stop taskManager for t;
60     propagate event to t.primaryId;
61     if(slaveTaskList does not contain tasks to
           execute)
62      transition to IDLE;
63    }
64   }
65  }
66  Concurrent Macro-State MASTER {
67   entry() {
68    role = MASTER;
69    concurrent transition to MANAGEMENT, RECOVERY
          and COORDINATION;
70   }
71   on event becomeSlave() {
72    if(MANAGEMENT in NOT_PRIMARY and RECOVERY in
          NOT_BACKUP and COORDINATION in
          NOT_COORDINATOR)
73     transition to SLAVE;
74   }
75   Sequential Macro-State MANAGEMENT {
76    //see Figure 6
77   }
78   Sequential Macro-State RECOVERY {
79    //see Figure 7
80   }
81   Sequential Macro-State COORDINATION {
82    //see Figure 8
83   }
84  }
85 }
```

Fig. 3.5: Pseudo-code describing the behavior of a generic node.

Slaves in `ACTIVE` state can receive three events:

- *taskAssigned*, which adds other tasks to those already being executed (*line 46*).
- *primaryUpdate*, which informs the slave that the primary master for a given job has changed (*line 49*). This message, sent by the new primary master, includes the identifier of the job whose primary master has changed (*updatedJobId*), and the identifier of the new primary master (*updatedPrimaryId*). Each task in the *slaveTaskList* whose job identifier equals *updatedJobId* is updated accordingly, and the associated Task Manager is notified by propagating the event to it (lines 50-55).
- *taskCompleted*, an inner event sent by the Task Manager of the same node, on completion of a given task (*line 57*). The event includes the identifier of the completed task (*completedTaskId*) and its status (*completedTaskStatus*). The slave identifies the corresponding tasks in the *slaveTaskList*, stops the corresponding Task Manager, and propagates the event to the primary master (*lines 58-60*). If there are no more tasks being executed, the slave transits to the `IDLE` state.

A node that is promoted to the `MASTER` macro-state sets its role accordingly (*line 68*), and concurrently transits to three parallel states (*line 69*),

as depicted in Figure 3.4 and mentioned earlier in this section: `MANAGEMENT`, `COORDINATION` and `RECOVERY`. If a master receives a *becomeSlave* event from the coordinator (*line 71*), and the internal states of the `MANAGEMENT`, `COORDINATION` and `RECOVERY` concurrent macro-states are respectively equal to `NOT_PRIMARY`, `NOT_BACKUP` and `NOT_COORDINATOR`, it transits to the `SLAVE` state (*lines 72-73*).

```
1  Sequential Macro-State MANAGEMENT {
2   entry() {
3    transition to NOT_PRIMARY;
4   }
5   State NOT_PRIMARY {
6    on event jobAssigned(Job job, UserIdType
          userId) {
7     PrimaryJobType t(job, userId);
8     add t to primaryJobList;
9     start jobManager for t;
10    transition to PRIMARY;
11   }
12   on inner event jobReassigned(BackupJobType bt)
          {
13    PrimaryJobType pt(bt.job, bt.userId,
          bt.jobStatus, bt.jobTaskList,
          bt.backupMasterList);
14    add pt to primaryJobList;
15    foreach(NodeIdType b in pt.backupMasterList)
16     send event primaryUpdate(pt.job.jobId,
          nodeId) to b;
17    foreach(JobTaskType t in pt.jobTaskList)
18     send event primaryUpdate(pt.job.jobId,
          nodeId) to t.slaveId;
19    send event primaryUpdate(pt.job.jobId,
          nodeId) to pt.userId;
20    start jobManager for pt;
21    transition to PRIMARY;
22   }
23  }
24  State PRIMARY {
25   on event jobAssigned(Job job, UserIdType
          userId) {
26    //same as lines 7 to 9
27   }
28   on inner event jobReassigned(BackupJobType bt)
          {
29    //same as lines 13 to 20
30   }
31   on inner event jobCompleted(JobIdType
          completedJobId, JobStatusType
          completedJobStatus) {
32    PrimaryJobType t = tuple in PrimaryJobList
          with job.jobId=completedJobId;
33    stop jobManager for t;
34    foreach(NodeIdType b in t.backupMasterList)
35     send event
          backupJobCompleted(completedJobId) to b;
36    propagate event to t.userId;
37    if(primaryJobList does not contain jobs to
          execute)
38     transition to NOT_PRIMARY;
39   }
40  }
41 }
```

Fig. 3.6: Pseudo-code describing the behavior of a master node performing Management activities.

The `MANAGEMENT` macro-state is described in Figure 3.6. Initially, the master transits to the `NOT_PRIMARY` state, where two events can be received:

- *jobAssigned*, through which a user submits a job to the master (*line 6*). The message includes the *job* to be managed and the identifier of the submitting user (*userId*). In response to this event, a new *primaryJob* is created, it is added to the *primaryJobList*, and associated with a *JobManager* (*lines 7-9*). Then, the master transits to the `PRIMARY` state (*line 10*).
- *jobReassigned*, an inner event sent by the Backup Job Manager of the same node to notify this master that it has been elected as the new primary for a *backupJob* received as parameter (*line 12*). The following operations are performed: a new *primaryJob*, created from *backupJob*, is added to the *primaryJobList* (*lines 13-14*); all the other backup masters associated with this job are notified about the primary master change, by sending a *primaryUpdated* message to them (*lines 15-16*); a *primaryUpdate* message is also sent to all the slaves that are managing tasks part of this job (*lines 17-18*), as well as to the user that submitted the job (*line 19*); finally, a new Job Manager is started for the *primaryJob* and the state is changed to `PRIMARY`.

When a master is in the PRIMARY state, besides the *jobAssigned* and *jobReassigned* events, which are identical to those present in the NOT_PRIMARY state, it can receive a *jobCompleted* inner event from a Job Manager, on completion of a given job (*line 31*). The event includes the identifier of the completed job (*completedJobId*) and its status (*completedJobStatus*). The master identifies the job in the *primaryJobList*, stops the corresponding Job Manager, and sends a *backupJobCompleted* event to all the backup masters associated with this job (*lines 32-35*). Then, the event is propagated to the user that submitted the job (*line 36*) and, if there are no more jobs being managed, the master transits to the NOT_PRIMARY state.

```
1  Sequential Macro-State RECOVERY {
2   entry() {
3    transition to NOT_BACKUP;
4   }
5   State NOT_BACKUP {
6    on event backupJobAssigned(PrimaryJobType
         primaryJob, NodeIdType primaryId) {
7     BackupJobType t(primaryJob, primaryId);
8     add t to backupJobList;
9     start backupJobManager for t;
10    transition to BACKUP;
11   }
12  }
13  State BACKUP {
14   on event backupJobAssigned(PrimaryJobType
         primaryJob, NodeIdType primaryId) {
15    //same as lines 7 to 9
16   }
17   on event primaryUpdate(JobIdType updatedJobId,
         NodeIdType updatedPrimaryId) {
18    BackupJobType t = tuple in BackupJobList with
          job.jobId=updatedJobId;
19    t.primaryId = updatedPrimaryId;
20    propagate event to backupJobManager for t;
21   }
22   on event jobUpdate(JobIdType updatedJobId,
         List<NodeIdType> updatedBackupMasterList,
         List<JobTaskType> updatedJobTaskList) {
23    BackupJobType bt = tuple in backupJobList
          with job.jobId=updatedJobId;
24    bt.backupMasterList = updatedBackupMasterList;
25    if(updatedJobTaskList is not empty)
26     foreach(JobTaskType t in updatedJobTaskList)
27      update t in bt.jobTaskList;
28   }
29   on [inner] event backupJobCompleted(JobIdType
         completedJobId) {
30    BackupJobType t = tuple in BackupJobList with
          job.jobId=completedJobId;
31    stop backupJobManager for t;
32    if(backupJobList does not contain jobs to
          execute)
33     transition to NOT_BACKUP;
34   }
35  }
36 }
```

Fig. 3.7: Pseudo-code describing the behavior of a master node performing Recovery activities.

Figure 3.7 describes the RECOVERY macro-state. Masters in the NOT_BACKUP state can receive a *backupJobAssigned* event from the Job Manager of another node that is acting as the primary master for a given job (*line 6*). In this case, the node adds a *backupJob* to the *backupJobList*, starts a Backup Job Manager, and transits to the BACKUP state (line 7-10).

In the BACKUP state four events can be received:

- *backupJobAssigned*, which is identical to that received in the NOT_BACKUP state, except for the absence of the state transition (*lines 14-15*).
- *primaryUpdate*, which informs this backup master that the primary master for a given job has changed (*line 17*). The *backupJob* in the *backupJobList* is updated accordingly and the same event is propagated to the corresponding Backup Job Manager (*lines 18-20*).
- *jobUpdate*, which is a message sent by the primary master each time a change happens to the information associated with a given job (*line 22*). The message includes three fields: the identifier of the job whose information has changed (*updatedJobId*); the (possibly updated) list of backup masters associated with *updatedJobId* (*updatedBackupMasterList*);

the tasks associated with *updatedJobId* that have been updated (*updatedJobTaskList*). Note that the *updatedJobTaskList* does not contain whole tasks information, but only the information that has to be updated. In response to this event, the *backupMasterList* and all the relevant tasks in *jobTaskList* are updated (*lines 23-27*).

- *backupJobCompleted*, an event that notifies the completion of a given *backupJob* (*line 29*). In most cases this messages is sent by the primary master on job completion. In the other cases, it is an inner event sent by the Backup Job Manager of the same node, because the node has been elected as the new primary master. In both cases, the Backup Job Manager associated with the job is stopped and, if the *backupJobList* does not contain other jobs to execute, the node transits to the `NOT_BACKUP` state (*lines 30-33*).

```
1 Sequential Macro-State COORDINATION {
2  entry(){
3   transition to NOT_COORDINATOR;
4  }
5  State NOT_COORDINATOR {
6   on inner event coordinatorFailure(NodeId
        coordinatorId) {
7    transition to ELECTING_COORDINATOR;
8   }
9   //specific events based on the election
        algorithm
10  }
11  State ELECTING_COORDINATOR {
12   //specific events based on the election
        algorithm
13  }
14  State WAITING_COORDINATOR {
15   //specific events based on the election
        algorithm
16  }
17  Sequential Macro-State COORDINATOR {
18   entry() {
19    notify other master nodes that this node is
         the coordinator;
20    transition to ACTIVE;
21   }
22   State ACTIVE {
23    entry() {
24     int N = desired #master nodes - current
          #master nodes;
25     if(N>0) {
26      List<NodeIdType> idleSlaveList = search for
           N idle slave nodes;
27      foreach(NodeIdType s in idleSlaveList)
28       send event becomeMaster() to s;
29     }
30     else if(N<0) {
31      List<NodeIdType> idleMasterList = search
           for N idle master nodes;
32      foreach(NodeIdType m in idleMasterList)
33       send event becomeSlave() to m;
34     }
35     transition to IDLE;
36    }
37   }
38   State IDLE {
39    do() {
40     transition to ACTIVE after a timeout;
41    }
42   }
43  }
44 }
```

Fig. 3.8: Pseudo-code describing the behavior of a master node performing Coordination activities.

Finally, the pseudo-code associated with the `COORDINATION` macro-state is described in Figure 3.8. Initially, a master transits to the `NOT_COORDINATOR` state; if a coordinator failure is detected by the network module, the master transits to the `ELECTING_COORDINATOR` state. Here, a distributed election algorithm starts. In particular, assuming to use the Bully algorithm [45], the following procedure takes place:

- A node in the `ELECTING_COORDINATOR` state sends an election message to all masters with higher identifier; if it does not receive a response from a master with a higher identifier within a time limit, it wins the election and passes to the `COORDINATOR` macro-state; otherwise, it transits to the `WAITING_COORDINATOR` state.
- A node in the `WAITING_COORDINATOR` waits until it receives a message from the new coordinator, then it transits to the `NOT_COORDINATOR` state. If it

does not receive a message from the new coordinator before the expiration of a timeout, the node returns to the ELECTING_COORDINATOR state.

When a master enters the COORDINATOR macro-state, notifies all the other masters that it became the new coordinator, and transits to the ACTIVE state (*lines 19-20*). Whenever the coordinator is ACTIVE, it performs its periodical network maintenance operations: if there is a lack of masters (i.e., the number of desired masters is greater than the current number of masters), the coordinator identifies a set of slaves that can be promoted to the master role, and sends a *becomeMaster* message to each of them (*lines 25-29*); if there is an excess of masters, the coordinator transforms a set of idle masters to slaves, by sending a *becomeSlave* message to each of them (*lines 30-34*).

3.3.2 Job and Task Management

As described in Section 3.2.2, MapReduce jobs and tasks are managed by processes called Job Managers and Task Managers, respectively. In particular, Job Managers are primary masters' processes, while Task Managers are slaves' processes. Moreover, masters run Backup Job Managers to manage those jobs they are responsible for as backup masters. In the following we describe the algorithmic behavior of Job Managers, Task Managers and Backup Job Managers.

Figure 3.9 describes the behavior of a Job Manager. The Job Manager includes only two states: ASSIGNMENT and MONITORING. As soon as it is started, the Job Managers transits to the ASSIGNMENT state (*line 4*).

```
1  StateDiagram JOB_MANAGER {
2   PrimaryJobType managedJob;
3   entry() {
4    transition to ASSIGNMENT;
5   }
6   State ASSIGNMENT {
7    entry() {
8     int B = desired #backup nodes - current
          #backup nodes;
9     int S = desired #slave nodes - current #slave
          nodes;
10    if(B>0) {
11     List<NodeIdType> backupList = search for B
           backup nodes;
12     foreach(NodeIdType b in backupList) {
13      send event backupJobAssigned(managedJob,
           nodeId) to b;
14      add b to managedJob.backupMasterList;
15     }
16    }
17    List<JobTaskType> assignedTaskList = empty
          list;
18    if(S>0) {
19     List<NodeIdType> slaveList = search for S
           slave nodes;
20     foreach(NodeIdType s in slaveList) {
21      List<Task> taskList = task assignment for s;
22      foreach (Task t in taskList){
23       JobTaskType jt(t, s, ASSIGNED);
24       update jt in managedJob.jobTaskList;
25       add jt to assignedTaskList;
26      }
27      send event taskAssigned(taskList, nodeId)
           to s;
28     }
29    }
30    if(B>0 or S>0) {
31     foreach(NodeIdType b in
          managedJob.backupMasterList)
32      send event jobUpdate(managedJob.job.jobId,
           managedJob.backupMasterList,
           assignedTaskList) to b;
33    }
34    transition to MONITORING;
35   }
36  }
37  State MONITORING {
38   do() {
39    transition to ASSIGNMENT after a timeout;
40   }
41   on inner event backupMasterFailure(NodeId
        backupId) {
42    remove backupId from
         managedJob.backupMasterList;
43    transition to ASSIGNMENT;
44   }
45   on inner event slaveFailure(NodeId
        failedSlaveId) {
46    foreach(JobTaskType t in
         managedJob.jobTaskList)
47     if(t.slaveId = failedSlaveId){
48      t.slaveId = null id;
49      t.taskStatus = NOT ASSIGNED;
50      update t in managedJob.jobTaskList;
51     }
52    transition to ASSIGNMENT;
53   }
54   on event taskCompleted(TaskIdType
        completedTaskId, TaskStatusType
        completedTaskStatus) {
55    JobTaskType t = tuple in
         managedJob.jobTaskList with
         taskId=completedTaskId;
56    t.taskStatus = completedTaskStatus;
57    List<JobTaskType> updatedTaskList = empty
         list;
58    update t in updatedTaskList;
59    foreach(NodeIdType b in
         managedJob.backupMasterList)
60     send event jobUpdate(managedJob.job.jobId,
          managedJob.backupMasterList,
          updatedTaskList) to b;
61    if(completedTaskStatus != SUCCESS)
62     transition to ASSIGNMENT;
63    else if(all tasks in managedJob.jobTaskList
         have completed with success)
64     send inner event
          jobCompleted(managedJob.job.jobId,
          managedJob.jobStatus) to
          NODE.MASTER.MANAGEMENT;
65   }
66  }
67 }
```

Fig. 3.9: Pseudo-code describing the behavior of a Job Manager.

Briefly, the operations performed by a Job Manager in the `ASSIGNMENT` state are the following: *i*) it calculates the number of backup nodes (B) and slave nodes (S) needed to manage the job (*lines 8-9*); if $B > 0$, it identifies (up to) B masters that can be assigned the backup role, and sends a *backupJobAssigned* message to them (*lines 10-16*); similarly, if $S > 0$, it identifies S slaves, and assigns a subset of the tasks to each of them through a *taskAssigned* message (*lines 18-29*); finally, if $B > 0$ or $S > 0$, a *jobUpdate* event is sent to each backup master associated with the managed job (*lines 30-35*). The search for backup and slave nodes is performed by querying the discovery service.

In the `MONITORING` state four events can happen:

- An internal timeout elapses, causing a transition to the `ASSIGNMENT` state (*line 39*). This enforces the Job Manager to periodically check whether there are backup or slave nodes to be assigned.
- The failure of a backup master is notified by the network module (*backupMasterFailure*) (line 41). The failed backup master is removed from the *backupMasterList* and a transition to the `ASSIGNMENT` state is performed to find a suitable replacement (*lines 42-43*).
- The failure of a slave is detected (*slaveFailure*) (*line 45*), and is managed similarly to the *backupMasterFailure* event.

- A *taskCompleted* event is received from a slave (*line 54*). The event includes the identifier of the completed task (*completedTaskId*) and its status (*completedTaskStatus*). The Job Manager identifies the corresponding task from the *jobTaskList*, changes its status, and notifies all the backup masters about this update through a *jobUpdate* message (*lines 55-60*). If the task has not completed with success, the Job Manager returns to the `ASSIGNMENT` state to reassign it (*lines 61-62*). Finally, if all the tasks have completed with success, it sends a *jobCompleted* message to the `MANAGEMENT` state.

Figure 3.10 describes the behavior of a Task Manager. When started, the Task Manager transits to two concurrent states: `EXECUTION` and `PRIMARY_MONITORING` (*line 4*). The `EXECUTION` state executes the assigned task; on task completion, it sends a *taskCompleted* message to the `SLAVE` state (*lines 8-10*).

```
 1 StateDiagram TASK_MANAGER {
 2  SlaveTaskType managedTask;
 3  entry() {
 4   concurrent transition to EXECUTION,
        PRIMARY_MONITORING;
 5  }
 6  State EXECUTION{
 7   do(){
 8    executes managedTask.task;
 9    managedTask.task.taskStatus = task status;
10    send inner event
         taskCompleted(managedTask.task.taskId,
         managedTask.task.taskStatus) to
         NODE.SLAVE;
11   }
12  }
13  Sequential Macro-State PRIMARY_MONITORING{
14   entry(){
15    transition to CHECK_PRIMARY;
16   }
17   State CHECK_PRIMARY{
18    on inner event primaryMasterFailure(NodeId
         failedPrimaryId) {
19     transition to WAITING_PRIMARY;
20    }
21   }
22   State WAITING_PRIMARY{
23    do() {
24     transition to PRIMARY_UNAVAILABLE after a
          timeout;
25    }
26    on inner event primaryUpdate(JobIdType
         updatedJobId, NodeIdType
         updatedPrimaryId) {
27     transition to CHECK_PRIMARY;
28    }
29   }
30   State PRIMARY_UNAVAILABLE{
31    entry(){
32     send inner event
       taskCompleted(managedTask.task.taskId,
       FAILED) to NODE.SLAVE;
33    }
34   }
35  }
36 }
```

Fig. 3.10: Pseudo-code describing the behavior of a Task Manager.

The `PRIMARY_MONITORING` macro-state allows to monitor the primary master responsible for the job of the managed task. In the case a failure of the primary master is detected (*line 18*), the Task Managers waits for the election of a new primary (*line 26*); if the election is not completed within a time limit, the Task Manager enters a `PRIMARY_UNAVAILABLE` state and sends a *taskCompleted* event to the `SLAVE` state (*line 32*).

Finally, the Backup Job Manager pseudo-code is shown in Figure 3.11. Initially, the Backup Job Manager enters the `CHECK_PRIMARY` state (*line 4*). If a primary master failure is detected, the backup master passes to the `ELECTING_PRIMARY` state and a procedure to elect the new primary master begins. This election algorithm is the same to elect the network coordinator. The new primary transits to the `NEW_PRIMARY` state.

