ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/256912581

Real-World Distributed Supercomputing

Thesis · November 2010

CITATIONS
0

READS
132

1 author:

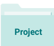
Niels Drost
Netherlands eScience Center

65 PUBLICATIONS 1,513 CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

The Ibis project View project



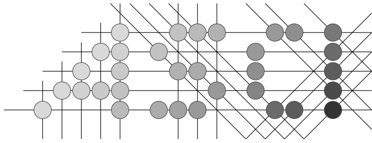
Via Appia View project

All content following this page was uploaded by Niels Drost on 20 May 2014.

The user has requested enhancement of the downloaded file.

Real-World Distributed Supercomputing

Niels Drost



Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number 219.

Nederlandse Organisatie voor Wetenschappelijk Onderzoek

This research was financially supported by the Netherlands Organisation for Scientific Research (NWO) in the framework of *Ibis: a Java-based grid programming environment*, project 612.060.214.

ISBN 978 90 8659 506 8

Cover image by Anthony Bannister / FotoNatura.com

Copyright © 2010 by Niels Drost

VRIJE UNIVERSITEIT

Real-World Distributed Supercomputing

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op donderdag 25 november 2010 om 11.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Niels Drost

geboren te Alkmaar

promotor:	prof.dr.ir. H.E. Bal
copromotoren:	dr. R.V. van Nieuwpoort
	dr. F.J. Seinstra

members of the thesis committee:

dr. D.H.J. Epema
dr. F. Huet
prof.dr.ir. C.T.A.M. de Laat
prof.dr. S. Portegies Zwart
prof.dr.ir. M.R. van Steen

Omit needless words

Contents

Contents vii

Acknowledgments ix

About the Cover xi

1 Introduction 1

2 Zorilla: Instant Cloud Computing 7
2.1 Requirements . . . 9
2.2 Zorilla . . . 11
2.2.1 Resource Discovery and Scheduling . . . 15
2.2.2 Deployment . . . 17
2.2.3 Job Management . . . 18
2.3 Related Work . . . 19
2.4 Conclusions . . . 20

3 ARRG: Real-World Gossiping 21
3.1 Real-World Problems & Related Work . . . 23
3.1.1 Node failures . . . 24
3.1.2 Network unreliability . . . 24
3.1.3 Non-atomic operations . . . 25
3.1.4 Limited connectivity . . . 26
3.2 Actualized Robust Random Gossiping . . . 26
3.3 The Fallback Cache . . . 29
3.4 Perceived Network Size . . . 31
3.5 Experiments . . . 32
3.5.1 The Fully Connected Scenario . . . 33
3.5.2 The X@Home Scenario . . . 33
3.5.3 The Real-World Distributed System (RWDS) Scenario . . . 36
3.5.4 Cyclon . . . 39
3.5.5 Pathological Situations . . . 40
3.5.6 Experiment Summary . . . 42
3.6 Conclusions . . . 42

4 Flood Scheduling: Simple Locality-Aware Co-allocation 45
4.1 Related Work . . . 47
4.2 Flood Scheduling . . . 48
4.2.1 Optimizations . . . 50
4.3 Experiments . . . 50
4.4 Conclusions . . . 52

5 JEL: Unified Resource Tracking 55
5.1 Requirements of Resource Tracking models . . . 58
5.1.1 Programming Model Requirements . . . 58
5.1.2 Environment Requirements . . . 59
5.2 The Join-Elect-Leave Model . . . 59
5.2.1 Joins and Leaves . . . 60
5.2.2 Elections . . . 61
5.2.3 Consistency models . . . 61
5.3 Applicability of JEL . . . 62
5.3.1 Master-Worker . . . 63
5.3.2 Divide-and-Conquer . . . 63
5.3.3 Message Passing (MPI-1) . . . 64
5.4 JEL Implementations . . . 65
5.4.1 Centralized JEL Implementation . . . 65
5.4.2 Distributed JEL Implementation . . . 68
5.5 Evaluation . . . 69
5.5.1 Low level benchmark: Join test . . . 70
5.5.2 Network bandwidth usage . . . 72
5.5.3 Low level benchmark in a dynamic environment . . . 73
5.5.4 Satin Gene Sequencing Application . . . 74
5.6 Related Work . . . 75
5.7 Conclusions . . . 76

6 Large Scale Experiments 79
6.1 Multimedia Content Analysis . . . 82
6.2 Divide-and-conquer GO . . . 84
6.3 Instant Cloud . . . 86
6.4 Competitions . . . 89

7 Summary and Conclusions 91

Bibliography 95

Samenvatting 103

Curriculum Vitae 107

Acknowledgments

Thanks!

Really, Thanks!

Although it has only my name on the cover, this thesis would not have been possible without a large number of people. It is impossible to thank them all here. Still, I can try :-)

Thanks Dick, Fabrice, Cees, Simon, and Maarten, for being in my promotion committee, and the useful comments on the draft version of this thesis. Thanks Colinda, for cleaning up my mess, and being supportive. Thanks Dineke, for fixing all the mistakes in the Dutch summary. Thanks to my family, for being the nicest people I know. Thanks Rob, for being my co-supervisor, and creating Ibis. Thanks Mom, for teaching me to take people as they are. Thanks Ana, for the trip to Romania. Thanks Nespresso, for keeping me awake. Thanks Fabrice, for the nice weather, and good ideas. Thanks Roelof, for all the work on IbisDeploy. It's great! Thanks Henri, for hiring me, teaching me to omit needless words, and being the nicest boss possibly imaginable. Thanks Martine, for the pubquiz, and the Lonely Planet. Thanks Jochem, for really trying to teach me Go. Thanks Nick, for finally getting the fish tank done (motivation right here!). Thanks Thilo, for sending me out on a trip abroad not once, but twice! Thanks Frank, for the opportunity to keep on working at the VU and on my thesis, and teaching me to get my line of reasoning straight. Thanks Daniël, for being your witness. Thanks Kees, for designing the Ibis logo. Thanks Jenny, for Tim. Thanks Jeroen, for many a Friday evening wasted. Thanks Liesbeth, for inviting Colinda to your birthday. Thanks Tobias, for still inviting me to Oktoberfest each and every year although I never seem to make it. Thanks Dad, for teaching me to keep working although no one is watching. Thanks Jacopo, for getting me to Australia. Thanks Marko, for always having dinner ready. Thanks Rutger, for making stuff really fast. Thanks Jason, for correcting all my *bendy sentences*, creating Ibis, and racing at Zandvoort! Thanks Arjen, for all the gadgets. Thanks Ben-Hai, for all the advice, and the loempias ;-) Thanks Internet, for giving me something to do when I was bored. Thanks Melanie, for walking in the sea. Thanks Tim, for growing up to be a nice person. Thanks Kees, for keeping the DAS-1/2/3 working like a charm, all the

time. Thanks Marieke, for the amusement park trips. Thanks Anthony Bannister, for taking a really nice picture of a Zorilla. Thanks Erik, for buying dinner each and every time you visit, and not crashing at 276 Km/h. Thanks Randy, for all the music. Thanks Gosia, for all the coffee (right back at ya!). Thanks Caroline, for the new staff room. Thanks Homme, for all the games. Thanks Maikel, for driving me home about a million times. Thanks Marjolein, for listening. Thanks Hieke, for the singing. Thanks Maik, for dragging me to STORM. Thanks Reinout, for doing all this work on open source projects, which I never seem to find the time for. Thanks Timo, for an emergency supply of capsules. Thanks Yvonne, for all the BBQs. Thanks Ian, Ian, and Ian at Cardiff, for the inspiration, insights, and beer. Thanks Ceriel, for fixing that bug keeping me from running that final experiment 2 hours before that very important deadline, about 10 times during the last 6 years. Thanks Govert, for fixing all the bicycle paths. You are my hero ;-) Thanks Wouter, for sharing a phone line, and lots of other stuff since. Thanks Dick, Elsbeth, and Jacqueline, for running in the rain. Thanks Herbert, for the Friday afternoon borrels. Thanks Klaas, for sharing a desk. Thanks Sander, for being a good friend from day one. Thanks Mathijs, for the pointers on how to get all the forms filled out for actually getting a PhD. Thanks everyone I forgot, for not being mad at me.

-Niels

About the Cover*

The **Striped Polecat** (*Ictonyx striatus*, also called the **African Polecat**, **Zoril**, **Zorille** or **Zorilla**) is a member of the Mustelidae family (weasels) which somewhat resembles a skunk. It is found in savannahs and open country in sub-saharan Africa excluding the Congo basin and west Africa.