```
1 StateDiagram BACKUP_JOB_MANAGER {
2  BackupJobType managedBackupJob;
3  entry() {
4   transition to CHECK_PRIMARY;
5  }
6  State CHECK_PRIMARY {
7   on inner event primaryMasterFailure(NodeId
        primaryId) {
8    transition to ELECTING_PRIMARY;
9   }
10  //specific events based on the election
        algorithm
11 }
12 State ELECTING_PRIMARY {
13  //specific events based on the election
        algorithm
14 }
15 State WAITING_PRIMARY {
16  on inner event primaryUpdate(JobIdType
        updatedJobId, NodeIdType
        updatedPrimaryId) {
17   transition to CHECK_PRIMARY;
18  }
19  //specific events based on the election
        algorithm
20 }
21 State NEW_PRIMARY {
22  entry() {
23   notify other backup masters that this node is
        the new primary master;
24   remove nodeId from
        managedBackupJob.backupMasterList;
25   send inner event
        backupJobCompleted(managedBackupJob.
        job.jobId) to NODE.MASTER.RECOVERY;
26   send inner event
        jobReassigned(managedBackupJob) to
        NODE.MASTER.MANAGEMENT;
27  }
28 }
29 }
```

Fig. 3.11: Pseudo-code describing the behavior of a Backup Job Manager.

The Backup Job Manager performs the following operations in the `NEW_PRIMARY` state (*lines 23-26*): informs the other backup masters that this node became the new primary master; removes itself from the *backupMasterList* of the managed *backupJob*; sends a *backupJobCompleted* message to the `RECOVERY` state; sends a *jobReassigned* message to the `MANAGEMENT` state.

3.3.3 User Behavior

```
1 StateDiagram USER {
2  UserIdType userId;
3  List<UserJobType> userJobList;
4  on user event submitJob(Job submittedJob) {
5   UserJobType t(submittedJob);
6   add t to userJobList;
7   NodeIdType masterId = search for a master node;
8   send event jobAssigned(submittedJob, userId)
        to masterId;
9  }
10 on event primaryUpdate(JobId updatedJobId,
       NodeIdType updatedPrimaryId) {
11  UserJobType t = tuple in userJobList with
        job.jobId=updatedJobId;
12  t.primaryId = updatedPrimaryId;
13 }
14 on event jobCompleted(JobIdType completedJobId,
       JobStatusType completedJobStatus) {
15  UserJobType t = tuple in userJobList with
        job.jobId=completedJobId;
16  t.jobStatus = completedJobStatus;
17 }
18 }
```

Fig. 3.12: Pseudo-code describing the behavior of a user node.

We conclude this section by describing the behavior of user nodes (see Figure 3.12). Each user includes an identifier (*userId*) and a list of jobs submitted (*userJobList*). Three events are possible:

- *submitJob*, a user-generated event (for this reason not listed in Table 3.1) which requires the submission of a new job (*line 4*). The job is added to the *userJobList*; then, a master is searched and the job is assigned to it using a *jobAssigned* message (*lines 5-8*). The search for a master is performed by querying the discovery service for nodes whose *role* is `MASTER`.
- *primaryUpdate*, which informs the user that the primary master has changed (*line 10*). The message includes the identifier of the job whose primary master has changed, and the identifier of the new primary master (*updatedPrimaryId*). The user identifies the job in the *userJobList*, and changes its *primaryId* to *updatedPrimaryId* (*lines 11-12*).

- *jobCompleted*, which notifies the user that a job has completed its execution (*line 14*). The message includes the identifier of the job and its status (*completedJobStatus*). The job is identified in the *userJobList*, and its status is changed to *completedJobStatus* (*lines 15-16*).

3.4 Implementation

We implemented a prototype of the P2P-MapReduce framework using the JXTA framework [46]. JXTA provides a set of XML-based protocols that allow computers and other devices to communicate and collaborate in a peer-to-peer fashion. Each peer provides a set of services made available to other peers in the network. Services are any type of programs that can be networked by a single or a group of peers.

In JXTA there are two main types of peers: *rendezvous* and *edge*. The rendezvous peers act as routers in a network, forwarding the discovery requests submitted by edge peers to locate the resources of interest. Peers sharing a common set of interests are organized into a *peer group*. To send messages to each other, JXTA peers use asynchronous communication mechanisms called *pipes*. Pipes can be either point-to-point or multicast, so as to support a wide range of communication schemes. All resources (peers, services, etc.) are described by *advertisements* that are published within the peer group for resource discovery purposes.

All master and slave nodes in the P2P-MapReduce system belong to a single JXTA peer group called *MapReduceGroup*. Most of these nodes are edge peers, but some of them also act as rendezvous peers, in a way that is transparent to the users. Each node exposes its features by publishing an advertisement containing basic information that are useful during the discovery process, such as its role and workload. Each advertisement includes an expiration time; a node must renew its advertisement before expiration; nodes associated with expired advertisements are considered as no longer present in the network.

Each node publishes its advertisement in a local cache and sends some keys identifying that advertisement to a rendezvous peer. The rendezvous peer uses those keys to index the advertisement in a distributed hash table called Shared Resource Distributed Index (SRDI), that is managed by all the rendezvous peers of *MapReduceGroup*. Queries for a given type of resource (e.g., master nodes) are submitted to the JXTA Discovery Service that uses SRDI to locate all the resources of that type without flooding the entire network. For example, if a user node wants to search for all the available masters, it submits a query to the JXTA Discovery Service asking for all the advertisements whose field *role* is equal to `MASTER`. Note that *M-net* and *S-net*, introduced in Section 3.2, are "logical" networks in the sense that queries to *M-net* (or *S-net*) are actually submitted to the whole *MapReduceGroup* but restricted to nodes having their field *role* equal to `MASTER` (or `SLAVE`) using the SRDI mechanisms.

Pipes are the fundamental communication mechanisms of the P2P-MapReduce system, since they allow the asynchronous delivery of event messages among nodes. Different types of pipes are employed within the system: bidirectional pipes are used between users and primary masters to submit jobs and return results, as well as between primary masters and their slaves to submit tasks and receive results notifications, while multicast pipes are used by primary masters to send job updates to their backups.

In JXTA pipes it is possible to rebind one endpoint without affecting the other endpoint. We use this feature when a failure occurs: in fact, the new primary master can bind the pipes that were previously used by the old primary master, without affecting the entities connected at the other endpoint (i.e., the user node and the slave nodes).

We conclude this section briefly describing the software modules inside each node and how those modules interact each other in a P2P-MapReduce network. Figure 3.13 shows such modules and interactions using the UML Deployment/Component Diagram formalism.

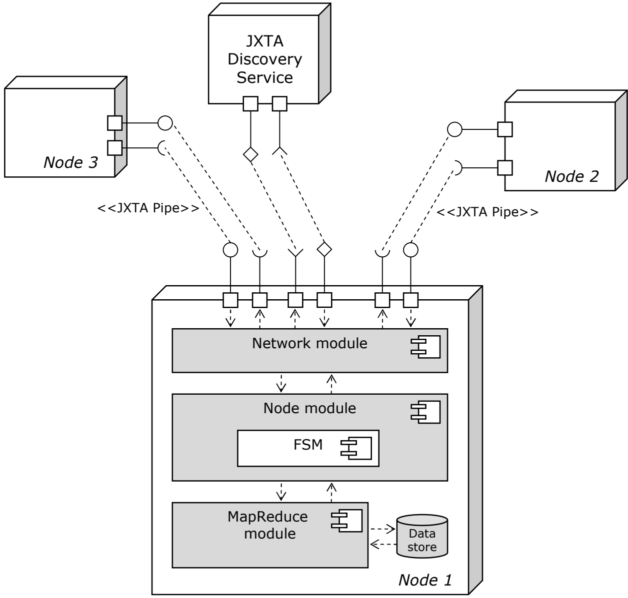


Fig. 3.13: UML Deployment/Component Diagram describing the software modules inside each node and the interactions among nodes.

Each node includes three software modules/layers: *Network*, *Node* and *MapReduce*:

- The Network module is in charge of the interactions with the other nodes by using the pipe communication mechanisms provided by the JXTA framework. When a connection timeout is detected on a pipe associated with a remote node, this module propagates the appropriate failure event to the Node module. Additionally, this module allows the node to interact with the JXTA Discovery Service for publishing its features and for querying the system (e.g., when looking for idle slave nodes).
- The Node module controls the lifecycle of the node in its various aspects, including network maintenance, job management, and so on. Its core is represented by the FSM component which implements the logic of the finite state machine described in Figure 3.4, steering the behavior of the node in response to inner events or messages coming from other nodes (i.e., job assignments, job updates, and so on).
- The MapReduce module manages the local execution of jobs (when the node is acting as a master) or tasks (when the node is acting as a slave). Currently this module is built around the local execution engine of the Hadoop system [16].

While the current implementation is based on JXTA for the Network layer and on Hadoop for the MapReduce layer, the layered approach described in Figure 3.13 is thought to be independent from a specific implementation of the Network and MapReduce modules. In other terms, it may be possible to adopt alternative technologies for the Network and MapReduce layers without affecting the core implementation of the Node module.

3.5 System Evaluation

A set of experiments has been carried out to evaluate the behavior of the P2P-MapReduce framework compared to a centralized implementation of MapReduce in the presence of different levels of churn. In particular, we focused on comparing P2P and centralized implementations in terms of fault tolerance, network traffic, and scalability.

The remainder of this section is organized as follows. Section 3.5.1 describes the experimental setup and methodology. Section 3.5.2 compares fault tolerance capability of P2P and centralized MapReduce implementations. Section 3.5.3 analyzes the systems in terms of network traffic. Section 3.5.4 concentrates on the scalability of the systems. Finally, 3.5.5 summarizes the main results of the present evaluation.

3.5.1 Experimental Setup and Methodology

The evaluation has been carried out by using a custom-made discrete-event simulator that reproduces the behavior of the P2P-MapReduce prototype described in the previous section, as well as the behavior of a centralized MapReduce system like that introduced in Section 2.4.1. While the P2P-MapReduce

prototype allowed us to perform functional testing of the system mechanisms on a small scale, the simulator allowed us to perform non-functional testing (i.e., performance evaluation) on large networks (thousands of nodes), which represent our reference scenario.

The simulator models joins and leaves of nodes and job submissions as Poisson processes; therefore, the interarrival times of all the join, leave and submission events are independent and obey an exponential distribution with a given rate. This model has been adopted in literature to evaluate several P2P systems (see, for example, [47] and [48]), for its ability to approximate real network dynamics reasonably well.

Table 3.2 shows the input parameters used during the simulation.

Symbol	Description	Values
N	Initial number of nodes in the network	5000, 7500, 10000, 15000, 20000, 30000, 40000
NM	Number of masters (% on N)	1 (P2P only)
NB	Number of backup masters per job	1 (P2P only)
LR	Leaving rate: avg. number of nodes that leave the network every minute (% on N)	0.025, 0.05, 0.1, 0.2, 0.4
JR	Joining rate: avg. number of nodes that join the network every minute (% on N)	equal to LR
SR	Submission rate: avg. number of jobs submitted every minute (% on N)	0.01
JT	Job type	A, B, C (see Table 3.3)

Table 3.2: Simulation parameters

As shown in the table, we simulated MapReduce systems having an initial size ranging from 5000 to 40000 nodes, including both slaves and masters. In the centralized implementation, there is one master only and there are not backup nodes. In the P2P implementation, there are 1% masters (out of N) and each job is managed by one master which dynamically replicates the job state on one backup master.

To simulate node churn, a joining rate JR and a leaving rate LR have been defined. On average, every minute JR nodes join the network, while LR nodes abruptly leave the network so as to simulate an event of failure (or a graceless disconnection). In our simulation $JR = LR$ to keep the total number of nodes approximatively constant during the whole simulation. In particular, we used five values for JR and LR: 0.025, 0.05, 0.1, 0.2 and 0.4, so as to evaluate the system under different churn rates. Note that such values are expressed as a percentage of N. For example, if $N = 10000$ and $LR = 0.05$, there are on average 5 nodes leaving the network every minute.

Every minute, SR jobs are submitted on average to the system by user entities. The value of such submission rate is 0.01, expressed, as for JR and

LR, as a percentage of N. Each job submitted to the system is characterized by two parameters: total computing time and number of tasks. To evaluate the behavior of the system under different loads, we defined three job types (JT), A, B and C, as detailed in Table 3.3.

Type	Total computing time (hours)	Number of tasks
A	100	200
B	150	300
C	200	400

Table 3.3: Job types and associated parameters (average values)

Hence, the jobs submitted to the system simulator belong either to type A, B or C. For a given submitted job, the system calculates the amount of time that each slave needs to complete the task assigned to it as the ratio between the total computing time and the number of tasks required by that job. Tasks are assigned to the slaves with the lowest workload, i.e., with the lowest number of assigned tasks. Each slave keeps the assigned tasks in a priority queue. After the completion of the current task, the slave selects for execution the task that has failed the highest number of times among those present in the queue.

In order to compare the P2P-MapReduce system with a centralized MapReduce implementation, we analyzed several scenarios characterized by different combinations of simulation parameters (i.e., network size, leaving rate, job type). For each scenario under investigation, the simulation ends after the completion of 500 jobs. At the end of the simulation, we collect four performance indicators:

- The *percentage of failed jobs*, which is the number of jobs failed expressed as a percentage of the total number of jobs submitted.
- The *percentage of lost computing time*, which is the amount of time spent executing tasks that were part of failed jobs, expressed as a percentage of the total computing time.
- The *number of messages* exchanged through the network during the whole simulation process.
- The *amount of data* associated with all the messages exchanged through the network.

For the purpose of our simulations, a "failed" job is a job that does not complete its execution, i.e., does not return a result to the submitting user entity. The failure of a job is always caused by a not-managed failure of the master responsible for that job. The failure of a slave, on the contrary, never causes a failure of the whole job because its task is reassigned to another slave.

3.5.2 Fault Tolerance

As mentioned earlier, one of the goals of our simulations is to compare the P2P and centralized implementations in terms of fault tolerance, i.e., the percentage of failed jobs and the corresponding percentage of lost computing time. The results discussed in this section have been obtained considering the following scenario: $N = 10000$ and LR ranging from 0.025 to 0.4. Figure 3.14 compares the percentage of failed jobs in such scenario, for each of the job types defined above: (a) JT=A; (b) JT=B; (c) JT=C.

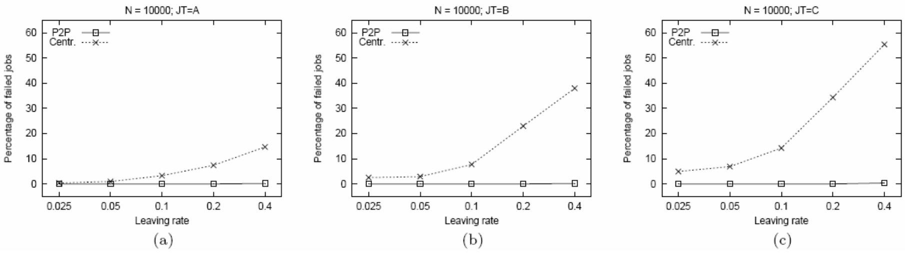


Fig. 3.14: Percentage of failed jobs in a network with 10000 nodes: (a) JT=A; (b) JT=B; (c) JT=C.

As expected, with the centralized MapReduce implementation the percentage of failed jobs significantly increases with the leaving rate, for each job type. For example, when $JT = B$, the percentage of failed jobs passes from 2.5 when $LR = 0.025$, to 38.0 when $LR = 0.4$. Moreover, we can observe that, fixed the value of LR, the percentage of failed jobs increases from JT=A to JT=B, and from JT=B to JT=C. For example, with LR=0.1, the percentage of failed jobs is 3.3 for JT=A, 7.8 for JT=B, and 14.2 for JT=C. This is motivated by the fact that longer jobs (as jobs of type C are compared to jobs of type B and A) are statistically more subject to be affected by a failure of the associated master.

In contrast to the centralized implementation, the P2P-MapReduce framework is limitedly affected by job failures. In particular, for any job type, the percentage of failed jobs is 0% for $LR \leq 0.2$, while it is ranges from 0.2% to 0.4% for $LR = 0.4$, even if only one backup master per job is used. It is worth recalling here that when a backup master becomes primary master as a consequence of a failure, it chooses another backup in its place to maintain the desired level of reliability, as discussed in Section 3.3.

Figure 3.15 reports the percentage of lost computing time in centralized and P2P implementations related to the same experiments of Figure 3.14, for different combinations of network sizes, leaving rates and job types. The figure also shows the amount of lost computing time, expressed in hours, in correspondence of each graph point for the centralized and P2P cases.

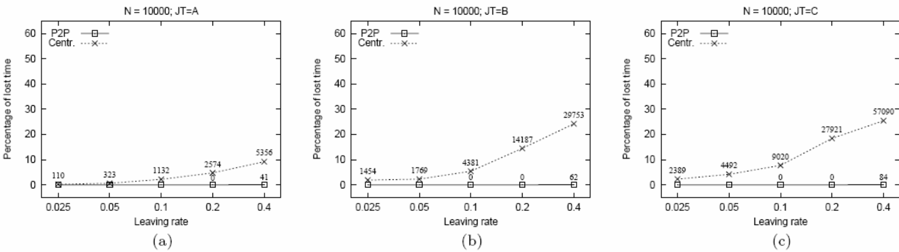


Fig. 3.15: Percentage of lost time in a network with 10000 nodes: (a) JT=A; (b) JT=B; (c) JT=C. The numbers in correspondence of each graph point represent the amount of lost computing time expressed in hours (some zero values are omitted for readability).