Like other polecats, this carnivore is nocturnal. It has several means of avoiding predators — including the ability to emit foul-smelling secretions from its anal glands, playing dead and climbing trees. The animal is mainly black but has four prominent white stripes running from the head, along the back to the tail. The Striped Polecat is typically 60 centimeters long including a 20-centimeter tail. It lives for up to 13 years.

The Striped Polecat is solitary, tolerating contact with others only to mate. Young are generally born between September and December, with one to three young per litter.

* From Wikipedia, the free encyclopedia [84].

Chapter 1

Introduction

Ever since the invention of the computer, users have desired higher and higher performance. For an average user the solution was simply a matter of patience: each newer model computer has been faster than the previous generation for as long as computers have existed. However, for some users this was not enough, as they required more compute power than any normal machine could offer. Examples of *high performance computing* users are meteorologists performing weather predictions using complex climate models, astronomers running simulations of galaxies, and medical researchers analyzing DNA sequences.

To explain some of the major challenges encountered by high performance computing users, we use an analogy: making coffee. What if I was responsible for making coffee for a group of people, for instance a group of scientists on break during a conference? If the group is small enough, I could use my own coffee maker, analogous to using my own computer to do a computation. However, this will not work if the group is too large, as it would take too long, leading to a large queue of thirsty scientists. I could also brew the coffee in advance, but that would lead to stale and probably cold coffee.

The obvious solution to my problem is to get a bigger, faster, coffee maker. I could go out and buy an industrial-size coffee maker, like the one in a cafeteria, or even a coffee vending machine. Unfortunately, these are very expensive. In computing, large, fast, expensive computers are called *supercomputers*.

Fortunately, several alternatives exist that will save money. Instead of a single big coffee maker, I could use a number of smaller machines (a *cluster* in computing terms). I could also rent a coffee maker (*cloud computing*), or even borrow one (*grid computing*). In reality, I would probably use a combination of these alternatives, for instance by using my own coffee maker, borrowing a few, and renting a machine.

Although combining machines from different sources is the cheapest solution, it may cause problems. For one, different coffee machines need different types of coffee, such as beans, ground coffee, pads, or capsules. Moreover, these different machines all need to be operated in different ways, produce coffee at different speeds, and may even produce a different result (for instance, espresso). In the

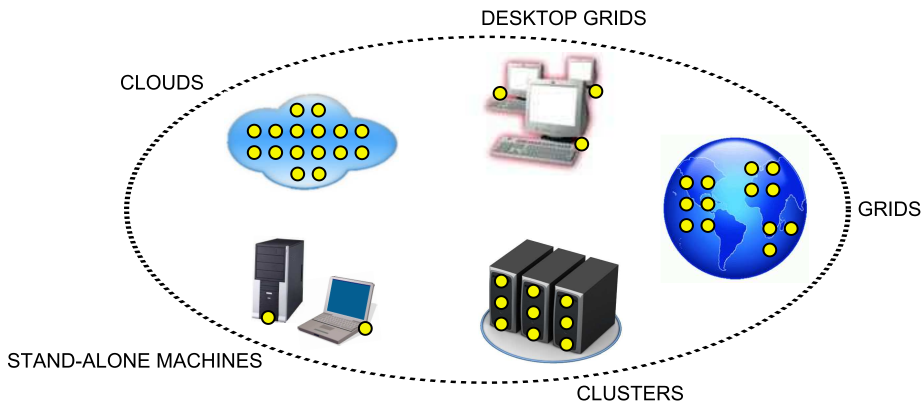


Figure 1.1: A 'worst-case' real-world distributed system.

end, I may be spending a considerable amount of time and effort orchestrating all these coffee makers.

When running high performance applications, users employ a strategy to acquire resources much like the one used in the example. Ideally, a single system (supercomputer, cluster, or otherwise) is used to run computations. However, a significant fraction of all users does not have access to a machine which is powerful enough *and* available when needed. Availability is a particularly large problem, as queue times on computing infrastructure may be considerable, and machines may be down for maintenance, or even switched off. As a result, users are forced to use a combination of resources, acquired from various sources.

In this thesis we use the term *real-world distributed system* (see Figure 1.1) for the collection of resources generally available to users. A real-world distributed system may consist of different systems, including clusters, grids, desktop grids, clouds, as well as stand-alone machines. By their nature, real-world distributed systems are very heterogeneous, containing resources with different paradigms, available software, access policies, etc. Also, these systems are very dynamic: new resources can be added to or removed from a system at any time, and as a real-world distributed system has many independent parts, the chance of a failure occurring at a given time is high.

Because of these problems, usage of real-world distributed systems for high performance computing is currently rather limited [12]. In general, users install and run their software manually on a small number of sites. Moreover, parallel applications are often limited to coarse-grained parameter-sweep or master-worker programs. More advanced use cases such as automatically discovering resources, or running applications across multiple sites, are currently impractical, if not impossible. This is unfortunate, as many scientific and industrial applications can benefit from the use of distributed resources (e.g., astronomy [41], multimedia [70], and medical imaging [56]).

The ultimate goal of the Ibis project [44], in which the research described in this thesis is embedded, is *to provide users of high performance computing with transparent, easy use of resources, even on very heterogeneous distributed systems.* This is not unlike the original goal of grid computing: "efficient and transparent (i.e., easy-to-use) wall-socket computing over a distributed set of resources" [35], sometimes referred to as the *promise of the grid*. The distributed computing paradigms introduced since then, including desktop grids, volunteer computing, and more recently cloud computing, all share many of the goals of grid computing, ultimately trying to give end-users access to resources with as little effort as possible.

The central research question of this thesis is *how to run distributed supercomputing applications on very heterogeneous, dynamic, systems.* Problems faced include how to *find* resources in such systems, how to *acquire* these resources once found, and how to *start* an application on available resources. Also, it is vital to *track* exactly which resources are available in a computation once the application is running. In all cases, we explicitly take into account *distributed supercomputing* applications, in which resources from multiple sites cooperate in a single high-performance distributed computation.

In this thesis, we focus mainly on the system aspects of running distributed supercomputing applications, i.e., how to get an application to run on a large-scale distributed system. Problems related to how to create such distributed supercomputing applications in the first place, and how the distributed application should communicate once running, are out of scope of this work. Examples of these problems are performance optimizations ,and how to effectively debug large scale applications.

In the Ibis project, other research is done that complements that described in this thesis, and that addresses problems not covered by this work. Topics include research on how to *communicate reliably* [54] and *efficiently* [60], and how to create programming models [10, 53, 61] that allow users to make efficient use of all resources available in these large systems.

Research overview

Recently, cloud computing has emerged as a high-performance compute platform, offering applications a homogeneous environment by using virtualization mechanisms to hide most differences in the underlying hardware. Unfortunately, not all resources available to a user offer cloud services. Also, combining resources of multiple cloud systems is far from trivial. To use all resources available to a user, software is needed which easily combines as many resources as possible into one coherent computing platform.

To address these problems we introduce Zorilla: a peer-to-peer (P2P) middleware that creates an *instant cloud* from any available set of compute resources. Zorilla imposes minimal requirements on the resources used, is platform independent, and does not rely on central components. In addition to providing functionality on *bare* resources, Zorilla can exploit locally available middleware. Using Zorilla as

an example, we show that the combination of virtualization and P2P techniques greatly simplifies the design and implementation of many aspects of middleware, including resource discovery, deployment, management, and security. Zorilla acts as a software platform for much of the research presented in this thesis.

Gossiping is an effective way of disseminating information in large dynamic systems, and is used in Zorilla for resource discovery. Until now, most gossiping algorithms have been designed and evaluated using simulations. However, these algorithms often cannot cope with several real-world problems that tend to be overlooked in simulations, such as node failures, message loss, non-atomicity of information exchange, and firewalls. We introduce *Actualized Robust Random Gossiping* (ARRG), an algorithm specifically designed to take all of these real-world problems into account *simultaneously*.

In Zorilla, P2P techniques are used effectively to run applications on large systems. However, the lack of central components make scheduling on P2P systems inherently difficult. Especially distributed supercomputing applications are problematic, as these require the simultaneous allocation of multiple, possibly distributed, resources (so-called *co-allocation*). As a possible solution, we introduce *flood scheduling*. Flood scheduling supports co-allocation, is locality aware, decentralized, and flexible. We show that flood scheduling is a good alternative to centralized algorithms.