The lost computing time follows a similar trend as the percentage of failed jobs, and it results affected by the same dependence from the job type. For example, when LR=0.4, the percentage of lost computing time for the centralized system passes from 9.2 for JT=A to 25.5 for JT=C, while the percentage of time lost by the P2P system is under 0.1% in the same configurations. The difference between centralized and P2P is even clearer if we look at the absolute amount of computing time lost in the various scenarios. In the worst case (LR=0.4 and JT=C), the centralized system loses 57090 hours of computation, while the amount of lost computing time with the P2P-MapReduce system is only 84 hours.

3.5.3 Network Traffic

This section compares the P2P and centralized implementations of MapReduce in terms of network traffic, i.e., the number of messages exchanged through the network during the whole simulation, and the corresponding amount of data expressed in MBytes. The amount of data is obtained by summing the sizes of all the messages that are exchanged through the network.

In order to calculate the size of each messages, Table 3.4 lists the sizes of all the basic components that may be found in a message.

Message components	Size (Bytes)
Header	260
Identifier (e.g., jobId, taskId)	4
Code (job.code, task.code)	4000
URL (e.g., job.input, job.output)	150
Integer (e.g., job.M, job.R)	4
Status (e.g., task.type, jobStatus)	1

Table 3.4: Sizes of message components

Each message includes a *header* that represents the fixed amount of traffic each message generates independently from the specific payload. Its size has been determined experimentally by measuring the average amount of traffic generated to transfer an empty message from a host to another host using a TCP socket. The sizes for *identifier*, *integer* and *status* variables are those used in common system implementations. The size of the *code* component is the average code size observed on a set of MapReduce applications; the size of the *URL* component has been calculated similarly.

For example, let's calculate the size of a *jobAssigned* message. From Table 3.1, we know that a *jobAssigned* message includes three parts: 1) one *Job* tuple; 2) one *UserIdType* variable; 3) one *header* (implicitly present in each message). While the size of the second and third parts are known (respectively 4 and 260 Bytes), the size of the first part must be calculated as the sum of each of its fields. From Section 3.2.1, a *Job* tuple includes the following fields: *jobId* (4 Bytes), *code* (4000 Bytes), *input* (150 Bytes), *output* (150 Bytes), *M* (4 Bytes) and *R* (4 Bytes), for a total of 4312 Bytes. Therefore, the size of a *jobAssigned* message is equal to 4576 Bytes.

The size of messages that include lists, like *taskAssigned*, is calculated taking into account the actual number of elements in the list, and the size of each such elements. For the messages generated by the discovery service and by the election algorithms, we proceeded in the same way. We just mention that most of such messages are very small since they include only a few fields.

For the purpose of the evaluation presented below, we distinguish four categories of messages:

- *Management*: messages exchanged among nodes to manage jobs and tasks execution. Referring to Table 3.1, the management messages are those associated with the following (not inner) events: *jobAssigned*, *jobCompleted*, *taskAssigned* and *taskCompleted*.
- *Recovery*: messages exchanged among primary masters and their backups to dynamically replicate job information (*backupJobAssigned*, *backupJob-Completed*, *jobUpdate* and *primaryUpdate*), as well as to elect a new primary in the case of a master failure (messages specific to the election algorithm used).

- *Coordination*: messages generated by the coordinator to perform network maintenance operations (*becomeMaster* and *becomeSlave*), as well as to elect the new coordinator (specific to the election algorithm).
- *Discovery*: messages generated to publish and search information about nodes using the JXTA Discovery Service.

Management messages are present both in the P2P and in the centralized case, since they are generated by the standard execution mechanisms of MapReduce. In contrast, recovery, coordination and discovery operations are performed only by the P2P-MapReduce system, therefore the corresponding messages are not present in the centralized case.

We start focusing on the total traffic generated, without distinguishing the contribution of the different categories of messages, in order to obtain an aggregate indicator of the overhead generated by the two systems. As for the previous section, the results presented here are obtained considering a network with $N = 10000$ and LR ranging from 0.025 to 0.4. In particular, Figure 3.16 compares the total number of messages exchanged for three job types: (a) JT=A; (b) JT=B; (c) JT=C.

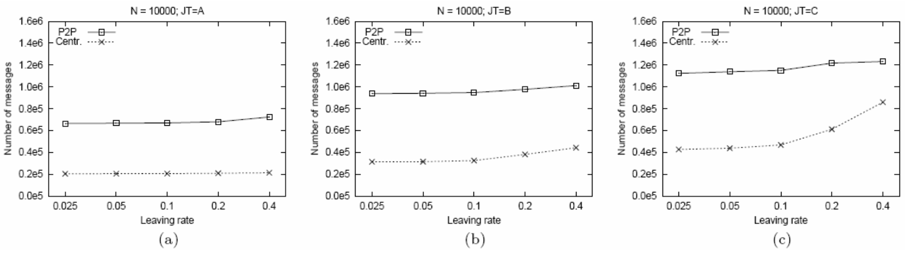


Fig. 3.16: Number of messages exchanged in a network with 10000 nodes: (a) JT=A; (b) JT=B; (c) JT=C.

As shown by the graphs, the total number of messages generated by the P2P system is higher than that generated by the centralized system in all the considered scenarios. This is mainly due to the presence in the P2P system of discovery messages, which are not present in the centralized system. We will discuss later in this section the impact of the different types of messages also in terms of amounts of data exchanged.

We observe that in both cases - P2P and centralized - the number of messages increases with the leaving rate. This is due to the fact that by increasing the leaving rate also the number of failed jobs increases; since failed jobs are resubmitted, a subsequent increase in the number of management messages is produced. As shown in the figure, such increase is higher with the centralized MapReduce implementation, being higher the number of failed jobs compared to P2P-MapReduce.

The increase in the number messages is higher for heavy jobs (i.e., JT=B or JT=C), since their failure requires the reassignment of a greater number of tasks, thus producing a higher number of management messages. For example, Figure 3.16c shows that with JT=C, the number of messages for the P2P case passes from 1.12 millions when $LR = 0.025$, to 1.23 millions when $LR = 0.04$, which corresponds to an increase of about 10%. In contrast, the number of messages for the centralized case passes from 0.43 to 0.86 millions, which corresponds to an increase of 100%.

Figure 3.17 shows the amount of data associated with all the messages exchanged through the network.

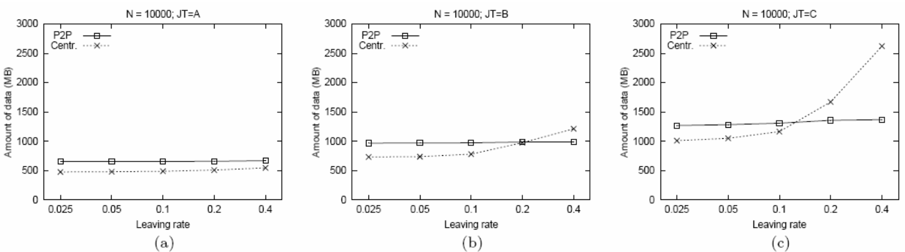


Fig. 3.17: Data exchanged (MBytes) in a network with 10000 nodes: (a) JT=A; (b) JT=B; (c) JT=C.

For the P2P case, the amount of data for a given job type increases very little with the leaving rate. In fact, the few jobs that fail even with the higher leaving rate, produce a relatively little number of additional management messages and so they have a limited impact in terms of amount of data exchanged. For the centralized case, the amount of data for a given job type increases significantly with the leaving rate, since the percentage of failed jobs grows faster than the P2P case.

It is interesting to observe that, in some scenarios, the amount of data exchanged in the centralized implementation is greater than the amount of data exchanged in P2P-MapReduce. In our simulations this happens when $LR > 0.2$ for JT=B (see Figure 3.17b), and when $LR > 0.1$ for JT=C (see Figure 3.17c). In particular, with $LR = 0.4$ and JT=C, the amount of data exchanged is equal to 1369 MB for the P2P system and 2623 MB for the centralized implementation.

We conclude the traffic analysis by showing what is the contribution of the different types of messages (management, recovery, coordination and discovery) in terms of number of messages and corresponding amount of data exchanged through the network. Figure 3.18 presents the results of such analysis for a network with $N = 10000$ and JT=C.

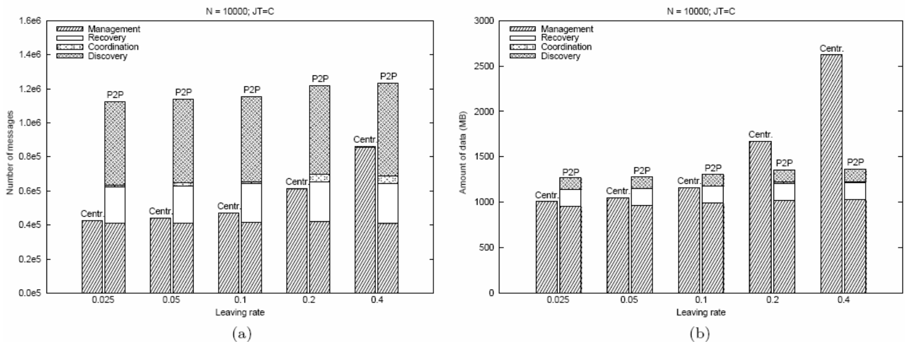


Fig. 3.18: Detailed traffic in a network with $N = 10000$ and JT=C: (a) number of messages; (b) amount of data.

As stated earlier, in the centralized case only management messages are generated. Therefore, their number and corresponding amount of data are the same already shown in Figures 3.16c and 3.17c. We just highlight that, for high values of leaving rate, the number of messages and the amount of data grows significantly.

For the P2P case, we observe that the management messages represent only one third of the total number of messages. Discovery messages represent 40% in terms of number of messages, but only 10% in terms of amount of data. This is due to the fact that the size of discovery messages is very small, as mentioned earlier, and so they do not produce a significant network overhead. Also recovery and coordination messages have limited impact on the total network traffic, both in terms of number of messages and amount of data.

3.5.4 Scalability

We finally conducted a set of simulations to evaluate the behaviors of the P2P and centralized MapReduce implementations by varying the network size. In particular, Figure 3.19 compares P2P and centralized systems with $LR = 0.1$, JT=C, and N ranging from 5000 to 40000, in terms of: (a) percentage of failed jobs; (b) percentage (and absolute amount) of lost computing time; (c) number of messages; (d) amount of data exchanged.

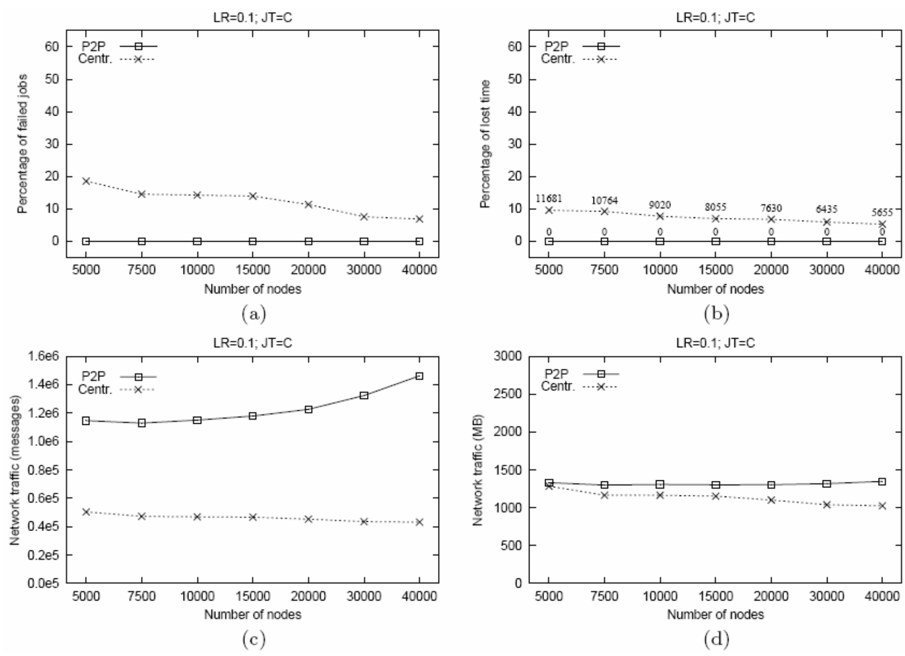


Fig. 3.19: Comparison of P2P and centralized systems with $LR = 0.1$, JT=C, and N ranging from 5000 to 40000: (a) percentage of failed jobs; (b) percentage (and absolute amount) of lost computing time; (c) number of messages; (d) amount of data exchanged.

As shown in Figure 3.19a, the percentage of failed jobs for the centralized case slightly decreases when the network size increases. This is due to the fact that jobs complete faster in larger networks, since the number of slaves increases and the job type is fixed (JT=C in our case), and so they are less affected by failure events. On the other hand, in the P2P case the percentage is always zero, independently from the network size. For the percentage of lost computing time (see Figure 3.19b) a similar trend can been noted.

Regarding network traffic (see Figures 3.19c and 3.19d), we observe that, in the P2P case, the number of messages slightly increases with the number of nodes. This is due to the higher number of discovery and coordination messages that are generated in larger networks. However, in terms of amount of data this increment is negligible. Also for the centralized system the variation is not significant passing from 5000 to 40000 nodes.

3.5.5 Remarks

The results discussed above confirm the fault tolerance level provided by the P2P-MapReduce framework compared to a centralized implementation

of MapReduce, since in all the scenarios analyzed the amount of failed jobs and the corresponding lost computing time was negligible. The centralized system, on the contrary, was significantly affected by high churn rates, producing critical levels of failed jobs and lost computing time.

The experiments have also shown that the P2P-MapReduce system generates more messages than a centralize MapReduce implementation. However, the difference between the two implementations reduces as the leaving rate increases, particularly in the presence of heavy jobs. Moreover, if we compare the two systems in terms of amount of data exchanged, we see that in many cases the P2P-MapReduce system is more efficient than the centralized implementation.

We have finally assessed the behavior of P2P-MapReduce with different network sizes. The experimental results showed that the overhead generated by the system is not significantly affected by an increase of the network size, thus confirming the good scalability of our system.

In summary, the experimental results show that even if the P2P-MapReduce system consumes in most cases more network resources than a centralized implementation of MapReduce, it is far more efficient in job management since it minimizes the lost computing time due to jobs failures.

3.6 Conclusion

The P2P-MapReduce framework exploits a peer-to-peer model to manage node churn, master failures, and job recovery in a decentralized but effective way, so as to provide a more reliable MapReduce middleware that can be effectively exploited in dynamic Cloud infrastructures.

This chapter provided a detailed description of the basic mechanisms that are at the base of the P2P-MapReduce system, presented a prototype implementation based on the JXTA peer-to-peer framework, and an extensive performance evaluation of the system in different network scenarios.

The experimental results showed that, differently from centralized master-server implementations, the P2P-MapReduce framework does not suffer from job failures even in presence of very high churn rates, thus enabling the execution of reliable MapReduce applications in very dynamic Cloud infrastructures.

4

COMPSs applications on the Cloud

The growth of Cloud services and technologies has brought many advantages and opportunities to scientific communities offering users efficient and cost-effective solutions to their problems of lack of computational resources. Even though the Cloud paradigm does not address all the issues related to the porting and execution of scientific applications on distributed infrastructures, it is widely recognized that, through Clouds, researchers can provision compute resources on a pay-per-use basis, thus avoiding to enter in a procurement process that implies investment costs for buying hardware or access procedures to supercomputers.

Recently, several Grid initiatives and distributed computing infrastructures [49] [50] [51] have started to develop Cloud services in order to provide existing services through virtualized technologies for the dispatch of scientific applications. These technologies allow the deployment of hybrid computing environments where the provision of private Clouds is backed up by public offerings such as Azure[9] or Amazon[10]. The VENUS-C[52] project in particular aims to support research and industry user communities to leverage Cloud computing for their applications through the provision of a hybrid platform that provides commercial (Azure) and open source Cloud services.

In such a hybrid landscape, there are technical challenges such as interoperability that need to be addressed from different points of view. The interoperability concept can refer to different things at many levels. It could mean the ability to keep the behavior of an application when it runs on different environments such as a cluster, a Grid or an IaaS provided infrastructure like Amazon instances. At lower level, it might refer to a single application running in many Clouds being able to share information, which might require having a common set of interfaces and the ability of users to use the same management tools, server images and other software with a variety of Cloud computing providers and platforms.

From a programming framework perspective these issues have to be solved also at different levels, developing the appropriate interfaces to interact with several Cloud providers, ensuring that the applications are executed on dif-

ferent infrastructure without having to adapt them and handling data movements seamlessly amongst different Cloud storages.

The COMP Superscalar[20] programming framework allows the programming of scientific applications and their execution on a wide number of distributed infrastructures. In Cloud environments, COMPSs provides scaling and elasticity features allowing to adapt the number of available resources to the actual need of the execution. The availability of connectors for several providers makes possible the execution of scientific applications on hybrid Clouds taking into account the above mentioned issues related to the porting of applications to a target Cloud and their transparent execution with regards to the underlying infrastructure. This chapter describes the developments for making COMPSs interoperable with Windows Azure Platform through the design of a specific adaptor.

The rest of the chapter is organized as follows. Section 4.1 describes the COMPSs framework. Section 4.2 details the developed Azure GAT Adaptor. Section 4.3 illustrates the porting of a data mining application to COMPSs. Section 4.4 evaluates the performance of the ported application. Section 4.5 discusses the related work. Section 4.6 presents the conclusions and the future work.

4.1 The COMPSs framework

COMPSs is a programming framework, composed of a programming model and an execution runtime, whose main objective is to ease the development of applications for distributed environments.

The programming model aims to keep the programmers unaware of the execution environment and parallelization details. They are only required to create a sequential application and specify which application methods will be executed as remote tasks. This specification is done with an “annotated interface”, i.e. a description about methods to execute on remote nodes and their parameters. Specifically, the description includes the name of the class that implements the method(s) and, for each parameter, its type (e.g., primitive, file) and direction (in, out or in/out). The user can also express capabilities that a resource must fulfill to run a certain method (e.g., CPU number and type, memory, disk size). Figure 4.1 shows an example of BLAST [14] application porting in COMPSs, as outlined in [53]: the user specifies the main of the application (a), method that will be executed on remote nodes, and the annotated interface (c).