Zorilla allows users to run applications on dynamic systems. Unfortunately, this may cause the set of resources used to change during a computation as machines crash, reservations end, and new resources become available. It is vital for applications to respond to these changes, and therefore necessary to keep track of the available resources, a notoriously difficult problem. We propose a new functionality to be added to any system designed for dynamic environments: *resource tracking*. We introduce a general solution to resource tracking: the Join-Elect-Leave (JEL) model. JEL provides *unified* resource tracking for parallel and distributed applications across environments. JEL is a simple yet powerful model based on notifying when resources have *Joined* or *Left* the computation. We demonstrate that JEL is suitable for resource tracking in a wide variety of programming models, and compare several JEL implementations.

Zorilla, and the techniques it incorporates described in this thesis, greatly enhance the applicability of real-world distributed systems for everyday users. Instead of limiting usage of these systems to a single site at a time, it is now possible to routinely use large numbers of resources, possibly distributed across the globe. Moreover, the work described in this thesis, combined with the complementary research done in the Ibis project, allows users to do this transparently, and with little effort. Instead of constantly managing files, jobs, and resources, users can now focus on the actual computations performed with their application.

The software* developed for this thesis, and other software part of the Ibis project, has been put to the test in international competitions [5]. In several of these competitions we have won awards, i.e., at the International Scalable Comput-

*Zorilla and other software referred to in this thesis can be freely downloaded from `http://www.cs.vu.nl/ibis`

ing Challenge at CCGrid 2008 (Lyon, France), at the International Data Analysis Challenge for Finding Supernovae at IEEE Cluster/Grid 2008 (Tsukuba, Japan), and at the Billion Triples Challenge at the 2008 International Semantic Web Conference (Karlsruhe, Germany). For more details, see Chapter 6. These awards clearly show the real-world applicability of our research.

Thesis outline

This thesis is structured as follows. In Chapter 2 we discuss the design and implementation of Zorilla, our P2P middleware. In Chapter 3, we introduce our ARRG gossip algorithm, and explore the problems in designing and applying gossiping algorithms in real systems. Chapter 4 addresses the problem of scheduling on P2P systems, and introduces *flood-scheduling* as a possible solution. In Chapter 5, we discuss resource tracking and introduce our JEL model. In Chapter 6, we perform world-wide experiments using up to 8 sites with over 400 cores. These experiments show Zorilla, and other techniques and algorithms described in this thesis, in real-world scenarios. Finally, in Chapter 7 we summarize the results of this research, and provide conclusions.

Chapter 2

Zorilla: Instant Cloud Computing*

The emergence of real-world distributed systems (see Figure 1.1) has made the running of applications complex for end-users. The heterogeneity of these systems makes it hard to install and run software on multiple resources, as each site requires configuring, compiling and possibly even porting the application to the specific resource. Current systems also lack so-called *global functionality*, such as system-wide schedulers and global file systems. Standardized hardware and software, as well as global functionality, requires coordination between all resources involved. In grids, coordination is done in a Virtual Organization (VO) specifically created for each grid. Since a real-world distributed system is created ad hoc, no such VO can exist. The resulting heterogeneity and lack of global functionality greatly hinders usability.

Recently, cloud computing has emerged as a promising new computing platform. Although originally designed to run mostly web servers, cloud computing, is now also used as a high performance computing platform [43, 65]. Although clouds are capable of offering high-level services, high performance computing is mostly concerned with low-level computing infrastructure, so-called *Infrastructure as a Service (IaaS)*.

One of the defining properties of cloud computing is its use of virtualization techniques. The use of a Virtual Machine such as Xen, VirtualBox, or the Java Virtual Machine (JVM) allows applications to run on any available system. Software is created and compiled once for a certain virtual environment, and this environment is simply deployed along with the software. Virtualization of resources is an effective way of solving the problem of heterogeneity in distributed systems today.

Although cloud computing allows a user to run applications on any available cloud resource, this is only part of the required solution. Users also have access to clusters, grids, desktop grids, and other platforms, which do not offer virtualiza-

*This chapter is based on our paper submitted for publication [28].

tion. Moreover, ad hoc created collections of resources still lack global functionality such as a global filesystem. What is needed is a platform capable of turning *any* (possibly distributed) collection of resources into a single, homogeneous, and easy to use system: an *instant cloud*.

Although an instant cloud system must support any number of resources, typically, everyday scenarios will most likely be somewhat simpler. As an example use case of an instant cloud, a scientist may have access to a local cluster. When this cluster is busy, or simply not powerful enough, he or she can combine the processing power of this cluster with acquired cloud computing resources, for instance Amazon EC2 [29] resources. Alternatively, the scientist can acquire additional resources by deploying the instant cloud on a number of desktop machines. Unfortunately, it is impossible to predict exactly which types of resources are combined by users. As a result, instant cloud middleware must support all combinations of resources. In this chapter, we will assume the worst-case scenario of *all* possible types of resources, ensuring the resulting system is applicable in all possible scenarios.

This chapter investigates middleware specially designed to create instant clouds on real-world distributed systems. This middleware has unique design requirements. For instance, since it needs to function on ad hoc created systems, it too must support ad hoc installation. This is different from existing grid and cloud middleware, where a more or less stable system is assumed. Also, instant clouds are created by users, not system administrators. As a result, instant cloud middleware must be very easy to install, and not require special privileges. Moreover, since resources usually utilize some form of middleware already, our new middleware must be able to cooperate with this local middleware.

The use of virtualization allows for a simplified design of our instant cloud middleware. For example, finding resources commonly requires complex resource discovery mechanisms. Virtualization allows applications to run on a much bigger fraction of all resources, allowing a greatly simplified resource discovery mechanism. In addition to virtualization, we also use peer-to-peer (P2P) techniques in implementing instant cloud middleware. P2P techniques allow for easy installation and maintenance-free systems, and are highly suited for large-scale and dynamic environments. Together, virtualization and P2P techniques combine into a relatively simple design for our cloud middleware.

In this chapter we introduce Zorilla: our P2P instant cloud middleware. Zorilla is designed to run applications remotely on systems ranging from clusters and desktop grids, to grids and clouds. Zorilla is fully P2P, with no central components to hinder scalability or fault-tolerance. Zorilla is implemented entirely in Java, making it highly portable. It requires little configuration, resulting in a system which is trivial to install on any machine with a Java Virtual Machine (JVM). Zorilla can be either installed permanently on top of a bare-bone system, or deployed on-the-fly exploiting existing local middleware.

As Zorilla can be installed on virtually any system, the size of a Zorilla system can vary greatly, from dozens of large clusters, for very few nodes with a lot of resources, to an entire University campus full of student PCs running Zorilla, for

thousands of nodes with few resources. In contrast, it could consist of 4 desktop machines in a single room too. How to get Zorilla itself running on these machines is outside the scope of this thesis, though one option is to use IbisDeploy (see Chapter 6).

Zorilla is a prototype system, explicitly designed for running *distributed supercomputing applications* concurrently on a distributed set of resources. It automatically acquires resources, copies input files to the resources, runs the application remotely, and copies output files back to the users' local machine.

Being a prototype system, our current implementation of Zorilla focuses on this single use-case, does not include all functionality present in a typical middleware. Most notable are its limited security mechanisms, and its lack of long term file storage functionality. Other groups are researching distributed filesystems and security in a P2P context [58, 75], and we consider integrating such systems as future work.

The contributions of this chapter are as follows:

- We establish the requirements of instant cloud middleware.
- We describe the design and implementation of Zorilla: a new lightweight, easy to use Java-based P2P instant cloud middleware.
- We show how the combination of virtualization and P2P helps in simplifying the design and enhancing the functionality of middleware. We especially explore resource discovery, deployment, management, and security.
- We show how the use of instant cloud middleware designed for real-world distributed systems brings closer the goal of easy-to-use distributed computing on these systems.

In this chapter we provide an overview of Zorilla. Chapter 3 and Chapter 4 provide more details on several techniques used in Zorilla. In Chapter 5 we discuss resource tracking, a vital part of application running in dynamic systems such as instant clouds, and explicitly supported in Zorilla. Chapter 6 shows several worldwide experiments done with Zorilla.

The rest of this chapter is organized as follows. In Section 2.1 we define the requirements of instant cloud middleware. Section 2.2 gives a description of Zorilla, our P2P instant cloud middleware. We discuss related work in Section 2.3. Finally, we conclude in Section 2.4.

2.1 Requirements

In this section we discuss the requirements of instant cloud middleware.

Resource independence: The primary function of instant cloud middleware is to turn *any* collection of resources into one coherent platform. The need for *resource independence*, the ability to run on as many different resources as possible, is paramount.

Middleware independence: As most resources already have some sort of middleware installed, instant cloud middleware must be able to interface with this *local middleware*[†]. The implementation of instant cloud middleware must be such that it is as portable as possible, functioning on different types of local middleware. This and the requirement of resource independence can be summed up into one requirement as well: *platform independence*.

Decentralization: Traditional (grid) middleware uses central servers to implement functionality spanning multiple resources such as schedulers and distributed file systems. Centralized solutions introduce a single point of failure, and are a potential performance bottleneck. In clusters and grids this is taken into account by hosting these services on high capacity, partially redundant machines. However, in an instant cloud, it is hard to guarantee such machines are available: resources are not under the control of the user, and reliability is hard to determine without detailed knowledge of resources. Therefore, middleware should rely on as little central functionality as possible. Ideally, instant cloud middleware uses no centralized components, and instead is implemented in a completely decentralized manner.

Malleability: In an instant cloud, the set of available resources may change, for instance if a resource is removed from the system by its owner. Middleware systems should support *malleability*, correctly handling new resources joining and leaving.

System-level fault-tolerance: Because of the many independent parts of a real-world distributed system, the chance that some resource fails at a given time is high. Middleware systems should be able to handle these failures gracefully. Failures should not hinder the functioning of the entire system, and failing resources should be detected, and if needed replaced. Note that this does not include *application-level fault-tolerance*: restoring the state of any application running on the failing resource. Application-level fault-tolerance is usually implemented either in the application programming model, or the application itself. Support for application-level fault-tolerance in the middleware can be limited to failure detection and reporting.

Easy deployment: Since an instant cloud is created ad hoc by end-users, middleware is typically deployed by the user, possibly for the duration of only a single experiment. Therefore, instant cloud middleware needs to be as easy to deploy as possible. Complicated installation and setup procedures defeat the purpose of clouds. Also, no additional help from third parties, such as system administrators, should be required to deploy the middleware.

Parallel application support: Many high-performance applications can benefit from using multiple resources in parallel, even on distributed systems [5].

[†] We will use the term *local middleware* for existing middleware installed on resources, throughout this chapter.

Parallel applications require scheduling of multiple (distributed) resources concurrently, tracking which resources are available (see Chapter 5), and providing reliable communication in the face of firewalls and other problems [54].

Global file storage: Besides running applications, clouds are also used for storing files. In the simplest case files are used as input and output of applications. However, long term storage of data, independently of applications, is also useful. Ideally, instant cloud middleware should provide a single filesystem spanning the entire system. This filesystem must be resilient against failures and changes in available storage resources.

Security: As in all distributed systems, instant clouds must provide security. Middleware must protect resources from users, as well as users from each other. Because of the heterogeneous nature of the resources in an instant cloud, and the lack of a central authority, creating a secure environment for users and applications is more challenging than in most systems.

The large number of requirements for instant cloud middleware presented above lead us to the conclusion that using existing techniques for implementing instant cloud middleware is not possible. Some of the fundamental assumptions of traditional (grid) middleware (e.g. the presence of a reliable, centralized server), do not hold in an instant cloud. Therefore, our instant cloud middleware, discussed in the next section, uses a number of alternative approaches for implementing middleware functionality.

2.2 Zorilla

In this section we describe the design of Zorilla, our *prototype* instant cloud middleware. We will first give an overview of Zorilla, followed by a more detailed discussion of selected functionality. The main purpose of Zorilla is to facilitate running applications (jobs) remotely on any resource in the instant cloud.

Zorilla relies heavily on P2P techniques to implement functionality. P2P techniques have proved very successful in recent years in providing services on a large scale, especially for file sharing applications. P2P systems are highly robust against failures, as they have no central components which could fail, but instead implement all functionality in a fully distributed manner. In general, P2P systems are also easier to deploy than centralized systems, as no centralized list of resources needs to be kept or updated. One downside of P2P systems is a lack of trust. For instance, a reliable authentication system is hard to implement without any central components. We argue that P2P techniques can greatly simplify the design of middleware, if the limitations of P2P techniques are dealt with. Implementing all functionality of middleware using P2P techniques is the ultimate goal of our research.

A Zorilla system is made up of nodes running on all resources, connected by a P2P network (see Figure 2.1). Each node in the system is completely independent,

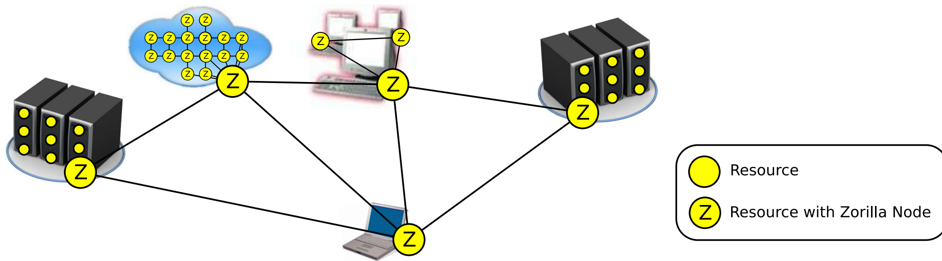


Figure 2.1: Example of an instant cloud created by Zorilla. This instant cloud consists of two clusters, a desktop grid, a laptop, as well as cloud resources (for instance acquired via Amazon EC2). On the clusters, a Zorilla node is run on the headnode, and Zorilla interacts with the local resources via the local scheduler. On the desktop grid and the cloud a Zorilla node is running on each resource, since no local middleware capable of scheduling jobs is present on these systems. All Zorilla nodes are connected by a P2P overlay network.

and implements all functionality required of a middleware, including handling submission of jobs, running jobs, storing of files, etc. Each Zorilla node has a number of local resources. This may simply be the machine it is running on, consisting of one or more processor cores, memory, and data storage. Alternatively, a node may provide access to other resources, for instance to all machines in a cluster. Using the P2P network, all Zorilla nodes tie together into one big distributed system. Collectively, nodes implement the required global functionality such as resource discovery, scheduling, and distributed data storage, all using P2P techniques.

Jobs in Zorilla consist of an application and input files, run remotely on one or more resources. See Figure 2.2 for an overview of the life cycle of a (parallel) job in a Zorilla system.

Zorilla has been explicitly designed to fulfill the requirements of an instant cloud middleware. Table 2.1 presents an overview of the requirements, and how Zorilla adheres to these. As in most cloud computing platforms, virtualization is used extensively in Zorilla. Zorilla is implemented completely in Java, making it *resource independent*: it is usable on any system with a suitable Java Virtual Machine (JVM). Virtualization is also used when applications are started. Instead of exposing the application to the underlying system, we hide this by way of a *virtual machine* (VM), currently either the JVM or Sun VirtualBox [79]. Although using virtual machines causes a decrease in performance, we argue that this is more than offset by the increase in usability and flexibility of the resulting system.

Another virtualization technique used in Zorilla is the use of a middleware independent API to access resources. As said, resources in a real-world distributed system commonly have existing local middleware installed. Zorilla will have to interact with this local middleware to make use of these resources. We use a generic API to interface to resources, in this case the JavaGAT [59]. The JavaGAT allows

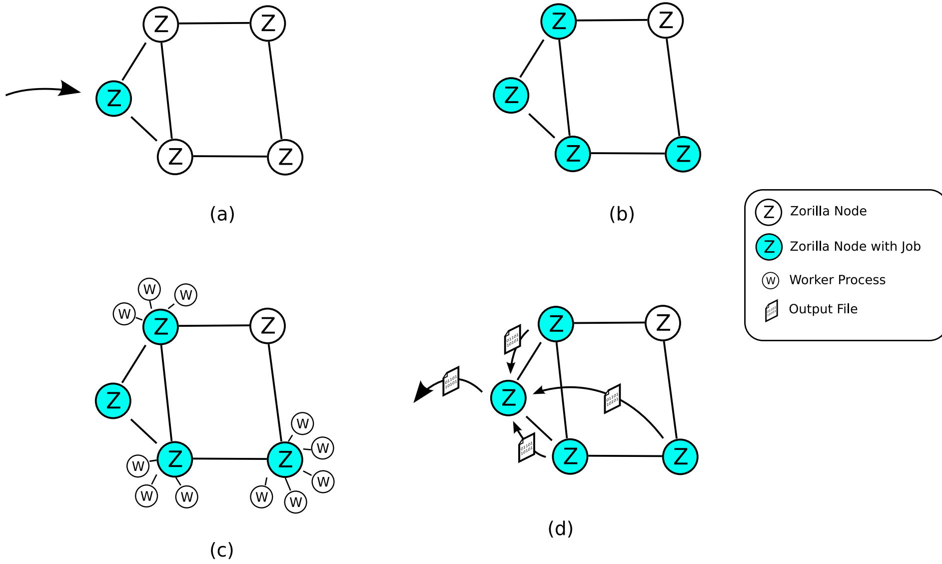


Figure 2.2: Job life cycle in a Zorilla system consisting of 5 nodes connected through an overlay network. (a) A (parallel) job is submitted by the user to a node. (b) The job is disseminated to other nodes. (c) Local schedulers at each node decide to participate in the job, and start one or more Worker processes (e.g. one per processor core available). (d) Output files are copied back to the originating node.