```
public static void main(String args[]) throws Exception {
   sequences[] = split(inputFile, nFrags);
   for (fragment: sequences)
   {
     output = "resFile" + index + ".txt";
     BlastImpl.alignment(db, fragment, output, ..., cmdArgs);
     seqOutputs.add(output);
     index++;
   }
   assemblyPartitions(seqOutputs, resultFile, tempDir, nFrags);
}
```

(a)

```
public void alignment(String db, String fragment,
                      String resFile, ..., String cmdArgs){

   String cmd = blastBinary+ " " +"-p blastx -d " + db +
                " -i " +fragment+ " -o "+resFile+" "+cmdArgs;

   Process simProc = Runtime.getRuntime().exec(cmd);
   ...
}
```

(b)

```
public interface BlastItf {
  @Method(declaringClass = "blast.worker.BlastImpl")
  @Constraints(storageElemSize = 0.3f, processorCPUCount = 4)
  void alignment(
     @Parameter(type = Type.STRING, direction = Direction.IN)
     String db,
     @Parameter(type = Type.FILE, direction = Direction.IN)
     String fragment,
     @Parameter(type = Type.FILE, direction = Direction.OUT)
     String resFile,
     @Parameter(type = Type.STRING, direction = Direction.IN)
     String blastBinary,
     @Parameter(type = Type.STRING, direction = Direction.IN)
     String cmdArgs);
}
```

(c)

Fig. 4.1: COMPSs application definition: (a) Application main; (b) Method executed on a remote node; (c) Annotated interface.

The runtime is in charge of optimizing the performance of the application by exploiting its inherent concurrency. Figure 4.2 shows the operations performed by runtime in order to execute a COMPSs application. The runtime intercepts any call to a selected method creating a representative task and finding the data dependencies with all the previous ones that must be considered along the application run. The task is added to a task dependency graph as a new node and such dependencies are represented by edges of the graph. Tasks with no dependencies enter the scheduling step and are assigned to available resources. This decision is made according to a scheduling algorithm that takes into account data locality, task constraints and the workload of each node. According to this decision the input data for the scheduled task are transferred to the selected host and the task is remotely submitted. Once

a task finishes, the task dependency graph is updated, possibly resulting in new dependency-free tasks that can be scheduled.

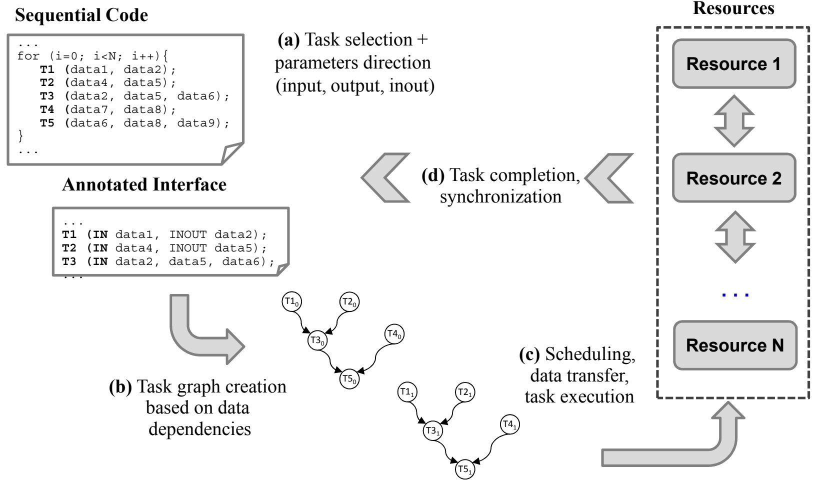


Fig. 4.2: Operations performed by COMPSs runtime during the execution of an application.

Paper[53] describes how COMPSs could also benefit from Infrastructure-as-a-Service (IaaS) offerings. Through the monitoring of the workload of the application, the runtime determines the excess/lack of resources and turns to Cloud providers enforcing a dynamic management of the resource pool. In order to make COMPSs interoperable with different providers, a common interface is used, which implements the specific Cloud provider API. Currently, there exist connectors for Amazon EC2 and for providers that implement the Open Cloud Computing Interface (OCCI)[7] and the Open Virtualization Format (OVF)[8] specifications for resource management.

4.2 The Azure JavaGAT Adaptor

In order to solve the interoperability issues related to the execution of tasks using an heterogeneous pool of resources in distributed environments, COMPSs adopts JavaGAT[54] as the uniform interface to underlying Grid and Cloud middlewares implemented in several adaptors. Whenever a task has to be executed on a specific resource, COMPSs manages all the data transfers and submits the task using the proper adaptor. The Azure JavaGAT Adaptor here described enriches COMPSs with data management and execution capabilities that make it interoperable with Azure and implemented using two subcomponents.

Data management is supported by a subcomponent called Azure File Adaptor. It allows to read and write data on the Azure Blob Storage (*Blobs*), to deploy the *libraries* needed to execute on Azure and to store the input and output data (*taskdata*) for the tasks. The Azure Resource Broker Adaptor, on the other side, is responsible for the task submission. Following the Azure Work Queue pattern, this subcomponent adds into a *Task Queue* the tasks that must be executed on an Azure resource by a *Worker*. The implementation of these COMPSs workers as Worker Role instances is based on a previous work on a Data Mining Cloud App framework[55]. In order to keep the runtime informed about each task execution, the status of the tasks is updated in a *Task Status Table*. The whole architecture of the Azure JavaGAT Adaptor is depicted in Figure 4.3.

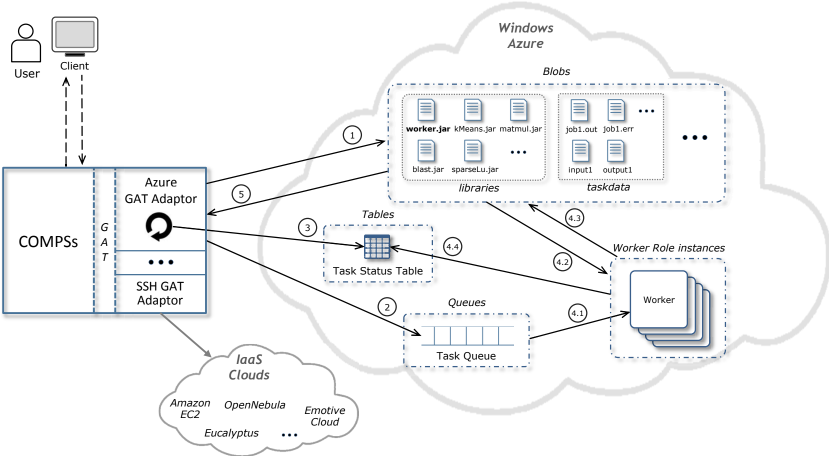


Fig. 4.3: The Azure GAT adaptor architecture.

The numbered components in Figure 4.3 correspond to each item in the list below, which describes the different stages of a remote task execution on Azure. The whole process starts when the COMPSs runtime decides to execute a dependency-free task t in the platform following the next steps:

1. The Azure GAT adaptor, through the Azure File adaptor, prepares the execution environment uploading the input application files and libraries into the Blob containers, *taskdata* and *libraries*.
2. The adaptor, via the Azure Resource Broker, inserts a task t description into the *Task Queue*.
3. The adaptor sets the status of the task t to *Submitted* in the *Task Status Table* and polls periodically in order to monitor its status until it becomes *Done* or *Failed*.

4. An idle worker W takes the task t description from the queue and, after parsing all the parameters, it runs the task. This step can be divided in the following sub-steps:
   4.1. The worker W takes the task t from the *Task Queue* starting its execution on a virtual resource. The worker sets the status of t to *Running*.
   4.2. The worker gets the needed input data and the needed libraries according to the description of t. To this end, a file transfer is performed from the Blob, where the input data is located, to the local storage of the resource, and the task is executed.
   4.3. After a task completion, the worker W moves the resulting files in the *taskdata* Blob container.
   4.4. The worker updates the status of the task in the *Task Status Table* setting it to a final status that could be *Done* or *Failed*.
5. When the adaptor detects that the task t execution has finalized, it notifies the execution end to the runtime which looks for new dependency-freed tasks to be executed. If the output files are not going to be used by any other task, the runtime downloads them from the Azure Blob.

4.3 Data mining on COMPSs: a classifier-based workflow

In order to validate the described work, a data mining application has been adapted to run in a Cloud environment through COMPSs. Such application runs multiple instances of the same classification algorithm on a given dataset, obtaining multiple classification models, then chooses the one that classify in a more accurate way. Thus, the aim is twofold: first, validate the implementation checking that the system is able to manage the execution on different Cloud deployments; second, compare the performance of the proposed solution on an hybrid Cloud scenario. The rest of the section describes the data mining application as a workflow (Section 4.3.1), its Java implementation (Section 4.3.2) and the porting to COMPSs (Section 4.3.3).

4.3.1 The application workflow

Figure 4.4 depicts the four general steps of the classifier-based workflow:

1. **Dataset partition:** the initial dataset is split into two parts: a training set, which trains the classifiers, and a test set to check the effectiveness of the achieved models.
2. **Classification:** during this step, the training dataset is analyzed in parallel using multiple instances of the same classifier algorithm with different parameters.
3. **Evaluation:** the quality of each classification model is measured using different performance metrics (e.g., number of misclassified items, precision and recall measure, F-measure).

4. **Model selection:** finally, the best model is selected optimizing the chosen performance metrics.

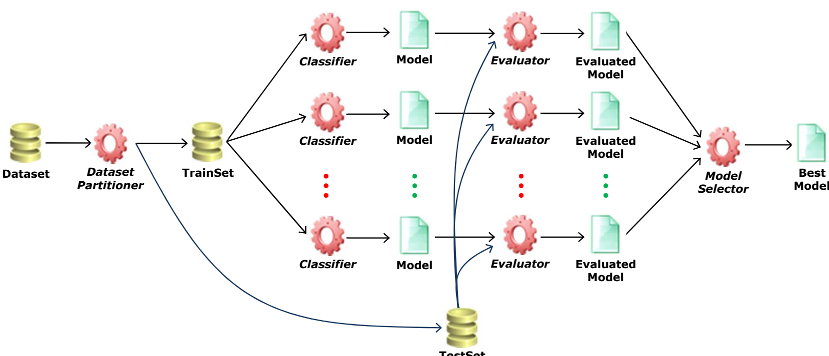


Fig. 4.4: The data mining application workflow

4.3.2 The application implementation

Following the described workflow, the initial dataset is divided in two parts: $2/3^{rd}$ are left as a training set and the remaining $1/3^{rd}$ is used as test set. The classification algorithm is the *J48*, provided in Weka[56] data mining toolkit, based on C4.5[57] algorithm. This algorithm builds a decision tree using the concept of information entropy to classify the different items in the training set. The different models are obtained varying the *confidence value* parameter of J48 in a range of values (i.e., from 0.05 to 0.50). Such range is divided in a certain number of intervals specified by the user as an application parameter. Each model is evaluated using, as a performance metric, the number of misclassified items. Figure 4.5 presents the main code of the application:

```
1  public static void main(String args[]) throws Exception {
2    ...
3    //Run remote method
4    for (int i = 0; i < n_itvls; i++){
5       c_val = c_min_val+i*(c_max_val-c_min_val)/(num_itvls-1);
6       //***************** Remote methods ******************//
7       models[i]=WorkflowImpl.learning(trainSet, c_val);
8       reports[i]=WorkflowImpl.evaluate(models[i], testSet);
9    }
10   //Selection of the best model in binary tree way
11    int n = 1;
12    while (n < n_itvls){
13      for (int i = 0; i < n_itvls; i+= 2 * n){
14        if (i + n < n_itvls) {
15          //***************** Remote method ******************//
16          WorkflowImpl.getBestIndex(reports[i], reports[i+n]);
17        }
18      }
19     n *= 2;
20    }
21    //Read best model
22    J48 bestModel = models[reports[0].getIndex()];
23 }
```

Fig. 4.5: Main application code.

As described in the application workflow section, the methods in lines *7* and *8* correspond to the classification and evaluation steps of the workflow. The *c_min_val* and *c_max_val* are the limits of the confidence value range, and *num_itvls* is the number of intervals specified by user. The model selection step (lines $11 - 22$) is performed in binary tree way in order to exploit the possibility to be parallelized by COMPSs as detailed in along the next section.

4.3.3 Parallelization with COMPSs: the interface

The main step of the porting of an application to COMPSs includes the preparation of a Java annotated interface provided by the programmer in order to select which methods will be executed remotely. For each annotated method, the interface specifies some information like the name of the class that implements it, and the type (e.g., primitive, object, file) and direction (e.g., in, out or in/out) of its parameters. The user can add some additional metadata to define the resource features required to execute each method. The Figure 4.6 shows the annotated interface for the presented application.

```
public interface WorkflowItf {
  @Constraints(processorCPUCount = 1, storageElemSize= 1.0)
  @Method(declaringClass = "workflow.WorkflowImpl")
  J48 learning(
     @Parameter(type = Type.OBJECT, direction = Direction.IN)
     Instances trainSet,
     @Parameter(type = Type.FLOAT, direction = Direction.IN)
     float confFactor);

  @Method(declaringClass = "workflow.WorkflowImpl")
  Report evaluate(
     @Parameter(type = Type.INT, direction = Direction.IN)
     int i,
     @Parameter(type = Type.OBJECT, direction = Direction.IN)
     J48 model,
     @Parameter(type = Type.OBJECT, direction = Direction.IN)
     Instances testSet);

   @Method(declaringClass = "workflow.WorkflowImpl")
   void getBestIndex(
     @Parameter(type = Type.OBJECT, direction = Direction.INOUT)
     Report rep0,
     @Parameter(type = Type.OBJECT, direction = Direction.IN)
     Report rep1);
}
```

Fig. 4.6: Application Java interface.

The COMPSs runtime intercepts the invocations in the main code to any method contained in this interface by generating a task-dependency graph. Figure 4.7 shows an example of the resulting dependency graph of the data mining application. The red circles corresponds to *learning* tasks which forwards their results to the *evaluate* tasks represented in yellow, creating dependencies between them. All these evaluations end in a reduction process implemented using *getBestIndex* tasks, colored as blue, which find the model that minimizes the number of classification errors.

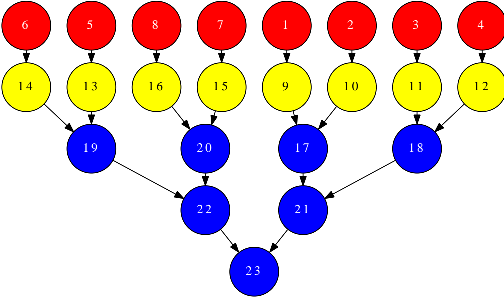


Fig. 4.7: Task dependency graph automatically generated by COMPSs.

4.4 Performance evaluation

In order to evaluate the performance of the workflow application, a set of experiments has been conducted using three different configurations: *i*) a private Cloud environment managed by Emotive Cloud[58] middleware; *ii*) a public Cloud testbed made of Azure instances; *iii*) an hybrid configuration using both private and public Clouds.

The private Cloud included a total of 96 cores available in the following way: 4 nodes with 12 Intel Xeon X5650 Six Core at 2.6GHz processors, 24GB of memory and 2TB of storage each, and 3 nodes with 16 AMD Opteron 6140 Eight Core at 2.6GHz processors, 32GB of memory and 2TB of storage each. The nodes were interconnected by a Gigabit Ethernet network and the storage was offered through a GlusterFS[59] distributed file system running in a replica configuration mode providing a total of 8TB of usable space. On this testbed 8 quad-core virtual instances with 8GB of memory and 2GB of disk space have been created running a fresh Debian Squeeze Linux distribution.

The public testbed based on Windows Azure was composed of up to 20 small virtual instances with 1.6GHz single core processor, 1.75GB of memory and 225GB of disk space each. In order to reduce the impact of data transfer on the overall execution time, the Azure's Affinity Group feature has been exploited allowing the storage and servers to be located in the same data center for performance reasons.

The *covertype*[1] dataset has been used as data source. This dataset contains information about forest cover type of a large number of sites in the United States. Each instance, corresponds to a site observation and contains 54 attributes that describe the main features of a site (e.g., elevation, aspect, slope, etc.). A subset with 290.000 instances has been taken from this dataset creating a new 36MB large one.

Table 4.1 presents the execution times and the speedup of an application run with 100 different models and up to 20 and 32 processors available in the public and private Cloud deployments respectively. Table 4.2 presents the results of the same experiment running on the hybrid Cloud scenario; in this case, Cloud outsourcing is used to expand the computing pool out of the private Cloud domain.

As depicted in Figure 4.8, execution times are similar in both cases where a single Cloud provider is used: Emotive Cloud and Azure. The speedup keeps a quasi-linear gain along the execution up to the point where the outsourcing starts. The trend changes observed in the speedup curve are not originated by the usage of outsourced resources but by a workload unbalance due to the impossibility to adjust the total number of tasks (constrained by the specific use case) to the amount of available resources. When the number of resources allows a good load balancing, the speedup curve recovers some of the lost performance as depicted in the 32+12 case where the gain is increased over the

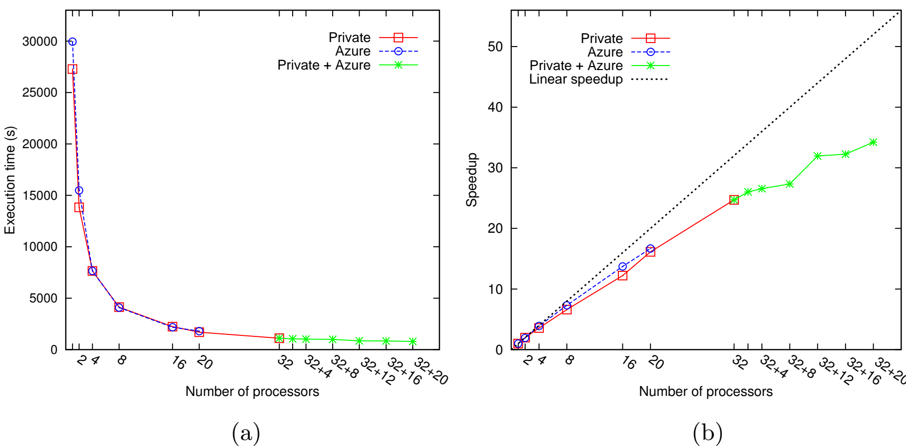
[1] http://kdd.ics.uci.edu/databases/covertype/covertype.html

N. of cores	Private Cloud (Emotive Cloud)		Public Cloud (Microsoft Azure)	
	Execution time	Speedup	Execution time	Speedup
1	7:34:41	1	8:19:05	1
2	3:50:25	1.97	4:18:04	1.93
4	2:07:35	3.56	2:07:30	3.91
8	1:08:51	6.6	1:08:15	7.31
16	0:37:13	12.22	0:36:22	13.72
20	0:28:11	16.13	0:29:55	16.68
32	0:18:24	24.71	N/A	N/A

Table 4.1: Private and public Cloud deployment execution times.

N. of cores	Private Cloud + Azure	
	Execution time	Speedup
32 + 2	0:17:29	26.01
32 + 4	0:17:07	26.56
32 + 8	0:16:38	27.34
32 + 12	0:14:14	31.94
32 + 16	0:14:06	32.25
32 + 20	0:13:17	34.23

Table 4.2: Hybrid Cloud deployment execution times.



(a) (b)

Fig. 4.8: Execution time and speedup values depending on the number of processors.

ideal line. Generally, when the workload does not depend on the application input, the COMPSs runtime scheduler is able to adapt the number of tasks to the number of available resources.

4.5 Related work

There already exist several frameworks that enable the programming and execution of applications in the Cloud and several research products are being developed to enhance the execution of applications in the Azure Platform. MapReduce [29], a widely-offered programming model, permits the processing of vast amounts of data by dividing it into many small blocks that are processed in parallel (i.e, map phase) and their results merged (i.e., reduce phase).

Hadoop[16] is an open source software platform which implements MapReduce using the Hadoop Distributed File System (HDFS). HDFS creates multiple replicas of data blocks for reliability and places them on compute nodes so they can be processed locally. Hadoop on Azure[60] is a new Apache Hadoop based distribution for Windows Server. Microsoft Daytona[61] presents an iterative MapReduce runtime for Windows Azure designed to support a wide class of data analytics and machine learning algorithms. Also Google supports MapReduce executions in its Google App Engine[62], which provides a set of libraries to invoke external services and queue units of work (tasks) for execution. Twister[63] is an enhanced MapReduce runtime with an extended programming model that supports iterative MapReduce computations efficiently. These public Cloud platforms have a high level of user's intervention in the porting of the applications requiring the use of specific APIs and the deployment and execution of the applications on their own infrastructure thus avoiding to port the code to another platform. COMPSs, on the contrary, can execute the applications on any supported Cloud provider without the need to adapt the original code to the specific target platform nor writing the map and reduce functions as in MapReduce frameworks.

Manjrasoft Aneka[26] platform provides a framework for the development of application supporting not only the MapReduce programming model but also a Task Programming and Thread Programming ones. The applications can be deployed on private or public Clouds such as Windows Azure, Amazon EC2, and GoGrid Cloud Service. The user has to use a specific .NET SDK for the porting of the code also to enact legacy code. Microsoft Generic Worker[64] has been extended in the context of the VENUS-C project to ease the porting of legacy code in the Azure platform. Even if the user does not have to change the core of the code, the creation of workflows is not automated, as is in COMPSs, in any of them; but has to be explicitly enacted through separated executions. Moreover, an application executed through the Generic Worker, can not be ported to other platforms.

4.6 Conclusions and future work

This chapter presents the extensions of COMPSs programming framework to make it able to execute e-Science applications also on the Azure Platform. The contribution include the development of a JavaGAT adaptor that allows the scheduling of COMPSs tasks on Azure instances taking care of the related data transfers, and the implementation of a set of components deployed on Azure to manage the execution of the tasks internally to the instances. The proposed approach has been validated through the execution of a data mining workflow ported to COMPSs and executed on an hybrid testbed composed of a private Cloud managed by Emotive Cloud and Azure machines. The results demonstrate that the runtime is able to manage and schedule the tasks on different infrastructures in a transparent way, keeping the overall performance of the application.

Future work includes the creation of a new connector in COMPSs to support the dynamic resource provisioning in Azure and enhancements to the Azure JavaGAT adaptor to optimize data transfers among different Clouds, and the possibility to specify input files already available on the Azure storage. The scheduling of the COMPSs runtime will be also optimized to better balance the execution of tasks taking also into account the required time to transfer data.

5

Data Mining Cloud Framework

In many application areas, Knowledge Discovery in Databases (KDD) techniques are used to extract useful knowledge from large datasets. Very often, distributed KDD approaches must be used, since datasets are too large to be analyzed in a single site, or because they are inherently distributed across many locations and cannot be moved to a central site for processing. Several distributed KDD systems have been proposed so far. In most cases, those systems had to face with infrastructure-level issues, such as resource allocation, execution management, fault tolerance, and so on.

In this chapter we study how Cloud computing technologies can be exploited to implement a distributed KDD system without worrying about low-level aspects, since they are already addressed by the Cloud infrastructure. First, we discuss the functional requirements of a generic distributed KDD system, and how these requirements can be fulfilled by a Cloud platform. Then, as a case study, we describe how we used a Cloud platform to design and develop a framework, the *Data Mining Cloud Framework*, which supports the distributed execution of KDD applications.

The Data Mining Cloud Framework supports both parameter sweeping and workflow-based KDD applications. The architecture of the system and its implementation on Windows Azure are described and discussed. In addition, some experimental results from real applications implemented on the framework are presented.

5.1 Cloud-based Data Mining

In this section we identify the main requirements that should be satisfied by a generic distributed KDD system. The system requirements are divided into *functional* and *non-functional* requirements: the former specifies which functionalities the system should provide; the latter includes quality criteria mostly related to system performance. Then, how Clouds can be exploited as effective infrastructures for handling knowledge discovery applications.

5.1.1 Functional requirements

The functional requirements that should be satisfied by a generic distributed KDD system can be grouped into two main classes: *resource management* and *application management* requirements. The former refers to requirements related to the management of all the resources (data, tools, results) that may be involved in a knowledge discovery application; the latter refers to requirements related to the design and execution of the applications themselves.

Resource management

Resources of interests in distributed KDD applications include *data sources*, *knowledge discovery tools*, and *knowledge discovery results*. Therefore, a distributed knowledge discovery system should deal with the following resource management requirements:

- **Data management** Data sources can be in different formats, such as relational databases, plain files, or semi-structured documents (e.g., XML files). The system should provide mechanisms to store and access such data sources independently from their specific format. In addition, metadata formalisms should be defined and used to describe the relevant information associated with data sources (e.g., location, format, availability, available views), in order to enable their effective access and manipulation. In particular, metadata can be exploited to index data sources and to search them based on the features of interest.
- **Tool management** Knowledge discovery tools include algorithms and services for data selection, pre-processing, transformation, data mining, and results evaluation. The system should provide mechanisms to access and use such tools independently from their specific implementation. Metadata should be used to describe the most important features of KDD tools (e.g., their function, location, usage). Hence, metadata information can be used to search for knowledge discovery tools based on users and applications needs.
- **Result management** The knowledge obtained as the result of a knowledge discovery process is represented by a data mining model. The system should provide mechanisms to store and access such models, independently from their structure and format. As for data and tools, data mining models should be described by metadata to explain and interpret their content, and to enable their effective retrieval. By retrieving the results of previous knowledge discovery tasks, we can avoid re-computation of already inferred knowledge and use the available models as input for subsequent computations.

Application management

A distributed KDD system should provide effective mechanisms to design KDD applications (*design management*) and to control their concurrent execution (*execution management*):

- **Design management** Distributed knowledge discovery applications range from simple data mining tasks, to complex data mining patterns expressed as workflows. From a design perspective, three main classes of knowledge discovery applications can be identified: *single-task applications*, in which a single data mining task such as classification, clustering, or association rules discovery is performed on a given data source; *parameter sweeping applications*, in which a dataset is analyzed using multiple instances of the same data mining algorithm with different parameters; *workflow-based applications*, in which possibly complex knowledge discovery applications are specified as graphs that link together data sources, data mining algorithms, and visualization tools. A general system should provide environments to effectively design all the above-mentioned classes of KDD applications. Particular emphasis should be put on the possibility of composing new tasks from existing tasks, e.g., designing a complex workflow starting from single tasks or basic data mining patterns, such as parallel classification or meta-learning, in turn defined as workflow-based tasks.
- **Execution management** The system should provide a distributed execution environment that supports the concurrent execution of knowledge discovery applications designed by the users. As mentioned above, applications may range from single tasks to complex knowledge discovery workflows, therefore the execution environment should cope with such a variety of applications. In particular, the execution environment should provide the following functionalities, which are related to the different phases of application execution: accessing the data sources to be mined; allocating the needed compute resources; running the application based on the user specifications, which may be expressed as a workflow; presenting the results to the user. Additionally, the system should allow users to monitor the application execution.

5.1.2 Non-functional requirements

Non-functional requirements can be defined at three levels: *user*, *architecture*, and *infrastructure*. User requirements specify how the user should interact with the system; architecture requirements specify which principles should inspire the design of the system architecture; finally, infrastructure requirements describe the non-functional features of the underlying computational infrastructure.

User requirements

From a user point of view, the following non-functional requirements should be satisfied:

- *Usability.* The system should be easy to use by the end-users, without the need of undertaking any specialized training.
- *Ubiquitous access.* Users should be able to access the system from anywhere using standard network technologies (e.g., Web sites, Web services) either from a desktop PC or from a mobile device.
- *Data protection.* Data represents a key asset for the users; therefore, the system should protect data to be mined and inferred knowledge from both unauthorized access and intentional/incidental losses.
- *On-premises data/software support.* The system should offer users the possibility to interact with on-premises data sources and software.

Architecture requirements

The main non-functional requirements at the architectural level are:

- *Service-orientation.* The architecture should be designed as a set of network-enabled software components (services) implementing the different operational capabilities of the system, to enable their effective reuse, composition, and interoperability.
- *Openness and extensibility.* The architecture should be open to the integration of new knowledge discovery tools and services. Moreover, existing services should be open for extension, but closed for modification, according to the open-closed principle.
- *Independence from infrastructure.* The architecture should be designed to be as independent as possible from the underlying infrastructure; in other terms, the system services should be able to exploit the basic functionalities provided by different infrastructures.

Infrastructure requirements

Finally, from the infrastructure perspective, the following non-functional requirements should be satisfied:

- *Standardized access.* The infrastructure should expose its services using standard technologies (e.g., Web services), to make them usable as building blocks for high-level services or applications.
- *Heterogeneous/Distributed data support.* The infrastructure should be able to cope with very large and high dimensional data sets, stored in different formats (e.g., relational databases, semi-structured documents) in a single data center, or geographically distributed across many sites.

- *Availability.* The infrastructure should be in a functioning condition even in the presence of failures that affect a subset of the hardware/software resources. Thus, effective mechanisms (e.g., redundancy) should be implemented to ensure dependable access to sensitive resources such as user data and applications.
- *Scalability.* The infrastructure should be able to handle a growing workload (deriving from larger data to process or heavier algorithms to execute) in an efficient and effective way, by dynamically allocating the needed resources (processors, storage, network). Moreover, as soon as the workload decreases, the infrastructure should release the unneeded resources.
- *Efficiency.* The infrastructure should minimize resource consumption for a given task to execute. In the case of parallel/distributed tasks, efficient allocation of processing nodes should be guaranteed. Additionally, the infrastructure should be highly utilized so to provide efficient services.
- *Security.* The infrastructure should provide effective security mechanisms to ensure data protection, identity management, and privacy.

5.1.3 Cloud for distributed KDD

A key aspect of Cloud computing is that end-users do not need to have neither knowledge nor control over the infrastructure that supports their applications. In fact, Cloud infrastructures are based on large sets of computing resources, located somewhere "in the Cloud," which are allocated to applications on-demand. Cloud resources are provided in highly scalable way, i.e., they are allocated dynamically to applications depending of the current level of requests. Although similar in overall aims to a Grid systems, Clouds are different because hide the complexity of the underlying infrastructure, providing services ready to use where end-users pay only for the resources effectively used (pay-per-use).

Clouds can be exploited as effective infrastructures for handling knowledge discovery applications. In particular, KDD services may be implemented in within each of Cloud computing service models (see /refsec:ccom-service-models):

- *KDD as SaaS*, where a single well-defined data mining algorithm or a ready-to-use knowledge discovery tool is provided as an Internet service to end-users, who may directly use it through a Web browser.
- *KDD as PaaS*, where a supporting platform is provided to developers that have to build their own applications or extend existing ones. Developers can just focus on the definition of their KDD applications without worrying about the underlying infrastructure or distributed computation issues.
- *KDD as IaaS*, where a set of virtualized resources are provided to developers as a computing infrastructure to run their data mining applications or to implement their KDD systems from scratch.

In all three scenarios listed above, the Cloud plays the role of infrastructure provider, even if at the SaaS and PaaS layers the infrastructure can be transparent to the end-user. In the following we discuss an example of a proprietary PaaS environment that can be effectively exploited to implement KDD systems and applications: Windows Azure.

Therefore, it is useful to evaluate how the infrastructure requirements for a distributed KDD system, as introduced in Section 5.1.2, can be satisfied by current Cloud platforms:

- *Standardized access.* Most Cloud platforms offer their services through standard Web service technologies. The trend is to exploit the two main classes of Web services: *REST-compliant Web services*, to query or modify resources, identified by a URI, through a predefined set of stateless operations (i.e., HTTP requests); *Arbitrary Web services*, which allow to define arbitrary operations, but at the cost of a higher invocation overhead.
- *Heterogeneous/Distributed data support.* Cloud systems provide storage services that exploit large storage facilities geographically distributed across several data centers. Most Cloud platforms offer specialized data types which are optimized to provide high scalability, flexibility, and availability. Moreover, Cloud storage services allow users to remotely access data archived in many standard data types, such as XML, HTML, and so on.
- *Availability.* Cloud platforms use replication to create and maintain multiple copies of the same data or service. Whenever a problem affects the primary instance of a service or data, a secondary instance can be restarted or accessed. In most cases, data are divided into blocks and each block is replicated on a distributed file system. Through virtualization and serialization mechanisms, also services can stored as data, hence allowing their replication and migration.
- *Scalability.* Cloud platforms are able to dynamically allocate/deallocate resources whenever the workload produced by user requests increases/decreases. This elasticity is implemented by allocating a variable number of virtual machines on a large number of physical servers, which are typically distributed on several data centers.
- *Efficiency.* In Cloud systems, efficiency is mainly achieved using virtualization mechanisms. In fact, by means of virtualization, it is possible to run multiple virtual machines on the same real server, this way achieving high levels of utilization and therefore minimizing resource consumption. Parallel task execution is also supported by Cloud systems exploiting simple but efficient models, like the popular MapReduce programming model.
- *Security.* Cloud providers offer different security features such as data protection, identity management and data privacy. The large variety of security solutions implemented by the various Cloud providers, demands for standardization initiatives to ensure that appropriate levels of security are met according with precise assessment procedures.

5.2 Data Mining Cloud Framework

This section provides a conceptual description of Data Mining Cloud Framework that is independent from specific Cloud implementations. The remainder of this section describes system model, general architecture, execution mechanisms, and user interface.

5.2.1 System Model

The model introduced here provides an abstraction to describe the characteristics of applications as they are seen in our system. For the reader's convenience, Figure 5.1 illustrates the system model entities and their interrelationships using the UML Class Diagram formalism.

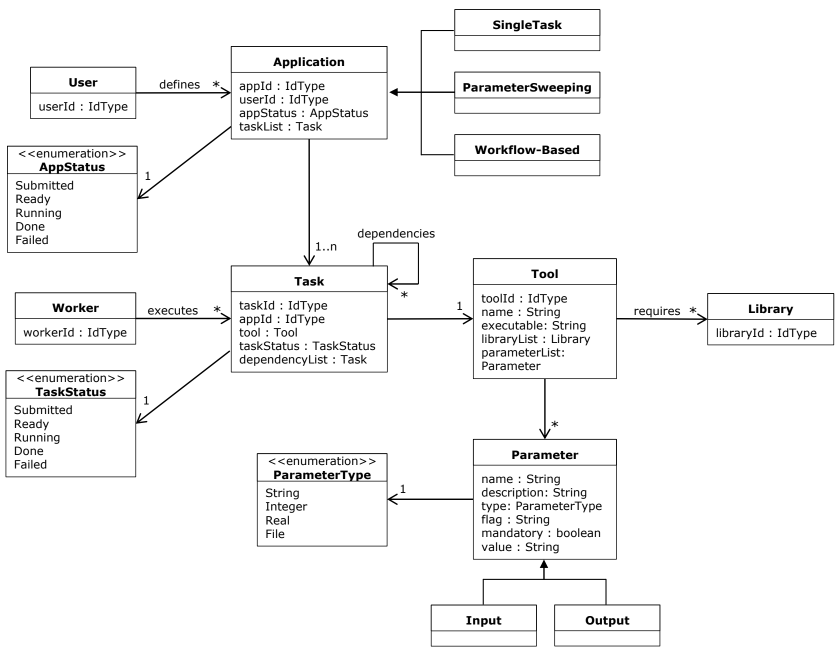


Fig. 5.1: System model described through the UML Class Diagram formalism.

An *application* is modeled as a tuple:

$$application = \langle appId,\ userId,\ appStatus,\ taskList \rangle$$

where *appId* is the application identifier, *userId* is the identifier of the user who submitted the application, *appStatus* represents the status of the application (submitted, ready, running, done, or failed), and *taskList* contains

the tasks that form the application. In case of single-task applications, the *taskList* will contain exactly one task; in case of parameter-sweeping applications, the *taskList* will contain multiple independent tasks; finally, the *taskList* of workflow-based applications will include multiple tasks with dependencies.

A *task* is modelled as a tuple:

$$task = \langle taskId, appId, tool, taskStatus, dependencyList \rangle$$

where *taskId* is the task identifier, *appId* is the identifier of the application the task belongs to, *tool* is a reference to the tool to be executed, *taskStatus* represents the task status (see Table 5.1), and *dependencyList* contains the identifiers of the other tasks this task depends on. A Task T_j depends on a task T_i (i.e., $T_i \rightarrow T_j$) if T_j can be executed only after that T_i has successfully completed its execution. Thus, the *dependencyList* of a task T_j contains a set of n tasks $T_1 ... T_n$ such that $T_i \rightarrow T_j$ for each $1 \leq i \leq n$.

Status	Description
submitted	The task has been created after application submission.
ready	The task is ready for execution, i.e., there are no tasks it depends on.
running	The task is currently being processed.
done	The task has completed successfully.
failed	The task has completed with failure.

Table 5.1: Task statuses.

A *tool* is defined as follows:

$$tool = \langle toolId, name, executable, libraryList, parameterList \rangle$$

where *toolId* is the tool identifier, *name* is a descriptive name for the tool, *executable* is a reference to the executable (program or script) that launches the tool, *libraryList* contains the references of the required libraries, and *parameterList* is a list of parameters used to configure the use of the tool.

A *parameter* is defined as a tuple:

$$parameter = \langle name, description, type, flag, mandatory, value \rangle$$

where *name* is the parameter name, *description* is a parameter description, *type* specifies the parameter type (e.g., string, integer, etc.), *flag* is a string that precedes the parameter value to allow its identification in a command line invocation, *mandatory* is a boolean that specifies whether the parameter is mandatory or not, *value* contains the parameter value. Note that a parameter can be either an *Input* or an *Output* parameter.

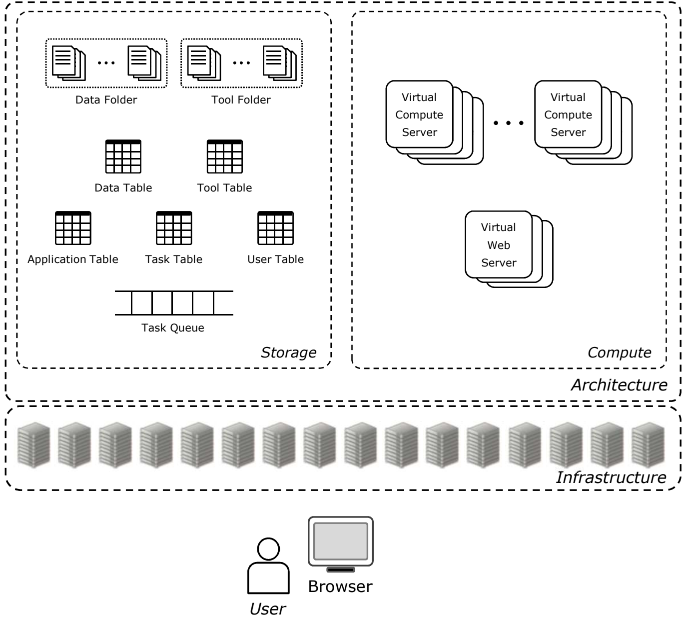


Fig. 5.2: System components.

5.2.2 General architecture

The Data Mining Cloud Framework architecture includes different kinds of components that can be grouped into storage and compute components (see Figure 5.2).

The storage components include:

- A *Data Folder* that contains data sources such as relational databases, plain files, or semi-structured documents and also the the results of knowledge discovery processes. Similarly, a *Tool folder* contains libraries and executable files for data selection, pre-processing, transformation, data mining, and results evaluation.
- *Data Table* and *Tool Table* contain metadata information associated with data sources and tools respectively.
- *Application Table*, *Task Table*, and *Users Table* keeps information about applications, tasks and users.
- The *Task Queue* contains the tasks ready to be executed.

The compute components are:

- A pool of *Virtual Compute Servers*, which are in charge of executing the data mining tasks submitted by users.

- A pool of *Virtual Web Servers* host the *Website*, by allowing users to submit, monitor the execution, and access the results of their data mining tasks.

5.2.3 Execution mechanisms

The following steps are performed to develop and execute a knowledge discovery application through the system:

1. A user accesses the Website and develop her/his application (either single-task, parameter-sweeping, or workflow-based) through a Web-based interface. After completing the application, she/he can submit it for execution.
2. After the application submission, a set of tasks are created and inserted into the Task Queue on the basis of the application submitted by the user.
3. Each idle Virtual Compute Server picks a task from the Task Queue, and starts its execution on a virtual server.
4. Each Virtual Compute Server gets the input dataset from the location specified by the application. To this end, a file transfer is performed from the Data Folder where the dataset is located, to the local storage of the Virtual Compute Server.
5. After task completion, each Virtual Compute Server puts the result on the Data Folder.
6. The Website notifies the user as soon as her/his task(s) have completed, and allows her/him to access the results.

The set of tasks created on the second step depends on the type of application submitted by the user. In the case of a single-task application, just one data mining task is inserted into the Task Queue. If the user submits a parameter-sweeping application, one task for each combination of the input parameters values is executed [1]. In the case of a workflow-based application, the set of tasks created depends on how many data mining tools are invoked within the workflow; initially, only the workflow tasks without dependencies are inserted into the Task Queue.

The actions performed by each Virtual Compute Server are detailed in Figure 5.3. The Virtual Compute Server cyclically checks whether there are tasks ready to be executed in TaskQueue. If so, the first task is taken from the queue (line 3) and its status is changed to 'running' (line 4). Two local folders are created to temporarily stage input data and tools, which include both executables and libraries (lines 5-6). Input and output lists are created on lines 7-13. Then, the transfer of all the needed input resources (files, executables and libraries) is performed (lines 14-18). At line 19, the Virtual Compute Server locally executes the *task* and waits for its completion.

[1] In general, the number of tasks is given by $\prod_{i=1}^{n} v_i$, where n is the number of input parameters and v_i is the number of values assumed by the ith parameter

If the *task* is 'done' (line 20), the output results are copied to a remote data folder (lines 21-22), and the *task* status is changed to 'done' also in the Task Table (line 23). Then, for each task *apptask* that belongs to the same application of *task* (line 24), if *apptask* has a dependency with *task* (line 25), that dependency is deleted (line 26). If *apptask* remains without dependencies (line 27), it becomes 'ready' and is added to the Task Queue (lines 28-29). If the *task* has failed (line 30), all the tasks that directly or indirectly depend on it will be marked as 'failed' (lines 31-40). Finally, the task is removed from the Task Queue (line 41), and the local data and tools folders are deleted (lines 42-43).