Requirement	Approach		Solution in Zorilla
	P2P	Virt.	
Resource independence		X	JVM
Middleware independence		X	JavaGAT [59]
Decentralization	X		P2P implementations of functionality
Malleability	X		Replacement resources allocated
System-level fault-tolerance	X		Faulty resources detected and replaced
Easy deployment	X	X	No server, sole requirement a JVM
Parallel application support	X	X	Flood scheduler (see Chapter 4) SmartSockets [54], JEL (see Chapter 5)
Global file storage	X		Per-job files only
Security		X	Sandboxing of applications

Table 2.1: Design overview of Zorilla. Listed are the requirements of an instant cloud, the approach used to address the issue (be it virtualization or peer-to-peer), and how this affects the design of Zorilla.

Zorilla to interact with a large number of middlewares, including Globus, Unicore, Glite, SGE, and PBS. Support for new middleware is added to JavaGAT regularly, and automatically available in Zorilla. The JavaGAT has a very stable API, and is currently being standardized by OGF as the SAGA API [39]. Zorilla uses the JavaGAT API whenever it uses resources, hiding the native API of the middleware installed on each specific resource. In effect, this makes Zorilla *middleware independent*.

The P2P design of Zorilla allows it to fulfill several of the established requirements of Table 2.1. All functionality is implemented without central components. Fault-tolerance and malleability is implemented in the resource discovery, scheduling, and job management subsystems of Zorilla. Any node failing has a very limited impact on the entire system, only influencing computational jobs the node is directly involved in. Likewise, removing a Zorilla node from the system is done by simply stopping the node. Other nodes will automatically notice that it is gone, and the remaining nodes will keep functioning normally.

Besides being useful techniques in themselves, the combination of virtualization and P2P provides additional benefits. Zorilla is very *easy to deploy*, partially because no central servers need to be setup or maintained. When a Zorilla node is started on a resource, it can be added to an existing Zorilla system by simply giving it the address of any existing node in the system. Also, as Zorilla is implemented completely in Java, it can be used on any resource for which a JVM is available, with no additional requirements. Another benefit of using both P2P and virtualization is that it allows Zorilla to *support parallel applications*. Zorilla explicitly allows distributed supercomputing applications by supporting applications which span multiple resources, and optimizing scheduling of these resources (see Section 2.2.1 and Chapter 4). Besides scheduling, parallel applications are also supported by offering reliable communication by way of SmartSockets (see Section 2.2.2), and resource tracking in the form of our JEL model (see Section 2.2.3 and Chapter 5).

Zorilla supports files when running applications as executables, virtual machine images, input files, and output files. Files are automatically staged to and from any resources used in the computation. To keep the design of Zorilla as simple as possible, files are always associated with jobs. This allows Zorilla to transfer files efficiently when running jobs, and makes cleanup of files trivial. However, this also limits the usage of files in Zorilla, as long-term file storage is not supported. We regard adding such a filesystem as future work.

The last requirement of Zorilla is security. The virtualization used by Zorilla allows us to minimize the access of applications to resources to the bare minimum, greatly reducing the risk of applications damaging a resource. However, Zorilla currently has little to no access restrictions. Since it is hard to implement a reliable authentication system using only P2P techniques, one alternative is to integrate support for the Grid Security Infrastructure (GSI) [34] also used in Globus into Zorilla.

2.2.1 Resource Discovery and Scheduling

We will now discuss several subsystems of Zorilla, starting with resource discovery. Whenever a job is submitted by a user, the first step in executing this job is allocating resources to it. In a traditional (grid) middleware system this is usually done by a centralized scheduler. In a P2P system, this approach obviously cannot be implemented. Instead, a distributed discovery and scheduling system is required.

Resource discovery in a P2P context is, in essence, a search problem. An important aspect of the resource discovery process is how exactly the required resources are specified, as this influences the optimal search algorithm considerably. One option for the specification of resources is to precisely specify the requirements of an application, including machine architecture, operating system (version), required software, libraries, minimum memory, etc. Unfortunately, finding a match for the above resource specification is difficult. As real-world distributed systems are very heterogeneous, a resource is likely to match only a small subset of the requirements. The chance of finding a resource fulfilling all of the requirements is akin to finding the proverbial needle in a haystack.

Instead of trying to search for resources matching all requirements of an application, we exploit the fact that virtualization is used when running applications. Using virtualization, any application can be deployed on any suitable hardware, independent of the software running on that hardware. The virtualization of resources greatly reduces the number of requirements of an application. What remains are mostly basic hardware requirements such as amount of memory, processor speed, and available disk space, in addition to a suitable virtual machine (Java, VirtualBox, VMware, Xen, or otherwise).

Most remaining requirements have a very limited range of values. For instance, any machine used for high performance computing is currently likely to have a minimum of 1GB of main memory. However, machines with over 16GB of main memory are rare. Other requirements such as processor speed, and hard disk size have a very limited range as well. Also, the number of virtual machines, and different versions of these virtual machines available, is not very large. Finally, most requirements can be expressed as *minimum* requirements, satisfied by a wide range of resources. From our analysis we conclude that the chances that a randomly selected machine matches the requirements of a randomly selected application are quite high when virtualization is used.

In Zorilla, the resource discovery process is designed explicitly for supporting virtualized resources. Because of virtualization, it is sufficient for our system to be capable of finding *commonly available* resources. Support for uncommon resources (the needle in a haystack), is not required. Instead, our system is *optimized for finding hay.*

As said, Zorilla explicitly supports parallel applications. This influences its design in a number of aspects, including resource discovery. As parallel applications require multiple resources, the middleware must support acquiring these. Besides the resources themselves, the connectivity between the acquired resources

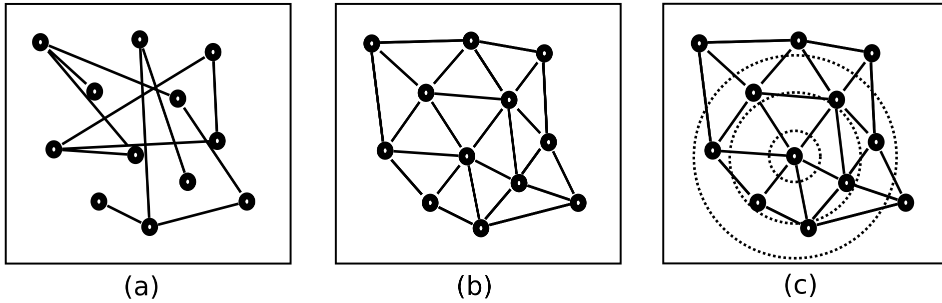


Figure 2.3: Resource discovery in Zorilla. (a) Random overlay created for resource discovery. (b) Neighbor connections created. (c) Flood scheduling (iterative ring search) performed using neighbors to schedule resources.

is also important. A parallel application may send and receive a large amount of data during its lifetime, so high bandwidth connections between the resources are required. Also, most parallel applications are sensitive to the latency between resources used. Zorilla supports parallel applications by allowing a user to request multiple resources, and by striving to allocate resources close (in terms of network latency) to the user. This gives applied resources a higher chance of having low latency, high bandwidth connections between them.

Resource discovery in Zorilla is a three step process (see Figure 2.3). First, a P2P overlay network consisting of all Zorilla nodes is built up. Second, the P2P overlay is then used to build up a list of close-by nodes or *neighbors*. Last, virtualized resources are searched using this neighbor list, using an *iterative flooding* algorithm. We will now briefly discuss each step in turn. For a more detailed description of the resource discovery and scheduling system of Zorilla, see Chapter 4).

Zorilla's overlay network is based on the ARRG *gossiping* algorithm (see Chapter 3). ARRG provides a *peer sampling service* [46] which can be used to retrieve information about peer nodes in the P2P overlay. Gossiping algorithms work on the principle of periodic information exchange between nodes. In ARRG, information on the nodes of the P2P network itself is kept in a limited size cache. On every gossip, entries in this cache are exchanged with peer nodes. These exchanges lead to a random subset of all nodes in the cache of each node. Taking entries from this cache thus yields a random stream of nodes in the P2P overlay (see Figure 2.3(a)).

Next, Zorilla uses the stream of random nodes to create a list of neighbors: nodes close-by in the network. For this purpose, Zorilla implements the Vivaldi [18] synthetic coordinate system. Vivaldi assigns coordinates in a Cartesian space to each node of a P2P overlay. Coordinates are assigned as to reflect the round trip latency between nodes. Given two Vivaldi coordinates, the distance between these two nodes can be calculated without any direct measurements. Vivaldi updates the coordinates of each node by periodically measuring the distance to a randomly selected node. Zorilla determines the distance to a node by comparing their *virtual*

coordinates with the coordinates of the local node. Zorilla continuously checks the random stream of nodes for potential neighbors, replacing far away neighbors with new close-by nodes (see Figure 2.3(b)).

Once a suitable P2P network and neighbor list is built up, this is then used for the actual allocation of resources to jobs. When a job is submitted at a node, Zorilla's *flood scheduling* algorithm (see Chapter 4) sends a request for resources to all neighbors of the node. Besides sending a reply if they have resources available, these neighbors in turn forward the message to all their neighbors. The search is bound by a maximum hop count, or *time to live (TTL)* for each request. If not enough resources are found, the search is repeated periodically with an increasingly larger TTL, causing more and more resources to be searched, further and further away (see Figure 2.3(c)). In effect, close-by resources (if available) are used before far away resources.

The resource discovery mechanism of Zorilla relies on the fact that resources are virtualized. Flooding a network for resources can be prohibitively expensive if a large portion of the network needs to be searched. This was for instance the case in the Gnutella [38] system, where flooding was used for searching for a specific file. However, since virtualization of resources allows us to assume resources to be common, Zorilla will on average only need to search a small number of nodes before appropriate resources are found. Moreover, the properties of the network automatically optimizes the result for parallel applications, with resources found as close-by (measured in round-trip latency) as possible.

The resource discovery mechanism of Zorilla is very robust due to its P2P nature. Failing nodes do not hinder the functioning of the system as a whole, as resource requests will still be flooded to neighboring nodes. Also, new resources added to the system are automatically used as soon as neighbors start forwarding requests. We conclude that the combination of P2P and virtualization allows us to create a simple, efficient and robust scheduling mechanism in Zorilla.

2.2.2 Deployment

After resources for a job have been discovered, the job is deployed. This requires copying all input files, application executables, and possibly virtual machine (VM) images, to all nodes participating in the computation. For this reason, the submitting node acts as a file server for the job. It hosts files required to run the job, and provides a place to store output files. Unfortunately, the submitting node quickly becomes a bottleneck if a large number of nodes is participating in the job, or if it has a slow network connection. To alleviate this problem we again use P2P techniques: instead of transferring files from the submitting node only, nodes also transfer files among each other. Whenever a node requires a certain input file, it contacts a random node also participating in the job, and downloads the file from this peer node, if possible. As a fallback, the submitting node is used when the file is not present at any peer node.

When all files are available, the application is started on all resources, using a VM. Our current prototype implementation supports the Java Virtual Machine

(JVM) and the generic *Open Virtualization Format* (OVF), using Sun VirtualBox [79]. Apart from the benefit of platform independence, using a VM to deploy the application has three advantages. First, it allows for a very simple scheduling mechanism, as described in Section 2.2.1. Second, using a VM greatly simplifies the deployment of an application, especially on a large number of resources. Normally, an application needs to be compiled or at least configured for each resource separately. With a VM, the entire environment required to run the application is simply sent along with the job. This approach guarantees that the application will run on the target resource, without the need for configuring the application, or ensuring that all dependencies of the application are present on the resource.

The third advantage of using a VM is that it improves security. Since all calls to the operating system go through the VM, the system can enforce security policies. For instance, Zorilla places each job in a *sandbox* environment. Jobs can only read and write files inside this sandbox, making it impossible to compromise any data on the given resources. Although not implemented in Zorilla, the VM could also be used to limit access to the network, for instance by letting a job connect only with other nodes participating in the same job. Traditionally, security in distributed systems relies primarily on *security at the gates*, denying access to unknown or unauthorized users. As virtualization provides complete containment of jobs, the need for this stringent policy is reduced: unauthorized access to a machine results mainly in a loss of compute cycles.

Besides running the application on all used resources, Zorilla also offers support for communication between these resources. Since the resources used may be in different domains, communication may be limited by Firewalls, NATs and other problems. To allow all resources to communicate, Zorilla deploys a SmartSockets [54] overlay network. Applications which use the SmartSockets communication library automatically route traffic over this overlay, if needed. This ensures reliable communication between all resources used, regardless of NAT and Firewalls.

2.2.3 Job Management

The last subsystem of Zorilla that we will discuss is job management. On traditional (grid) middleware this mostly consists of keeping track of the status of a job, for instance *scheduling*, *running*, or *finished*. Instant cloud middleware has an additional task when managing a job: keeping track of the resources of each job. As resources may fail or be removed from the system at any time, a node participating in a parallel job may become unavailable during the runtime of the job. Traditional middleware usually considers a job failed when one of the resources fails. However, in an instant cloud changes to the set of available resources are much more common, making this strategy inefficient. Instead, in Zorilla users can specify a policy for resource failures. A job may be canceled completely when a single resource fails, resource failures can simply be ignored, or a new resource can be acquired to replace the old one. The last two cases require the application to support removing and adding resources dynamically. This can for instance be

achieved with FT-MPI [33], or the Join-Elect-Leave (JEL) model (see Chapter 5). Zorilla explicitly supports JEL, where the application is notified of any changes to the resources. Using this information, the application, or the runtime of the application's programming model, can react to the changes.

Zorilla implements all management functionality on the submitting node. This node is also responsible for hosting files needed for the job, and collecting any output files. Although it is in principle possible to delegate the management of a job to other nodes, for instance by using a Distributed Hash Table, we argue that this is hard to do reliably, and regard it as future work.

2.3 Related Work

There are several projects that share at least some of the goals and techniques of Zorilla. The most obvious related work to Zorilla are cloud middlewares, including Amazon EC2 [29], Eucalyptus [63] and Globus Nimbus [62]. All these middlewares are designed to turn a collection of machines into a cloud. One difference to Zorilla is the fact that these middleware assume no other middleware to be present, while Zorilla can also run on top of other middleware. Also, these middlewares all have centralized components, while Zorilla is completely decentralized. These systems are assumed to be installed (semi) permanently by system administrators, while Zorilla can be deployed on demand by users. One advantage the above systems have over Zorilla is the fact that all are generic systems, while Zorilla is mostly targeted to running HPC applications.

Zorilla can, through its use of the JavaGAT, use many, if not all, compute resources available to it. Some cloud computing systems sometimes support a so-called *hybrid* cloud model, where local, private resources are combined with remote, public, clouds. However, this model is more limited than Zorilla, which is able to use any resources, be it clusters, grids, clouds, or otherwise. Examples of systems supporting hybrid clouds are Globus Nimbus [62], OpenNebula [73], and InterGrid [22].

An element of Zorilla also present in other systems is its use of P2P techniques to tie together resources into a single system. However, these other systems [1, 13, 16] focus on providing middleware on bare resources, not taking into account existing middleware. Also, not all these systems assume virtualized resources, leading to rather complex resource discovery and allocation mechanisms.

Much like Zorilla, WOW [37] is able to create a single, big, distributed system out of independent resources. However, WOW was designed for high-throughput, rather than high-performance distributed applications. WOW only supports system level virtualization (VMware), while Zorilla also supports the much more lightweight Java virtual machine. Also, WOW routes all traffic between machines over a DHT style overlay network, limiting network performance. Lastly, WOW uses client-server schemes for services such as scheduling and files, while Zorilla implements these using P2P techniques.