```
1  while true do
2      if TaskQueue.isNotEmpty() then
3          task ← TaskQueue.getTask();
4          TaskTable.update(task, 'running');
5          localDataFolder = <local data folder>;
6          localToolFolder = <local tool folder>;
7          inputList = <empty list>;
8          outputList = <empty list>;
9          foreach parameter in task.tool.parameterList do
10             if parameter is Input then
11                 inputList.add(parameter)/;
12             else
13                 outputList.add(parameter)/;
14         foreach input in inputList do
15             transfer(input, DataFolder, localDataFolder);
16         transfer(task.tool.executable, localToolFolder);
17         foreach library in task.tool.libraryList do
18             transfer(library, ToolFolder, localToolFolder);
19         taskStatus ← execute(task, localDataFolder, localToolFolder);
20         if taskStatus = 'done' then
21             foreach output in outputList do
22                 transfer(output, localDataFolder, DataFolder);
23             TaskTable.update(task, 'done');
24             foreach appTask in TaskTable.getTasks(task.appId) do
25                 if appTask.dependencyList.contains(task) then
26                     appTask.dependencyList.remove(task);
27                     if appTask.activeIncomingEdges.isEmpty() then
28                         TaskTable.update(appTask, 'ready');
29                         TaskQueue.addTask(appTask);
30         else
31             failedTasks = <empty set>;
32             tasksToAnalyze = <empty set>;
33             tasksToAnalyze.add(task);
34             while tasksToAnalyze.isNotEmpty() do
35                 tmpTask = tasksToAnalyze.remove();
36                 TaskTable.update(tmpTask, 'failed');
37                 failedTasks.add(tmpTask);
38                 foreach appTask in TaskTable.tasks(task.appId) do
39                     if failedTasks.notContains(appTask)
40                         && appTask.dependencyList.contains(tmpTask) then
40                         tasksToAnalyze.add(appTask);
41         TaskQueue.remove(task);
42         delete(localDataFolder);
43         delete(localToolFolder);
44     else
45         sleep(sleepTime);
```

Fig. 5.3: Cyclic operations performed by each Virtual Compute Server.

5.2.4 User Interface

As mentioned earlier, the Website allows a user to submit, monitor the execution, and access the results of the data mining tasks. The Website includes three main sections:

- *App submission* that allows users to submit single-task, parameter sweeping, or workflow-based applications;
- *App monitoring* that is used to monitor the status of submitted applications and to access results; the workflow-based applications submitted.
- *Data/Tool management* that allows users to manage input/output data and tools.

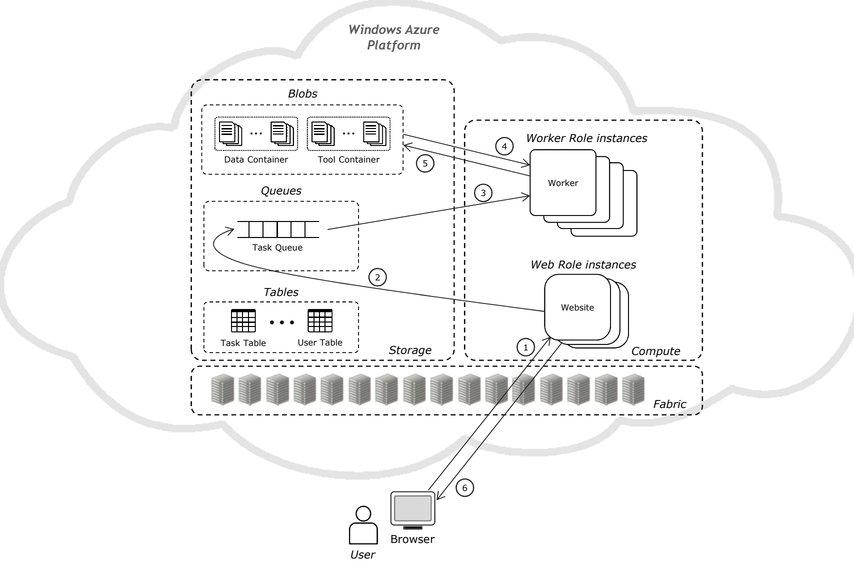


Fig. 5.4: Architecture of the Data Mining Cloud Framework.

5.3 Implementing the Data Mining Cloud Framework

We discuss in this section how our system was implemented on the Azure platform (see 2.3.1). First, we describe how Azure components can fulfill the functional requirements of a generic distributed KDD system (Section 5.3.1). Then, we describe how the generic components of our architecture are mapped to the components provided by Azure (Section 5.3.2). Finally, we introduce the Web interface of our system to illustrate the main functionalities provided to the users (Section 5.2.4).

5.3.1 Fulfilling the functional requirements with Azure

Based on our study summarized in Table 5.2, the Azure components and mechanisms can be effectively exploited to fulfill the requirements of a generic distributed KDD system that have been introduced in Section 5.1.1. We exploited these components and mechanisms to implement the Data Mining Cloud Framework described in the next section.

KDD system requirements		*Azure components*
Resource management	Data	- *Different data formats*: Binary large objects (Blobs); non-relational tables (Tables); queues for communication data (Queues); relational databases (SQL Database). - *Metadata support*: Tables/SQL Azure Databases to store data descriptions; custom description fields can be added to Blobs containing data sources.
	Tools	- *Implementation-independent access*: Tools can be exposed as Web services. - *Metadata support*: Tables/SQL Databases to store tools descriptions; custom description fields can be added to Blobs containing binary tools; WSDL descriptions for Web services.
	Results	- *Models storing*: Blobs to store results either in textual or visual form. - *Metadata support*: Tables/SQL Databases to describe models format; custom description fields can be added to Blobs containing data mining models.
Application management	Design	- *Single-task applications*: Programming the execution of a single Web service or binary tool on a single Worker role instance. - *Parameter sweeping applications*: Programming the concurrent execution of a set of Web services or binary tools on a set of Worker role instances. - *Workflow-based applications*: Programming the coordinated execution of a set of Web services or binary tools on a set of Worker role instances.
	Execution	- *Storage resources access*: Managed by the Storage layer. - *Compute resources allocation*: Managed by the Compute layer. - *Application execution and monitoring*: Web services/Worker role instances to run the single tasks; Tables to store tasks status information; Web role instance to present monitoring information. - *Results presentation*: Blobs/Tables to store/interpret the inferred models; Web role instance to present results.

Table 5.2: How Azure components can fulfill the functional requirements of a generic distributed KDD system

5.3.2 Implementing the system components on Azure

As shown in Figure 5.2, the architecture of our framework distinguishes its high-level components into two groups, *Storage* and *Compute*, following the same approach followed by Azure and other Cloud platforms. In this way, we were able to implement our Storage and Compute components by fully exploiting the Storage and Compute components and functionalities provided

by Azure. In particular, for the Storage components, we adopted the following mapping:

- *Data Folder* and *Tool Folder* are implemented as Blob containers.
- *Data Table*, *Tool Table*, *Application Table*, *Task Table*, and *Users Table* are implemented as non-relational Tables.
- The *Task Queue* is implemented as an Azure's Queue.

For the Compute components, the following mapping with Azure was adopted:

- The *Virtual Compute Servers* are implemented as *Worker Role* instances.
- The *Virtual Web Servers* are implemented as *Web Role* instances. Figure 5.4 shows the architecture of the Data Mining Cloud Framework, as it is implemented on Windows Azure.

Figure 5.4 shows how Azure components can be exploited to performs the steps described in Section 5.3. Each Worker Role instance executes the operations described by Algorithm 5.3. This requires file transfers to be performed when input/output data have to be moved between storage and servers. To reduce the impact of data transfer on the overall execution time, we exploit the Azure's *Affinity Group* feature, which allows storage and servers to be located near to each other in the same data center for optimal performance. The Web Role instances allow users to submit and manage their data mining applications, as detailed in the next section.

5.4 Parameter-sweeping data mining applications

In the following, we focus on task submission and management, by describing how the Data Mining Cloud Framework Website is used to submit a parameter sweeping data mining application.

After logging into the Website, a user goes the App Submission/Par. sweeping menu, and select the algorithm to be used (see Figure 5.5). A list of the available algorithms is shown to the user, who selects the one of interest. In the example, the *K-Means* clustering algorithm [65] from the Weka library is selected.

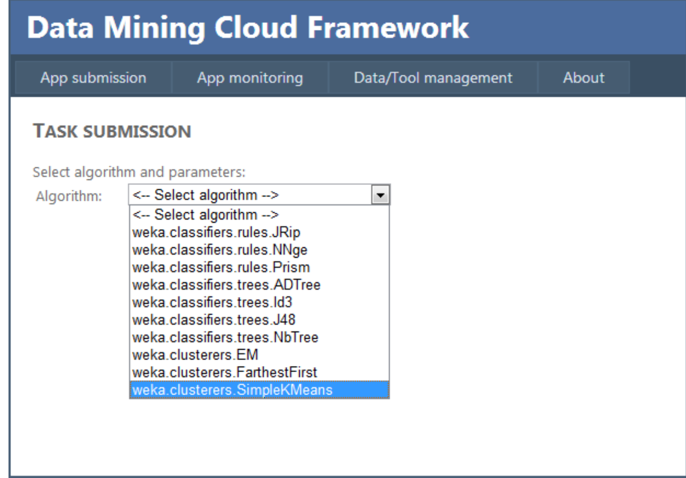


Fig. 5.5: Selection of the data mining algorithm.

As soon as the algorithm has been selected, the Website shows to the user a form with the relevant parameters that he/she can specify for the algorithm (see Figure 5.6). For K-Means, besides the input dataset, the relevant parameters are the *number of clusters* and the *seed*. The user can choose whether to sweep or not a certain parameter. In the example, the user chose to sweep both the number of clusters and the seed. For the former, a range of values is specified, while for the latter, a list of values is provided.

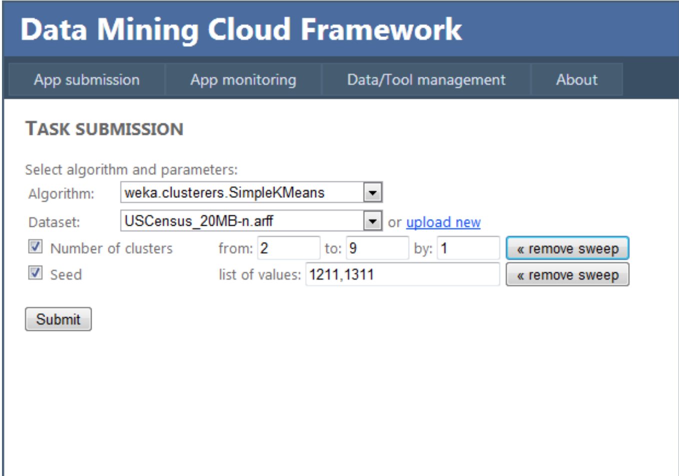


Fig. 5.6: Choice of the algorithm parameters.

After submission, the system generates a number of independent tasks that are executed on the Cloud as discussed earlier. The user can monitor the status of each single task through the Task Status section of the Website (see Figure 5.7). For each task, the current status (submitted, running, done or failed) and status update time are shown. Moreover, for each task that has completed its execution, the system enables two links: the first one (*Stat*) gives access to a file containing some statistics about the amount of resources consumed by the task; the second one (*Result*) visualizes the task result.

Data Mining Cloud Framework

App submission | App monitoring | Data/Tool management | About

TASK STATUS

Task ID	CurrentStatus	StatusUpdateTime	Statistics	Result	Archive
1634454118824362358-001	done	7/4/2011 7:34:08 PM	Stat	Result	
1634454118824362358-002	done	7/4/2011 7:33:10 PM	Stat	Result	
1634454118824362358-003	done	7/4/2011 7:34:00 PM	Stat	Result	
1634454118824362358-004	done	7/4/2011 7:34:19 PM	Stat	Result	
1634454118824362358-005	running	7/4/2011 7:33:11 PM	Stat	Result	
1634454118824362358-006	running	7/4/2011 7:34:00 PM	Stat	Result	
1634454118824362358-007	running	7/4/2011 7:34:09 PM	Stat	Result	
1634454118824362358-008	running	7/4/2011 7:34:20 PM	Stat	Result	
1634454118824362358-009	submitted	7/4/2011 7:32:14 PM	Stat	Result	
1634454118824362358-010	submitted	7/4/2011 7:32:15 PM	Stat	Result	

Fig. 5.7: Task status monitoring.

5.5 Workflow-based data mining applications

Here we show how the Web interface is used to design and execute workflow-based data mining applications.

We prototyped the programming interface and its services to support the composition and execution of workflow-based knowledge discovery applications in our Cloud framework. Following the approach proposed in [66] and extending it, we model a knowledge discovery workflow as a graph whose nodes represent resources (datasets, data mining tools, data mining models), implemented as Cloud services, and whose edges represent dependencies between resources. To support the workflow composition, we implemented a Website section that, using native HTML 5 features, allows users to design service-oriented knowledge discovery workflows with a simple drag-and-drop approach.

5.5.1 Workflow formalism

In our framework a workflow is a directed acyclic graph whose nodes represent resources and whose edges represent the dependencies among the resources. The graph is bipartite and composed by two types of resources (graphically depicted by the icons shown in Figure 5.8):

- *Data* node, which represents an input or output data source. Two subtypes exist: *Dataset*, which represents a data collection, and *Model*, which represents a model generated by a data mining tool (e.g., a decision tree, a set of association rules).
- *Tool* node, which represents a tool performing any kind of operation that can be applied to a data node (data mining, filtering, splitting, voting operations, etc.).

Fig. 5.8: Nodes types.

When a node is created in a workflow, a label with a unique name is attached below the corresponding icon. In order to ease the workflow composition and to allow users to monitor its execution, each resource icon has a symbol representing the status in which the corresponding resource is at a given time. The resource statuses can be divided into two categories:

- *Composition-time* statuses describe the resource during the workflow composition.
- *Run-time* statuses describe the resource during the workflow execution.

Table 5.3 and 5.4 show the composition and run-time statuses of Data and Tool resources.

Time	Symbol	Status	Description
Composition time		Undefined	The Data node does not refer to any data sources in the Cloud.
		Defined	The Data node refers to a data source available in the Cloud.
		Not Well Defined	The Data node is not well defined.
Run time		Not Yet Available	The data source has not yet been generated.
		Available	The data source has been created or was already present.
		Not Available	The data source has not been correctly generated.

Table 5.3: Data composition and run time statuses.

Time	Symbol	Status	Description
Composition time		Undefined	The Tool node does not refer to any data source.
		Defined	All Tool parameters and input/output edges have been correctly defined.
		Not Well Defined	Some Tool parameters or input/output edges have not been correctly defined.
Run time		Submitted	A task has been created starting from the Tool definition.
		Ready	The task is ready for execution.
		Running	The task is currently being processed.
		Done	The task has completed successfully.
		Failed	The task has completed with failure.

Table 5.4: Tool composition and run time statuses.

The nodes can be connected with each other through direct edges, establishing specific dependency relationships among them. 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. For example, Figure 5.9 shows a *J48* Tool (a Java implementation of the C4.5 algorithm [57]) that takes in input a *TrainSet* and generates a *Model*.

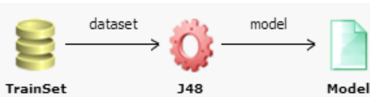


Fig. 5.9: J48 Tool connected to an input dataset and an output model.

For each Tool node, input/output connections are allowed on the basis of the Tool definition, which is stored Tool Table. An example of Tool definition (in JSON format) is shown in Figure 5.10. In this example, based on the Tool definition, the system allows the creation of a single edge from an input Data node with label *dataset*, and an edge to an output Data node with label *model*. In addition, the Tool definition states that the user can specify a single input parameters (the *confidence value*).