ProActive [4] is another system which, like Zorilla, strives to use Java and P2P techniques [15] to run high-performance computations on distributed systems. However, Proactive primarily supports applications which use an ActiveObject model, while Zorilla supports any application, even non-Java applications. Also, ProActive requires the user to manually handle all connection setup problems and to manually (and statically) select the appropriate middleware.

Another approach to creating a cloud spanning multiple resources is used in the InterGrid [22] project. Here, gateways are installed which allow users to allocate resources from all grids which enter into a peering arrangement with the local grid. If remote resources are used, InterGrid uses virtualization to create a software environment equal to the local system. Unlike Zorilla, where resources can be added on demand, InterGrid gateways and peering agreements need to be setup in advance by system administrators.

2.4 Conclusions

The emergence of real-world distributed systems has made running high-performance and large-scale applications a challenge for end-users. These systems are heterogeneous, faulty, and constantly changing. In this chapter we suggest a possible solution for these problems: instant cloud middleware. We established the requirements of such a middleware, consisting mainly of the capability to overcome all limitations of real-world distributed systems. Requirements include fault-tolerance, platform independence, and support for parallel applications.

We introduced Zorilla, a prototype P2P middleware designed for creating an instant cloud out of any available resources, including stand-alone machines, clusters, grids, and clouds used concurrently, and running distributed supercomputing applications on the resulting system. Zorilla uses a combination of Virtualization and P2P techniques to implement all functionality, resulting in a simple, effective, and robust system. For instance, the flood-scheduling system in Zorilla makes use of the fact that resources are virtualized, allowing for a simple yet effective resource discovery mechanism based on P2P techniques.

Zorilla serves as a platform for the research presented in this thesis. Chapters 3 and 4 provide more detail on the techniques used in Zorilla. In Chapter 5 we discuss resource tracking, a vital part of application running in dynamic systems such as instant clouds, and explicitly supported in Zorilla. Chapter 6 describes world-wide experiments performed with Zorilla.

Chapter 3

ARRG: Real-World Gossiping*

Information dissemination in distributed systems is usually achieved through broadcasting. Commonly, broadcasting is done by building a broadcast tree, along which messages are sent. This approach can also be taken in peer-to-peer (P2P) systems [76]. However, maintaining such a tree structure in a large dynamic network is difficult, and can be prohibitively expensive. The broadcast tree may have to be rebuilt frequently due to changing network conditions or machines, or *nodes*, joining and leaving.

For P2P systems, an alternative to building broadcast trees is to use flooding [86]. Unlike broadcasting, flooding does not use any specific network structure to control the flow of messages between nodes. Upon receiving a new message, a node simply sends it to *all* its neighbors. With flooding, nodes can receive messages multiple times from different neighbors. Especially in situations where many nodes need to simultaneously and frequently disseminate information, both tree-based broadcasting and flooding are ineffective, because the number of messages passing through each node quickly exceeds its network capacity, processing capacity, or both.

Gossiping algorithms offer [32] an alternative to broadcasting and flooding when efficiency, fault tolerance and simplicity are important. The aim of gossiping algorithms is to severely limit the resources used at each node at any point in time. These algorithms are usually based on a small cache of messages stored on each node. Periodically, nodes exchange, or *gossip*, these messages with each other, thus updating their caches. This results in each node receiving a constantly changing set of messages. Over time, each node is likely, but not guaranteed, to receive each message in the system. Thus, with gossiping, the resource usage of a node is bounded in exchange for a slower rate of information dissemination. Also,

*This chapter is based on our paper published in *Proceedings of the 16th IEEE International Symposium on High-Performance Distributed Computing (HPDC 2007)* [24].

gossiping does not guarantee messages to be received in the same order they were sent, and messages might be lost or delivered multiple times.

Gossiping techniques are used in many applications, such as replicated name services [21], content replication [74, 47, 81], self configuration and monitoring [8], and failure detection [68]. In Zorilla, gossiping techniques are to manage the overlay network used for resource discovery. Most research on gossiping has relied on theory and simulations for evaluation of algorithms. However, because of inherent limitations, simulations cannot take into account all aspects of a real-world system.

This chapter focuses on the design, deployment and evaluation of gossiping algorithms in real-world situations. We focus on a gossiping-based membership algorithm capable of providing a *uniform random set of members* at each node, a problem commonly solved with gossiping techniques. Membership algorithms are an important building block for gossiping algorithms in general. In a membership algorithm, the messages which are gossiped actually contain identifiers of nodes, or *members*, of the system. Since most applications for gossiping algorithms rely on disseminating information uniformly across a certain system, the random members produced by the algorithms described in this chapter are ideally suited as targets for gossip messages.

The contributions of this chapter are as follows.

- We give an overview of all known and some new difficulties encountered when moving from simulation to application. These difficulties include race conditions due to non-atomic gossips, failing nodes, unreliable networks, and the inability to reach nodes because of firewalls and Network Address Translation (NAT).

- Although most gossiping algorithms are able to cope with *some* of the problems in real-world systems, there currently are no gossiping algorithms specifically designed to cope with *all known problems simultaneously.* We therefore introduce a simple, robust gossiping algorithm named *Actualized Robust Random Gossiping* (ARRG). It is able to handle all aforementioned problems.

- To address the connectivity problems encountered, we introduce a novel technique for ensuring the proper functioning of a gossiping algorithm, a *Fallback Cache.* By adding this extra data structure to an existing gossiping algorithm, the algorithm becomes robust against connectivity problems such as firewalls. The Fallback Cache is not limited to ARRG, but it can be used with any existing gossiping algorithm. We have, for instance, also applied it to Cyclon [81]. Maintaining a Fallback Cache does not incur any communication overhead and does not alter the properties of the existing algorithm, except for making it more robust.

- To compare gossiping algorithms and techniques, we introduce a new performance measurement: *Perceived Network Size.* This novel metric has the

advantage that it can be measured locally on a single node. Traditional metrics require global knowledge, which is impractical, if not impossible, to obtain in real-world systems. In addition, the Perceived Network Size metric can be used to determine behaviors of gossiping algorithms previously evaluated using multiple separate metrics. Also, our metric is able to clearly show differences in both efficiency and correctness between gossiping algorithms.

- We evaluate ARRG in *simulations* and on a *real-world system*, using several realistic scenarios with NAT systems, firewalls and packet loss. In these conditions, ARRG significantly outperforms existing algorithms. We also apply ARRG's Fallback Cache technique to an existing gossiping algorithm (Cyclon), and compare its performance with ARRG. We show that, in systems with limited connectivity, algorithms with the Fallback Cache significantly outperform gossiping algorithms without it. Even under pathological conditions with a message loss rate of 50% and with 80% of the machines behind a NAT, ARRG still performs virtually the same as under ideal conditions. Existing gossiping algorithms fail under these circumstances.

ARRG forms the basis of the overlay network used in Zorilla for resource discovery (see Chapter 2). ARRG is also used in the distributed implementation of our JEL resource tracking model (see Section 5.4.2 on page 68). For large-scale experiments done with Zorilla, including ARRG, see Chapter 6.

The rest of this chapter is organized as follows. In Section 3.1 we focus on the problems present in real-world systems as opposed to simulations and theory. We investigate to what extent current gossiping algorithms are able to handle these problems. In Section 3.2 we introduce the ARRG algorithm, followed by the introduction of the Fallback Cache technique in Section 3.3. We introduce the Perceived Network Size metric in Section 3.4. In Section 3.5 we evaluate the different algorithms and techniques in a number of use cases. Finally, we conclude in Section 3.6.

3.1 Real-World Problems & Related Work

In this section we give an overview of all known problems that must be addressed when implementing an abstract gossiping algorithm in a real system, and identify some problems not currently considered. We also examine related work, focusing on the solutions for these problems used by current algorithms. Table 3.1 gives an overview of the capability of different gossiping algorithms to address these problems. Each algorithm is rated according to its ability to overcome each problem. *NC*, sometimes listed alongside a rating, indicates that the literature available for this protocol does not consider this problem. In these cases we analyzed the algorithm itself to determine the rating. We also list ARRG, a simple and robust gossiping algorithm introduced in Sections 3.2 and 3.3.

	Node Failures	Network Unreliability	Non-Atomicity	Limited Connectivity
SCAMP [36]	+/-	+/-	NA	- NC
lpbcast [31]	+ NC	+/- NC	NA	- NC
ClearingHouse [21, 2]	+	+ NC	+	- NC
PROOFS [74]	+	+	+/-	- NC
Newscast [47, 82]	+	+/- NC	+/- NC	- NC
Cyclon [81]	+	+/-	+/-	- NC
ARRG, no Fallback	+	+	+	+/-
ARRG	+	+	+	+

- = unaddressed
+/- = partially addressed
+ = fully addressed
NA = Not Applicable
NC = Not Considered in the literature covered by this chapter

Table 3.1: Robustness of gossiping algorithms.