```
"J48": {
    "libraryList": ["java.exe","weka.jar"],
    "executable": "java.exe -cp weka.jar weka.classifiers.trees.J48",
    "parameterList": [{
        "name": "input",
        "flag": "-t",
        "mandatory": true,
        "parType": "IN",
        "type": "file",
        "array": false,
        "description": "Input Dataset"
    },{
        "name": "confidence",
        "flag": "-C",
        "mandatory": false,
        "parType": "OP",
        "type": "real",
        "array": false,
        "description": "Confidence value",
        "value": "0.25"
    },{
        "name": "model",
        "flag": "-d",
        "mandatory": true,
        "parType": "OUT",
        "type": "file",
        "array": false,
        "description": "Output model"
    }]
}
```

Fig. 5.10: Example of Tool definition.

A user can define the relevant information associated with each workflow node through a configuration pop-up panel that appears selecting the node, as shown in Figure 5.11.

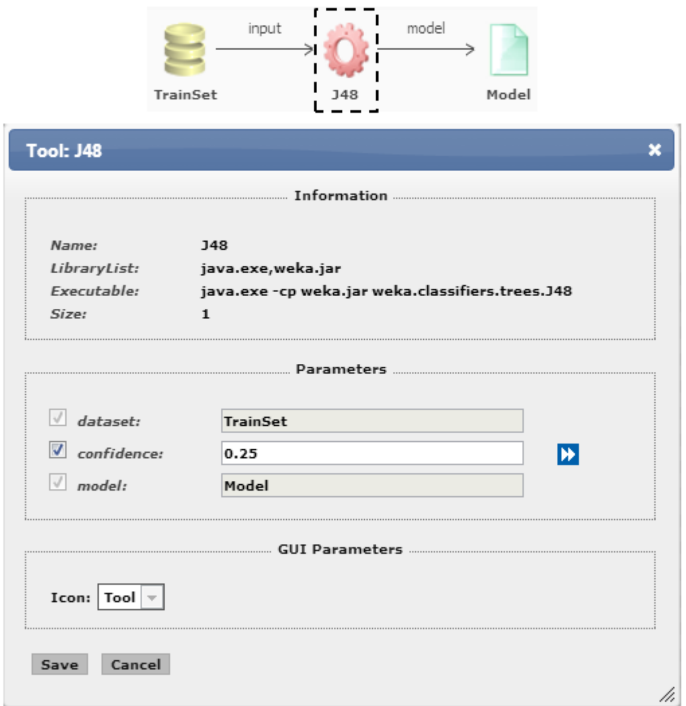


Fig. 5.11: Pop-up panel that shows the user-definable parameters for the J48 Tool.

Data and Tool nodes can be added to the workflow singularly or in array form, as shown in Figure 5.12.

Fig. 5.12: Single and array nodes types.

A Data array is an ordered collection of input/output data sources (e.g., collection of dataset to be analyzed, collection of models that a classifier can use to classify a dataset).

A Tool array represents multiple instances of the same tool; it can be used to execute two types of operations:

- *Parameter sweeping*, where a data source is analyzed using multiple instances of the same tool with different parameters. In this case, the size of the Tool array is equal to the number of different combinations of the parameters values.
- *Input sweeping*, where each element of a Data array is analyzed by the corresponding element of a Tool array. This requires the Tool array and Data array have the same size.

Figure 5.13 (a) shows an example of parameter sweeping. A Tool array, labeled as $J48[3]$, represents three instances of the J48 tool, where each instance is configured to use a different confidence value. Each instance of J48 analyzes the same input $TrainSet$; as a result, three independent classification models, denoted as $Model[3]$, are generated. In short, $J48[i]$ takes in input $TrainSet$ and generates $Model[i]$, for $1 \leq i \leq 3$.

An example of input sweeping is shown in Figure 5.13 (b). In this case, three instances of J48 ($J48[3]$), configured to use the same parameters, analyze in parallel three training sets ($TrainSet[3]$). In other terms, $J48[i]$ takes in input $TrainSet[i]$ to produce $Model[i]$, for $1 \leq i \leq 3$.

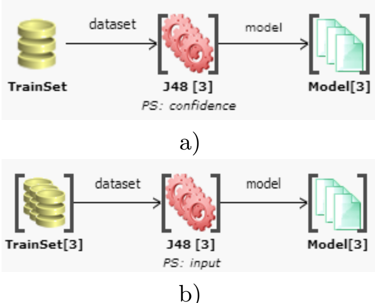


Fig. 5.13: a) Example of parameter sweeping; b) Example of input sweeping.

5.5.2 Workflow composition

Here we use a data mining application composed of several sequential and parallel steps as an example for presenting the main features of the visual programming interface of the Data Mining Cloud Framework.

Figure 5.14 shows a screenshot of the Website taken at the beginning the composition of a knowledge discovery workflow.

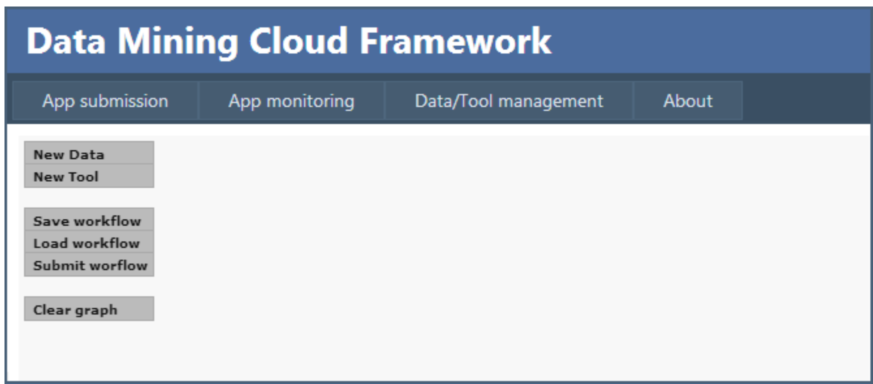


Fig. 5.14: Workflow interface.

On the top-left of the window, some buttons allow user to insert a new data or tool nodes into the workflow and also to save, load or submit a workflow. Once placed into the workflow, a node can be linked to others to establish the desired dependencies as described in the previous section.

The example application analyses a dataset by using n instances of the J48 classification algorithm that work on n partitions of the training set and generate n knowledge models. By using the n generated models and the test set n classifiers produce in parallel n classified datasets (n classifications). In the final step of the workflow, a voter generates the final dataset by assigning a class to each data item, choosing the class predicted by the majority of the models [67].

Figure 5.15 shows a snapshot of the visual interface where the first step of the workflow is designed. In particular, we can see the splitting of the original dataset in training and test set operated by a partitioning tool. A set of configuration parameters are associated with each workflow node. The parameters of a given node can be specified through a pop-up panel that appears when that node is selected. For example, the right part of Figure 5.15 shows the configuration panel for the partitioning tool. In this case, only one parameter can be specified, namely which percentage of the input dataset must be taken to produce the training set.

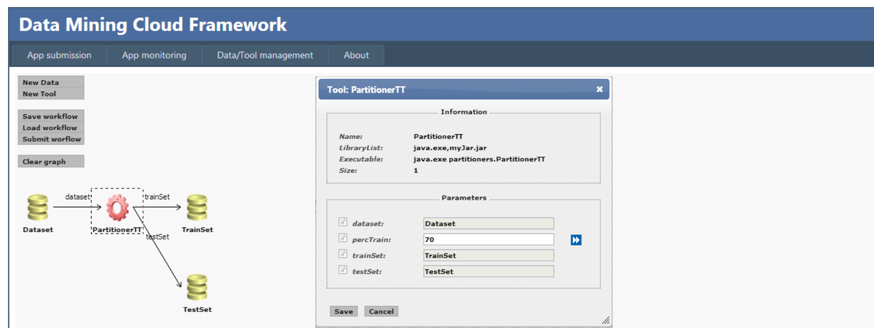


Fig. 5.15: First step: The input dataset is partitioned into train and test set.

As a second step, the training set is partitioned into 10 parts using another partitioning tool (see Figure 5.16). The 10 training sets resulting from the partitioning are represented in the workflow as a single data array node, labeled as $TrainSetPart[10]$.

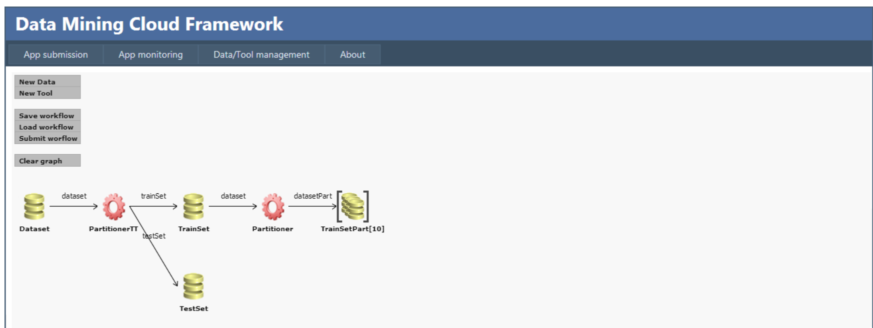


Fig. 5.16: Second step: The train set is partitioned into 10 parts.

Figure 5.17 shows the third step of the workflow, in which the 10 training sets are analyzed in parallel by 10 instances of the J48 classification algorithm, to produce the same number of classification models. A tool array node, labeled as $J48[10]$, is used to represent the 10 instances of the J48 algorithm, while another data array node, labeled as $Model[10]$, represents the models generated by the classification algorithms. In practice, this part of the workflow specifies that$J48[i]$ takes in input $TrainSetPart[i]$ to produce $Model[i]$, for $1 \leq i \leq 10$.

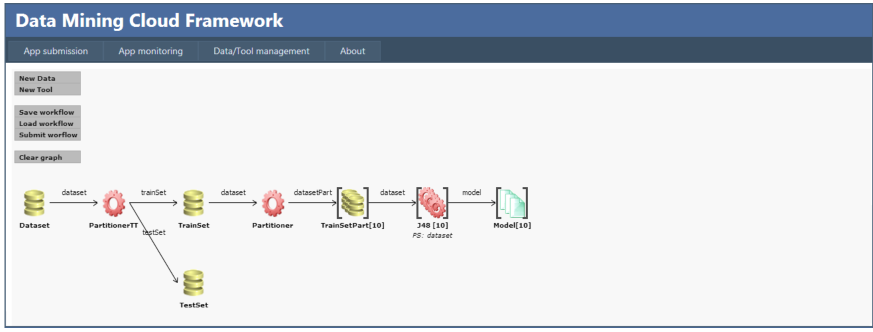


Fig. 5.17: Third step: Each train set part is analyzed by an instance of the J48 classification tool; as a result, 10 classification models are produced.

The fourth step classifies the test set using the 10 models generated on the previous step (see Figure 5.18). The classification is performed by 10 classifiers that run in parallel to produce 10 classified test sets. More in detail, $Classifier$[i] takes in input $TestSet$ and $Model$[i] to produce $ClassTestSet[i]$, for $1 \leq i \leq 10$.

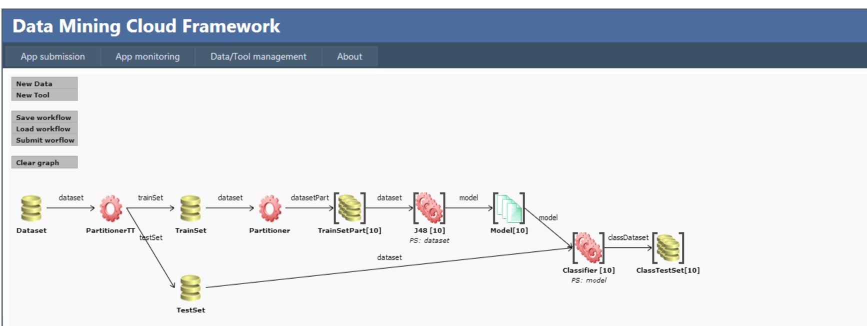


Fig. 5.18: Fourth step: The test set is classified by 10 classifiers, which perform the tasks using the 10 classification models generated on the previous step.

As the last step, the 10 classified test sets are passed to a voter tool that produces the final dataset, labeled as $FinalClassTestSet$ (see Figure 5.19).

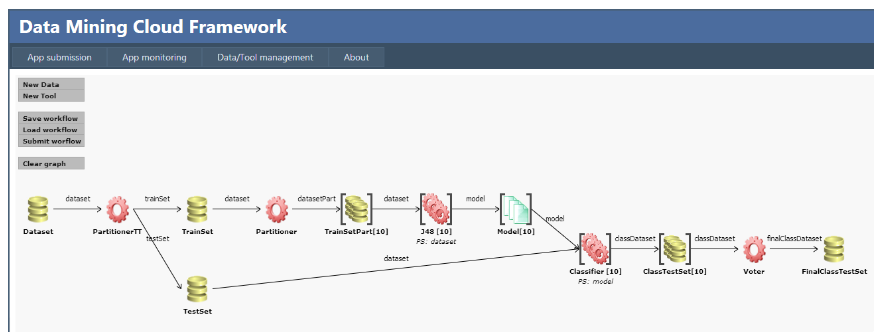


Fig. 5.19: Fifth step: The 10 instances of the classified test set are compared instance-by-instance by a voter to produce the final classified test set.

When the workflow is complete, it can be submitted for execution. User can submit a workflow pressing the *Submit workflow* button in the toolbar. After that, the workflow execution starts and proceeds as detailed in the next section.

5.5.3 Workflow execution

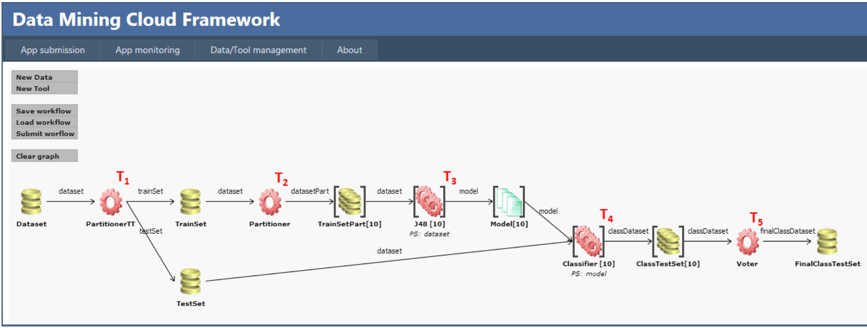


Fig. 5.20: Workflow application and its tasks.

The workflow defined in the previous section includes five tools (PartitionerTT, Partioner, J48, Classifier, and Voter), which are translated into five tasks, indicated as $\{T_1...T_5\}$, as shown in Figure 5.20. Differently from parameter-sweeping applications whose tasks are independent each other and therefore can be executed in parallel, the execution order of workflow tasks depends on the dependencies specified by the workflow edges. To ensure the correct execution order, each task is associated with a list of tasks that must be completed before starting its execution. Figure 5.21 shows a possible order

in which the tasks are generated and inserted into the Task Queue. For each task, the list of tasks to be completed before its execution is included. Note that task T_3, which represents the execution of ten instances of J48, is translated into ten sub-tasks $T_3[1]...T_3[10]$. Similarly, T_4 is translated into sub-tasks $T_4[1]...T_4[10]$.

According with the tasks dependencies specified by the workflow, the execution of T_2 will start after completion of T_1. As soon as T_2 completes, the ten sub-tasks that compose T_3 can be run concurrently. Each sub-task $T_4[i]$ can be executed only after completion of both T_2 and $T_3[i]$, for $1 \leq i \leq 10$. Finally T_5 will start after completion of all sub-tasks that compose T_4.

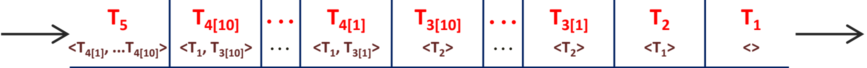


Fig. 5.21: A possible order in which the tasks are generated and inserted into the Task Queue.

Figure 5.22 shows a snapshot of the workflow taken during its execution. The figure shows that PartitionerTT has completed the execution, Partioner is running, while the other tools are submitted.

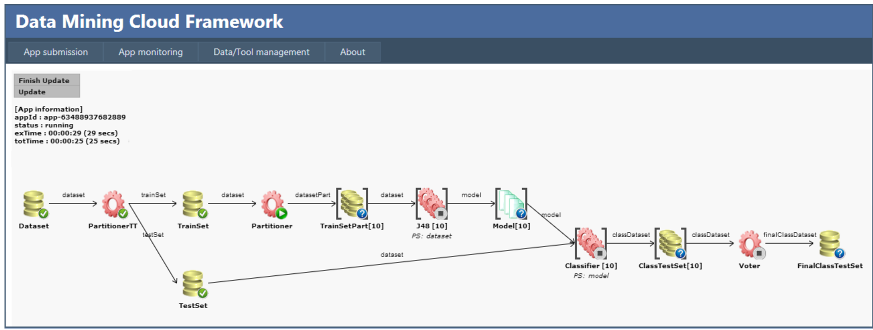


Fig. 5.22: The final workflow during its execution.

Figure 5.23 shows the workflow after the completion of its execution. Some statistics about the overall application are shown on the upper left part of the window. In this example, it is shown that, using 10 virtual machines, the workflow completed 264 seconds after its submission, with a total execution time (i.e., the sum of the execution times of all the tasks) of 1604 seconds.

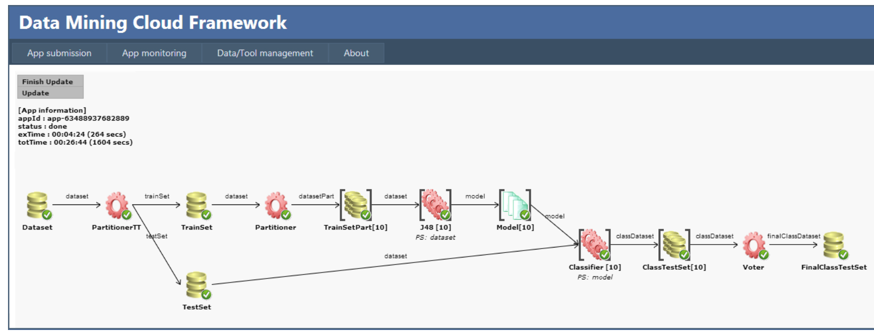


Fig. 5.23: The final workflow after completion of its execution.

5.6 Experimental results

Here we describe some data mining applications designed and executed with the Data Mining Cloud Framework. The Cloud environment was composed by 1 virtual server for the Web role instance (which hosts the Website), and up to 19 virtual servers for the Worker Role instances. Each virtual server was equipped with a single-core 1 GHz CPU, 0.75 GB of memory, and 20 GB of disk space, with a cost of $0.05 per hour. Each test has been executed by varying both the size of the input dataset and the number of virtual servers used to run the application. As performance indicators, we used the *turnaround time*, the achieved *speedup*, and the total *cost* paid.

5.6.1 Parameter sweeping data mining applications

In this section we present performance results obtained by executing *clustering* and *classification* parameter sweeping data mining applications on a set of publicly available datasets.

Clustering Application

The clustering application discussed here is the same introduced in Section 5.4 to describe the user interface for defining and executing parameter sweeping applications.

As input data source we used the *US Census 1990*'s dataset[2] from the UCI KDD archive [68], which contains part of US 1990's census information. Each tuple in the dataset contains information about a US citizen, consisting of 68 categorical attributes (e.g., sex, age). The original dataset is composed of about 2,458,000 instances, stored in a file of 345 MB. In order to evaluate the system with increasing workloads, we extracted three datasets with size

[2] http://kdd.ics.uci.edu/databases/census1990/USCensus1990.html

of 20 MB, 40 MB and 80 MB, with 143,000, 286,000 and 572,000 tuples, respectively.

For each of the three datasets, we submitted the execution of the K-Means clustering algorithm with the following swept parameters: *number of clusters* (N) from 2 to 9; *seed* (S) equal to 1211 or 1311. We have 8 different values for N, which combined with the 2 values of S generate 16 configurations. Therefore, for each dataset size, the Data Mining Cloud Framework executed 16 independent tasks.

N. of servers	20MB dataset		40MB dataset		80MB dataset	
	Turnaround time	Cost	Turnaround time	Cost	Turnaround time	Cost
1	0:51:37	$0.04	2:10:37	$0.11	13:43:45	$0.69
2	0:27:25	$0.04	1:07:23	$0.11	7:49:33	$0.72
4	0:14:22	$0.04	0:40:02	$0.11	4:14:39	$0.65
8	0:08:55	$0.04	0:23:11	$0.11	2:28:14	$0.65
16	0:05:43	$0.04	0:17:07	$0.11	2:09:32	$0.69

Table 5.5: Turnaround times and costs for the parameter sweeping clustering application.

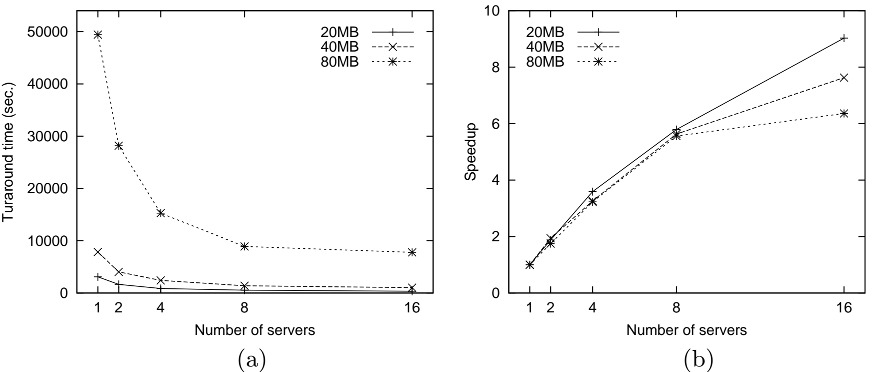


Fig. 5.24: Turnaround times and speedup values for the parameter sweeping clustering application by varying number of virtual servers and dataset size.

Table 5.5 presents the turnaround times and costs of the clustering application when 1, 2, 4, 8 and 16 virtual servers are used. The turnaround time includes the file transfer overhead (copying the input dataset from a Blob to the local storage of a virtual server). In our experiments this overhead was very low (only a few seconds) even with the largest of the three datasets,

due to the use of the Affinity Group feature provided by Azure, as already discussed in Section 2.3.1.

For the smallest dataset (20 MB), the turnaround time decreases from about 52 minutes obtained with a single server, to about 6 minutes using 16 servers. For the medium-size dataset (40 MB), the turnaround time passes from 2.2 hours to 17 minutes, while for the largest dataset (80 MB), the turnaround time ranges from 13.7 hours to 2.2 hours. As shown in the table, for a given dataset size, the total cost paid does not change with the number of virtual servers used, because the total execution time does not vary significantly by changing configuration.

Fig. 5.24 shows turnaround times and speedup values obtained by varying number of virtual servers and dataset size. For the 20 MB dataset, the speedup passes from 1.9 using 2 servers to 9.0 using 16 servers. For the 40 MB dataset, the speedup ranges from 1.9 to 7.6. Finally, with the 80 MB dataset, we obtained a speedup ranging from 1.8 to 6.4.

The speedup achieved does not increase linearly with the number of servers used since the 16 clustering tasks are very heterogeneous in terms of execution times. In fact, the turnaround time is bound to the execution time of the slowest task instances. In our case, the slowest task instances are those in charge of grouping data into a larger number of clusters (e.g., $N = 9$), which are much slower (up to six times) than those with small values of N.

Classification Application

In this second set of experiments, we use the Data Mining Cloud Framework to run a parameter sweeping classification.

The dataset *covertype* [3], has been used as data source. This 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 72 MB. 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. As the classification algorithm we used J48.

For each dataset size, we submitted to the Data Mining Cloud Framework the execution of the J48 algorithm, by sweeping its *confidence value* parameter from 0.05 to 0.50 with a step of 0.03, which produces 16 different tasks. Table 5.6 shows the turnaround times of the clustering application when 1, 2, 4, 8 and 16 virtual servers are used.

For the 9 MB dataset the turnaround time decreases from 3.9 hours obtained with a single server, to about 16 minutes using 16 servers. For the 18

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

N. of servers	9MB dataset		18MB dataset		36MB dataset	
	Turnaround time	Cost	Turnaround time	Cost	Turnaround time	Cost
1	3:53:15	$0.19	11:46:14	$0.59	41:07:17	$2.06
2	2:11:05	$0.19	5:53:56	$0.59	23:04:56	$2.06
4	1:00:05	$0.19	2:59:16	$0.59	10:18:40	$1.99
8	0:30:34	$0.19	1:30:01	$0.58	5:25:04	$2.03
16	0:16:12	$0.20	0:48:30	$0.60	2:52:44	$2.10

Table 5.6: Turnaround times and costs for the parameter sweeping classification application.

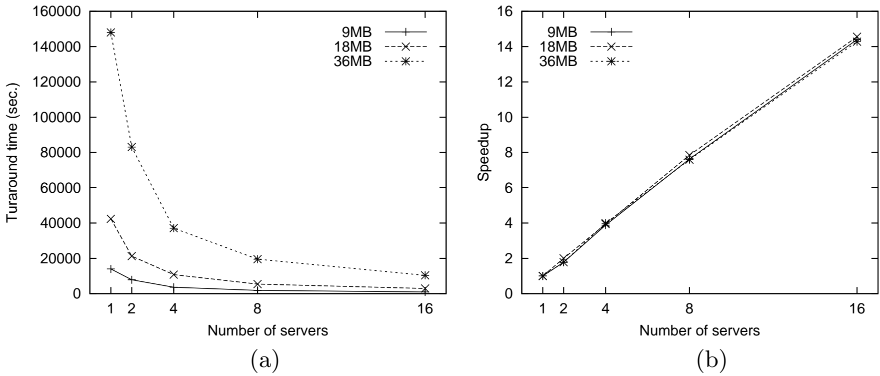


Fig. 5.25: Turnaround times and speedup values for the parameter sweeping classification application by varying number of virtual servers and dataset size.

MB dataset the turnaround time passes from 11.8 hours to 49 minutes. With the 36 MB dataset, the turnaround time ranges from about 41 hours to 2.9 hours. As already noted for the clustering application, also in this case the total cost paid does not significantly vary with the number of virtual servers used.

Fig. 5.25 shows turnaround times and speedup values by varying number of virtual servers and dataset size. For the 9 MB dataset, the speedup passes from 1.8 using 2 servers to 14.4 using 16 servers. For the 18 MB dataset, the speedup ranges from 2.0 to 14.6. Finally, with the 36 MB dataset, the speedup ranged from 1.8 to 14.3.

Note that, differently from the clustering experiments discussed earlier, in this case the speedup does increase linearly with the number of servers used, since the 16 classification tasks are homogeneous in terms of execution times.

5.6.2 Workflows-based data mining applications

In the following we describe two workflow-based data mining applications designed with our system.

Association analysis application

The application is based on DMET-Analyzer [69], a tool for the automatic association analysis among the variation of the patient genomes and the clinical conditions of patients. The dataset used in our experiments contains information about the status of a set of probes (i.e., fragments of DNA or RNA of variable length) of 28 subjects: 14 subjects have a specific disease, the others are healthy. Each dataset row represents the status of a probe, and each column represents a subject. The DMET-Analyzer calculates for each probe a *p-value*, which measures the correlation between a status of a probe and the disease.

To evaluate our framework with increasing workloads, we generated three synthetic datasets with size of 12.5 MB, 25 MB and 50 MB. These datasets have a constant number of subjects (28) but varies the number of probes examined: around 10,000 probes for the dataset of 12.5 MB, 20,000 for 25 MB, and 40,000 for 50 MB dataset.

The application workflow, shown in Figure 5.26, performs the following steps. The initial dataset is partitioned in n parts. The number of partitions n is equal to the number of available Workers (in our case it has been instantiated 16 Workers). Each part *DatasetPart*[i] is analyzed by an instance of DMET-Analyzer ($DMETAnalyzer[i]$) and generates $PartialModel[i]$, which contains the p-values of each probe. The partial models $PartialModel[n]$ are corrected using two statistical correctors: $Corrector_0$ uses an *FDR* correction, while $Corrector_1$ uses a *Bonferroni* correction. The $ModelMerger$ tool merges partial model parts into a single model. We use it to create three models $ModelNC$, $ModelFDR$ and $ModelBONF$, which are respectively the model with no corrections (composition of $PartialModel[n]$), the model with FDR correction (composition of $PartialModelFDR[n]$) and, the model with Bonferroni correction (composition of $PartialModelBONF[n]$). Finally, the $ModelsMerger$ tool combines $ModelNC$, $ModelFDR$ and $ModelBONF$ in order to generate a single file combining the information of the three models from which it derives.

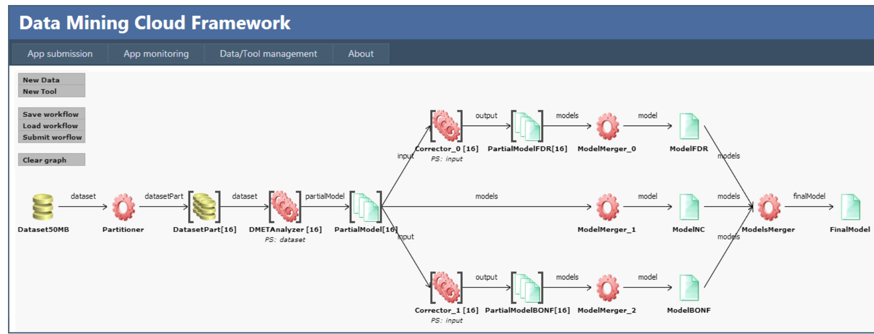


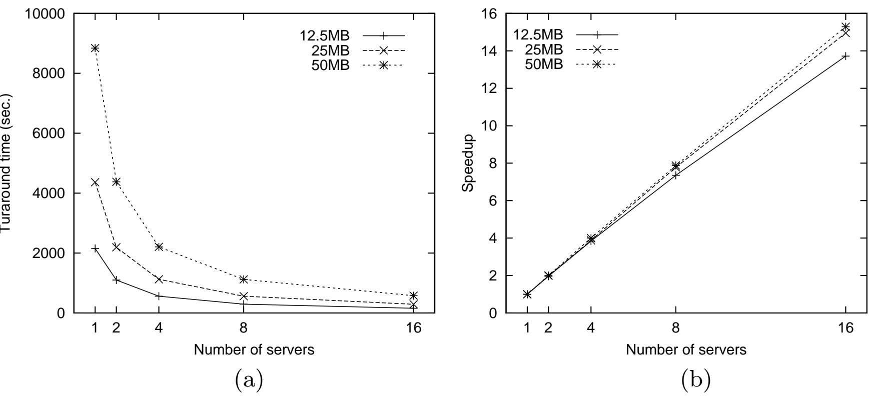
Fig. 5.26: DMET-Analyzer workflow-based application.

Table 5.7 shows the turnaround times of the application when 1, 2, 4, 8 and 16 virtual servers are used. For the 12.5 MB dataset the turnaround time decreases from around 35 minutes obtained with a single server, to about 2.6 minutes using 16 servers. For the 25 MB dataset the turnaround time passes from 1.2 hours to 5 minutes. With the 50 MB dataset, the turnaround time ranges from about 2.5 hours to around 10 minutes.

Fig. 5.27 shows turnaround times and speedup values by varying number of virtual servers and dataset size. For the 12.5 MB dataset, the speedup passes from 1.9 using 2 servers to 13.7 using 16 servers. For the 25 MB dataset, the speedup ranges from 2.0 to 14.9. Finally, with the 50 MB dataset, the speedup ranged from 2.0 to 15.2.

N. of servers	**12.5MB dataset** **Turnaround time**	**25MB dataset** **Turnaround time**	**50MB dataset** **Turnaround time**
1	0:35:54	1:12:45	2:27:19
2	0:18:21	0:36:34	1:12:57
4	0:09:21	0:18:43	0:36:46
8	0:04:53	0:09:21	0:18:41
16	0:02:37	0:04:52	0:09:38

Table 5.7: Turnaround times for the DMET-Analyzer workflow-based application.



(a) (b)

Fig. 5.27: Turnaround times and speedup values of the DMET-Analyzer workflow-based application.

Ensemble learning application

This application is based on the implementation of a multi-class cancer classifier based on the analysis of genes, as described in [70]. The input dataset is the Global Cancer Map (GCM)[4], which contains the gene expression profiles of 280 samples representing 14 common human cancer classes. For each sample is reported the status of 16,063 genes and the type of tumor (class label). The dataset is divided in training set containing 144 instances and test set with 46 instances.

The goal of this application is to demonstrate how our framework can be used to build a multi-class classifier (using an ensemble approach), and how such classifier can be efficiently used to classify a big unlabeled dataset. To this end, we designed two different workflows.

The first workflow is shown in Figure 5.28. The input training set, *GCM-train*, is analyzed by:

- 9 instances of the J48 [57] classification algorithm, obtained sweeping the *confidence value* and the *minNumObjects* (minimum number of instances per leaf) parameters;
- 9 instances of the JRip [71] classification algorithm, obtained sweeping the *numFolds* (number of folders) and *seed* parameters.

Each instance of J48, $J48[i]$, generates two outputs: a model ($Model_0[i]$) and an evaluation of such model ($EvalModel_0[i]$). The evaluation, obtained using a 10-fold cross-validation, measures different performance metrics such as number of misclassified items, precision, recall and F-measure. Each instance of JRip, $JRip[i]$, generates a model ($Model_1[i]$) and its evaluation ($EvalModel_1[i]$)

[4] http://tunedit.org/repo/BioInformatics_Seville/Global_Cancer_Map

Starting from the 18 models and the test set *GCM-testset*, 18 classifiers produce concurrently the same number of classified test sets ($ClassTestSet[18]$). In the final step, a voter generates $FinalClassTestSet$ by assigning a class to each data item, by choosing the class predicted by the majority of the models. Since that classifiers in the ensemble may have not the same accuracy, a weighted majority voting is performed [72]. For each model, we use as weight the number of correctly classified items, divided by the total number of items.

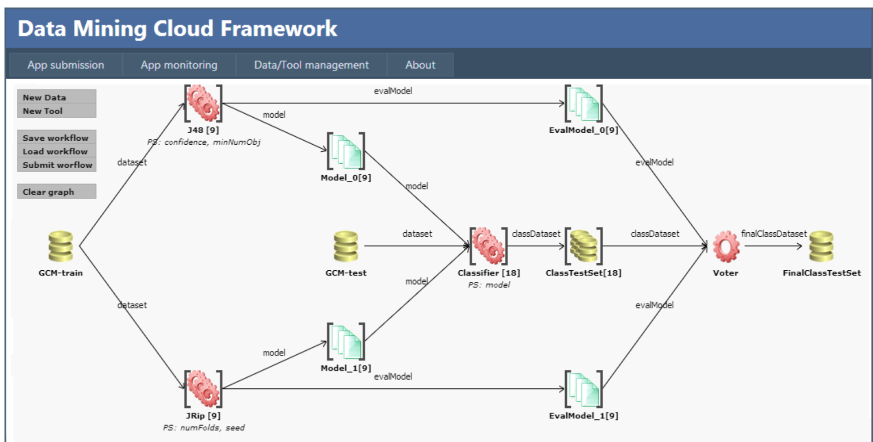


Fig. 5.28: Ensemble learning first workflow: models creation and combination

At this point, we can use our ensemble classifier to classify new datasets. This is shown in the second workflow (see Figure 5.29). We created an unlabeled dataset composed by 20,000 samples, $GCM2000[4]$, divided in four parts. Each part is analyzed using the set of models ($Model[18]$) and the corresponding evaluations ($EvalModel[18]$) generated by the previous workflow.

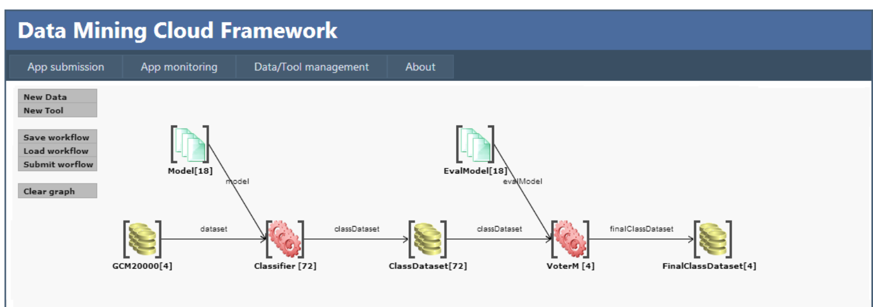


Fig. 5.29: Ensemble learning second workflow: Using of the classifier generated by first workflow to classify an unlabeled dataset.

Table 5.8 shows some statistics about the execution of the two workflow: the turnaround time, number of tasks, and the total execution time (i.e., the sum of the execution times of all the tasks) performed on 19 Workers.

Step	**Turnaround time**	**Number of tasks**	**Total exec. time**
First workflow	0:05:08	37	0:56:01
Second workflow	0:06:13	76	1:46:37

Table 5.8: Some statistics on the ensemble learning application.

5.7 Conclusions

We need new distributed infrastructures and smart scalable analysis techniques to solve more challenging problems in science. Cloud computing systems can be effectively used as scalable infrastructures for service-oriented knowledge discovery applications. Based on this vision, we designed the Data Mining Cloud Framework for large-scale data analysis on the Cloud. The workflow composition interface allows users to design and execute knowledge discovery applications defined as complex workflows.

We evaluated the performance of the system through the execution of parameter-sweeping and workflow-based knowledge discovery applications on a pool of virtual servers hosted by a Microsoft Cloud data center. The experimental results demonstrated the effectiveness of the framework, as well as the scalability that can be achieved through the execution of knowledge discovery applications on the Cloud.

Besides performance considerations, we point out that the main goal of the Data Mining Cloud Framework is providing an easy-to-use SaaS interface to reliable data mining algorithms, thus enabling end-users to focus on their data mining applications without worrying about low level computing and storage details, since they are transparently managed by the underlying Cloud infrastructure.

6

Conclusions

New parallel and distributed computing infrastructures are necessary to run smart scalable analysis techniques to solve more challenging problems in science, particularly when large data sets are involved or real-time evaluation is needed. Cloud computing systems are used today for implementing dynamic data centers and for high-performance computing on massive number of processors. They can also be effectively used as scalable infrastructures for running big data analysis that often involve large data sets and complex algorithms. The goal of this thesis was studying how the Cloud paradigm can be exploited to support scalable data processing and knowledge discovery applications in distributed scenarios.

As a first result of this study we presented the P2P-MapReduce system, an extension of current MapReduce implementations to make the programming model suitable for dynamic large-scale Cloud environments. The system exploits a peer-to-peer model to manage node churn, master failures, and job recovery in a decentralized but effective way, so as to provide a more reliable MapReduce middleware that can be effectively exploited in dynamic Cloud infrastructures. We provided a detailed description of the basic mechanisms that are at the base of the P2P-MapReduce system, discussed a prototype implementation based on the JXTA framework, and presented an extensive performance evaluation of the system in different network scenarios. The experimental results showed that, differently from centralized master-server implementations, the P2P-MapReduce framework does not suffer from job failures even in presence of very high churn rates, thus enabling the execution of reliable MapReduce applications in very dynamic Cloud infrastructures.

As a second result of our work, we described an extension of COMPSs, a framework that provides a programming model and a runtime system to ease the development of distributed applications and their execution on a wide range of computational infrastructures. The goal of the extension was enhancing the interoperability layer to support the execution of COMPSs applications into the Windows Azure Platform. The proposed approach has been validated through the execution of a data mining workflow ported to

COMPSs and executed on an hybrid testbed composed of machines from the Microsoft Azure Cloud and from a private infrastructure managed by Emotive Cloud. The results demonstrated that the COMPSs runtime is able to manage and schedule the tasks on different Cloud infrastructures in a transparent way, keeping the overall performance of the application.

Finally, we presented a Data Mining Cloud Framework designed to support the execution of data analysis applications on the Cloud. The framework allows users to design and execute three classes of knowledge discovery applications: single-task, parameter-sweeping, and workflow-based applications. A Web-based interface allows users to design and execute workflow-based applications with a visual drag-and-drop approach. We evaluated the performance of the system through the execution of a set of parameter-sweeping and workflow-based knowledge discovery applications on a pool of virtual servers hosted by a Microsoft Cloud data center. The experimental results demonstrated the effectiveness of our framework, as well as its scalability. Besides performance considerations, we point out that the Cloud approach implemented in our framework enables end-users to focus on their data mining applications at a high level of abstraction, thus freeing them from the need to cope with computing, storage and execution details, which are transparently managed by the system.

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 supervisors, Domenico Talia and Paolo Trunfio, who were very generous with their time and knowledge and helped me in each step to complete my PhD course.

I am also grateful to Rosa Maria Badia, who has given me the opportunity to work with her at the Barcelona Supercomputing Center. Many thanks also to other guys that I met in Barcelona.

And finally, but not least, thanks go to my family, colleagues, friends, and Laura for their indispensable support essential to reach this goal.

References

1. R. Barga, D. Gannon, D. Reed. 2011. “The Client and the Cloud: Democratizing Research Computing”. *IEEE Internet Computing*, 72-75, 2011.
2. A. Li, X. Yang, S. Kandula, M. Zhang. “CloudCmp: comparing public cloud providers”. 10th ACM SIGCOMM conference on Internet measurement (IMC ’10). New York, USA, 2010
3. SOAP, http://www.w3.org/TR/soap/
4. L. Richardson, S. Ruby. “RESTful Web Services”. O’Reilly & Associates, California, 2007.
5. J. J. Garrett. “Ajax: A New Approach to Web Applications”. Technical report, Adaptive Path, 2005.
6. HTML5, http://www.w3.org/TR/html5/
7. Open Cloud Computing Interface Working Group, http://www.occi-wg.org
8. Distributed Management Task Force Inc., Open Virtualization Format Specification v1.1, DMT Standar DSP0243,2010.
9. Microsoft Azure, http://www.microsoft.com/azure
10. Amazon Web Services, http://aws.amazon.com/
11. B. Sotomayor, R. S. Montero, I. M. Llorente, I. Foster. “Virtual Infrastructure Management in Private and Hybrid Clouds”, *IEEE Internet Computing*, vol. 13, 14-22, 2009.
12. D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. “The Eucalyptus Open-Source Cloud-Computing System”. In Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID ’09). Washington, USA,2009
13. G. Fox, D. Gannon. “Cloud Programming Paradigms for Technical Computing Applications” Technical report, Indiana University, 2012.
14. S. Altschul, W. Gish, W. Miller, E. Myers, D. Lipman. “Basic local alignment search tool”. *Journal of Molecular Biology*, 1990.
15. J. Dean, S. Ghemawat. “MapReduce: Simplified data processing on large clusters”. 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI’04), San Francisco, USA, 2004.
16. Hadoop. `http://hadoop.apache.org` (site visited December 2010).
17. S. Papadimitriou, J. Sun. “DisCo: Distributed Co-clustering with Map-Reduce: A Case Study towards Petabyte-Scale End-to-End Mining”. 8th IEEE International Conference on Data Mining (ICDM’08), Pisa, Italy, 2008.

18. J. Ekanayake, S. Pallickara, G. Fox. “Mapreduce for data intensive scientic analyses”. 4th IEEE International Conference on e-Science (e-Science’08), Indianapolis, USA, 2008.
19. D. Rao, D. Yarowsky. “Ranking and semi-supervised classification on large scale graphs using map-reduce”. Workshop on Graph-based Methods for Natural Language Processing (TextGraphs’09), Stroudsburg, USA, 2009.
20. E. Tejedor, R.M. Badia. “COMP Superscalar: Bringing GRID superscalar and GCM Together”. IEEE Int. Symposium on Cluster Computing and the Grid, Lyon, France, 2008.
21. Y. Gu, R. Grossman. “Sector and Sphere: The Design and Implementation of a High Performance Data Cloud”. *Philosophical Transactions, Series A: Mathematical, physical, and engineering sciences*, 367(1897), 2429-2445, 2009.
22. R. Grossman, Y. Gu, “Data Mining Using High Performance Data Clouds: Experimental Studies Using Sector and Sphere”. Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining, Las Vegas, USA, 2008.
23. C. Moretti , J. Bulosan , D. Thain , P. J. Flynn. “All-Pairs: An Abstraction for Data-Intensive Cloud Computing”. IEEE Int. Symposium on Parallel and Distributed Processing (IPDPS’08), Miami, USA, 2008.
24. M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly. “Dryad: distributed data-parallel programs from sequential building blocks”. 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys’07), Lisbon, Portugal, 2007.
25. J. Ekanayake, T. Gunarathne, G. Fox, A. S. Balkir, C. Poulain, N. Araujo, R. Barga. “DryadLINQ for Scientific Analyses”. 5th IEEE International Conference on e-Science (e-Science’09), Oxford, UK, 2009.
26. Y. Wei, K. Sukumar, C. Vecchiola, D. Karunamoorthy, R. Buyya. “Aneka Cloud Application Platform and Its Integration with Windows Azure”. CoRR, abs/1103.2590, 2011.
27. Amazon Elastic MapReduce. `http://aws.amazon.com/elasticmapreduce` (site visited December 2010).
28. Mapper API for Google App Engine. `http://googleappengine.blogspot.com/2010/07/introducing-mapper-api.html` (site visited December 2010).
29. J. Dean, S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. *Communications of the ACM*, 51(1), 107-113, 2008.
30. Google’s Map Reduce. `http://labs.google.com/papers/mapreduce.html` (site visited December 2010).
31. F. Marozzo, D. Talia, P. Trunfio. “Adapting MapReduce for Dynamic Environments Using a Peer-to-Peer Model”. 1st Workshop on Cloud Computing and its Applications (CCA’08), Chicago, USA, 2008.
32. F. Marozzo, D. Talia, P. Trunfio. “A Peer-to-Peer Framework for Supporting MapReduce Applications in Dynamic Cloud Environments”. In: N. Antonopoulos, L. Gillam (eds.), Cloud Computing: Principles, Systems and Applications, Springer, Chapter 7, 113-125, 2010.
33. A. Dou, V. Kalogeraki, D. Gunopulos, T. Mielikainen, V. H. Tuulos. “Misco: A MapReduce framework for mobile systems”. 3rd Int. Conference on Pervasive Technologies Related to Assistive Environments (PETRA’10), New York, USA, 2010.
34. Gridgain. `http://www.gridgain.com` (site visited December 2010).

35. Skynet. http://skynet.rubyforge.org (site visited December 2010).
36. MapSharp. http://mapsharp.codeplex.com (site visited December 2010).
37. J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, G. Fox. "Twister: A Runtime for Iterative MapReduce". 1st International Workshop on MapReduce and its Applications (MAPREDUCE'10), Chicago, USA, 2010.
38. Disco. http://discoproject.org (site visited December 2010).
39. M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, I. Stoica. "Improving MapReduce Performance in Heterogeneous Environments". 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08), San Diego, USA, 2008.
40. T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, R. Sears. "MapReduce Online". 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI'10), San Jose, USA, 2010.
41. C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis. "Evaluating MapReduce for multi-core and multiprocessor systems". 13th International Symposium on High-Performance Computer Architecture (HPCA'07), Phoenix, USA, 2007.
42. H. Lin, X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, Z. Zhang. "MOON: MapReduce On Opportunistic eNvironments". 19th International Symposium on High Performance Distributed Computing (HPDC'10), Chicago, USA, 2010.
43. B. Tang, M. Moca, S. Chevalier, H. He, G. Fedak. "Towards MapReduce for Desktop Grid Computing". 5th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC'10), Fukuoka, Japan, 2010.
44. G. Fedak, H. He, F. Cappello. "BitDew: A Data Management and Distribution Service with Multi-Protocol and Reliable File Transfer". *Journal of Network and Computer Applications*, 32(5), 961975, 2009.
45. H. Garcia-Molina. "Election in a Distributed Computing System". *IEEE Transactions on Computers*, 31(1), 48-59, 1982.
46. L. Gong. "JXTA: A Network Programming Environment". *IEEE Internet Computing*, 5(3), 88-95, 2001.
47. I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, H. Balakrishnan. "Chord: a scalable peer-to-peer lookup protocol for internet applications". IEEE/ACM Transactions on Networking, 11(1), 17-32, 2003.
48. K. Albrecht, R. Arnold, M. Gahwiler, R. Wattenhofer. "Join and Leave in Peer-to-Peer Systems: The Steady State Statistics Service Approach". Technical Report 411, ETH Zurich, 2003.
49. European Grid Infrastructure, http://www.egi.eu
50. StratusLab, http://www.stratuslab.eu
51. European Middleware Initiative, http://www.eu-emi.eu
52. Virtual multidisciplinary ENvironments USing Cloud infrastructures, http://www.venus-c.eu
53. D. Lezzi, R. Rafanell, A. Carrion, I. Blanquer, R.M. Badia, V. Hernandez. "Enabling e-Science applications on the Cloud with COMPSs". Cloud Computing: Project and Initiatives, 2011.
54. G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. T. Schütt, E. Seidel, B. Ullmer. "The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid". Proceedings of the IEEE, vol. 93, no. 3, Mar 2005.

55. F. Marozzo, D. Talia, P. Trunfio. “A Cloud Framework for Parameter Sweeping Data Mining Applications”. 3rd IEEE Int. Conference on Cloud Computing Technology and Science (CloudCom ’11), Athens, Greece, 2011.
56. H. Witten, E. Frank. *Data Mining: Practical machine learning tools with Java implementations*. Morgan Kaufmann Publishers, 2000.
57. J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.
58. I. Goiri, J. Guitart, J. Torres. “Elastic Management of Tasks in Virtualized Environments”. XX Jornadas de Paralelismo (JP 2009), Coruña, Spain, 2009.
59. GlusterFS Distributed Network File System, http://www.gluster.org
60. Hadoop on Azure, https://www.hadooponazure.com
61. Project Daytona, http://research.microsoft.com/en-us/projects/daytona
62. Google App Engine, http://code.google.com/intl/de/appengine
63. J. Ekanayake, Hui Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, G. Fox. “Twister: A Runtime for Iterative MapReduce”. 1st Int. Workshop on MapReduce and its Applications (MAPREDUCE’10), Chicago, USA, 2010.
64. Y. Simmhan, C. Ingen, G. Subramanian, J. Li. “Bridging the Gap between Desktop and the Cloud for eScience Applications”. 3rd IEEE Int. Conference on Cloud Computing (CLOUD ’10), Washington, USA, 2010.
65. J. B. MacQueen. “Some Methods for classification and Analysis of Multivariate Observations”. 5th Berkeley Symposium on Mathematical Statistics and Probability, Berkeley, UK, 1967.
66. E. Cesario, M. Lackovic, D. Talia, P. Trunfio, “Programming Knowledge Discovery Workflows in Service-Oriented Distributed Systems”. *Concurrency and Computation: Practice and Experience*, Wiley InterScience, 2012.
67. Z. H. Zhou.“*Semi-supervised learning by disagreement*”. 4th IEEE International Conference on Granular Computing, pp. 93, 2008.
68. S. Hettich, S. D. Bay. The UCI KDD Archive [http://kdd.ics.uci.edu]. Irvine, CA: University of California, Department of Information and Computer Science, 1999.
69. P. H. Guzzi, G. Agapito, M. T. Di Martino, M. Arbitrio, P. Tassone, P. Tagliaferri, M. Cannataro. “DMET-Analyzer: automatic analysis of Affymetrix DMET Data”. *BMC Bioinformatics*, 13, 2012.
70. S. Ramaswamy, P. Tamayo, R. Rifkin, S. Mukherjee, C.-H. Yeang, M. Angelo, C. Ladd, M. Reich, E. Latulippe, J.P. Mesirov, T. Poggio, W. Gerald, M. Loda, E.S. Lander and T.R. Golub. “Multiclass cancer diagnosis using tumor gene expression signatures”. *Proceedings of the National Academy of Sciences USA (PNAS)*. vol. 98(26), December, 2001.
71. W. W. Cohen. “Fast Effective Rule Induction”. Twelfth International Conference on Machine Learning, 115-123, 1995.
72. L. I. Kuncheva. “Combining Pattern Classifiers: Methods and Algorithms”. Wiley-Interscience. 2004.
73. D. Talia, P. Trunfio. How Distributed Data Mining Tasks can Thrive as Knowledge Services. *Communications of the ACM*, 53(7), 132-137, 2010.
74. V. Stankovski, M. T. Swain, V. Kravtsov, T. Niessen, D. Wegener, J. Kindermann, W. Dubitzky. Grid-enabling data mining applications with DataMiningGrid: An architectural perspective. *Future Generation Computer Systems*, 24(4), 259-279, 2008.

75. S. AlSairafi, F. S. Emmanouil, M. Ghanem, N. Giannadakis, Y. Guo, D. Kalaitzopoulos, M. Osmond, A. Rowe, J. Syed, P. Wendel. The Design of Discovery Net: Towards Open Grid Services for Knowledge Discovery. *Int. Journal of High Performance Computing Applications*, 17(3), 297-315, 2003.
76. P. Brezany, J. Hofer, A. M. Tjoa, A. Woehrer. "GridMiner: An Infrastructure for Data Mining on Computational Grids". Proc. APAC Conference and Exhibition on Advanced Computing, Grid Applications and eResearch (APAC'03), Gold Coast, Australia, 2003.
77. A. Congiusta, D. Talia, P. Trunfio. Distributed data mining services leveraging WSRF. *Future Generation Computer Systems*, 23(1), 34-41, 2007.
78. D. Talia, P. Trunfio, O. Verta. The Weka4WS framework for distributed data mining in service-oriented Grids. *Concurrency and Computation: Practice and Experience*, 20(16), 1933-1951, 2008.
79. S. Woodman, H. Hiden, P. Watson, J. Cala. "Workflows and Applications in e-Science Central". 5th IEEE International Conference on E-Science Workshops, UK, 2009.
80. W.i Lu, J. Jackson, R. Barga. 2010. "AzureBlast: a case study of developing science applications on the cloud". 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10). New York, USA, 2010.
81. G. Juve, E.Deelman, K. Vahi, G. Mehta, B. Berriman, B.P. Berman, P. Maechling. "Scientific workflow applications on Amazon EC2". 5th IEEE International Conference on E-Science Workshops, Oxford, UK, 2009.