3.1.1 Node failures

The first problem is that of node failures. A node may leave gracefully, when the machine hosting it is shut down, or it might fail, for instance when the machine crashes. In this context, the joining and leaving of nodes during the operation of the system is often called *churn*. Most, if not all existing gossiping algorithms take node failures into account.

A common method to handle node failures is *refreshing*, where new identifiers of nodes are constantly inserted in the network, replacing old entries. Combined with redundant entries for each node, this ensures that invalid entries are eventually removed from the network, and an entry for each valid node exists. All protocols listed in Table 3.1 exhibit this behavior, except the SCAMP [36] algorithm.

As an optimization, failures can also be detected. For example, the Cyclon [81] algorithm removes nodes from its cache when a gossip attempt fails. This technique requires the algorithm to be able to detect failures. If an algorithm only sends out messages, without expecting a reply, a failure cannot be detected without extra effort.

3.1.2 Network unreliability

The second problem we face is network unreliability. Most simulations assume that all messages arrive eventually, and in the same order as they were sent. In real networks, however, messages may get lost or arrive out of order. This may have an influence on the proper functioning of the algorithm, especially when the message loss rate increases. If non-reliable protocols such as UDP are used, message loss can be caused by network congestion, limited buffer space, and firewalls dropping UDP traffic.

A partial solution for this problem is to use a network protocol capable of guaranteeing message delivery and ordering, such as TCP. However, whereas UDP is connectionless, TCP connections have to be initiated, and consume more local and network resources than the more lightweight UDP protocol. Finally, using TCP still does not make communication completely reliable, as TCP connections themselves can still fail due to network problems, timeouts or crashes.

To completely overcome the network unreliability problem a gossip algorithm needs to expect, and be robust against, message failures and out of order delivery. In the ClearingHouse [2] algorithm this is, for instance, handled by sending out multiple requests, while only a portion of the replies are required for a proper functioning of the protocol.

A gossiping algorithm is, by definition, unable to completely overcome network unreliability if it tries to maintain an invariant between communication steps. For instance, the Cyclon [81] algorithm swaps entries in each gossip exchange, where some entries from one node are traded for entries from another. Because this swap mechanism does not create or destroy entries, the number of replicas of each node identifier in the entire network remains constant. However, if a reply message is lost, this invariant does not hold, as the entries in the message have already been removed from the sender, but are not added to the cache of the node to which the message is sent.

3.1.3 Non-atomic operations

The third problem that is encountered when implementing gossiping algorithms on a real system is the non-atomicity of operations that involve communication. For example, the *exchange* of data between the caches of two nodes is often considered to be an atomic operation during the design and simulation of gossiping protocols. In reality, the exchange consists of a request and a reply message, separated in time by the latency of the network. Therefore, after a node has initiated a gossip, it can receive one or more gossip *requests* before its own gossip has finished (i.e., the gossip reply has not arrived yet). With some existing protocols, this can lead to data corruption of the gossiping cache.

In simulations, message exchanges are often instantaneous, and network latency is implicitly assumed to be zero. In reality however, latency can of significance, as the window for the aforementioned race condition increases as the network latency increases. Simply delaying incoming gossip requests until the initiated gossip has finished, leads to a deadlock if there happens to be a cycle in the gossiping chain. Another possible solution, ignoring concurrent gossips, as the PROOFS [74] algorithm does, leads to a high failure rate, as shown in [80].

We argue that, when a gossiping algorithm is designed, exchanging information between nodes cannot be considered an atomic operation. Care must be taken that the state of the cache remains consistent, even when a request must be handled while a gossip attempt is underway.

3.1.4 Limited connectivity

The fourth and last problem that must be faced in real-world networks is limited connectivity between nodes in the network. Most private networks use firewalls that block incoming connections. This effectively makes machines unreachable from the outside. Systems can usually still make connections to the outside, though sometimes even these are restricted.

Another type of device which limits connectivity between computers is a *Network Address Translation* system. These NAT devices make it possible for multiple computers in a network to share a single external IP-Address. The drawback is that, in general, the machines behind this NAT are not reachable from the outside.

Though methods exist to make connections between machines despite firewalls and NATs [54], these techniques cannot successfully be applied in all circumstances, and some connectivity problems remain. Moreover, mechanisms to circumvent firewalls and NATs often require considerable configuration effort, and typically need information about the network topology.

Most current gossiping algorithms are designed with the explicit [80] assumption that any node can send messages to any other node. Therefore these algorithms are not resilient to network connectivity problems.

There are systems that try to overcome limited connectivity, such as SmartSockets [54], Astrolabe [67, 8], Directional Gossip [51, 52] and Failure Detection [68]. However, these systems use an explicit structure on top of a traditional gossiping algorithm. Usually, a hierarchical network is built up manually to route traffic. To overcome the connectivity problem without requiring manual configuration and explicit knowledge of the network, new techniques are needed. We will introduce a novel solution, the Fallback Cache, in Section 3.3.

3.2 Actualized Robust Random Gossiping

To address the problems mentioned in Section 3.1, we introduce a simple gossiping algorithm, named *Actualized Robust Random Gossiping* (ARRG). This gossiping algorithm is an example of a gossiping algorithm specifically designed for robustness and reliability. ARRG is able to address node failures, network unreliability and does not assume that operations involving communication are atomic. ARRG uses randomness as a basis for making all decisions. This is done to make the algorithm robust against failures and to reduce complexity.

The pseudo code for ARRG is shown in Figure 3.1. Every time a node instantiates a gossip, it selects a random target node from its cache, and sends it a random set of cache entries, containing a number of elements from its own cache. The exact number is a parameter of the algorithm, denoted by SEND_SIZE. The node also adds an entry representing itself to the list sent (lines 6–11).

Upon receipt of the message, the target sends back a random set of cache entries, again including itself (lines 24–27). It also adds the received entries to its cache, ignoring entries already present in its cache. Since the cache has a fixed

```
1 void selectTarget() {
2     return cache.selectRandomEntry();
3 }
4
5 void doGossip(Entry target) {
6     //select entries to send
7     sendEntries = cache.
8         selectRandomEntries(SEND_SIZE);
9     sendEntries.add(self);
10
11     //do request, wait for reply
12     sendRequest(target, sendEntries);
13     replyEntries = receiveReply();
14
15     //update cache
16     cache.add(replyEntries);
17     while(cache.size() > CACHE_SIZE) {
18         cache.removeRandomEntry();
19     }
20 }
21
22 Entry[] handleGossipMessage(
23                     Entry[] sendEntries) {
24     //select entry to send back
25     replyEntries = cache.
26         selectRandomEntries(SEND_SIZE);
27     replyEntries.add(self);
28
29     //update cache
30     cache.add(sendEntries);
31     while(cache.size() > CACHE_SIZE) {
32         cache.removeRandomEntry();
33     }
34
35     return replyEntries;
36 }
```

Figure 3.1: Pseudo code for ARRG.

maximum size, the target may need to remove some entries if the maximum size is exceeded. The purged entries are selected at random (lines 29–33).

When the initiator of the gossip receives the reply from the target, the received entries are added to the local cache, ignoring duplicates. If this action increased the size of the cache beyond its maximum, random entries are removed until the number of entries becomes equal to the size of the cache, denoted by CACHE_SIZE (lines 13–19).

ARRG was explicitly designed to address the problems listed in Section 3.1 in a simple and robust manner. The main difference between other algorithms and ARRG is the explicit design choice to use the simplest solution available for each functionality required, while taking the problems listed in Section 3.1 into account. Moreover, any decision must be as *unbiased* as possible. An example of a bias is the removal of nodes after a failed gossip exchange with that node. This creates a bias against nodes which are not directly reachable because of a firewall or a NAT, making the content of a node's cache less random.

Table 3.1 on page 24 lists the capabilities of ARRG in comparison to other algorithms. An algorithm which can also overcome many problems in real-world systems is ClearingHouse [2]. However, ClearingHouse is significantly more complex than ARRG. This may create a bias against some nodes, hindering robustness. For instance, ClearingHouse combines caches from other nodes with the identifiers of nodes which requested its own cache. Therefore, nodes with better connectivity have a higher chance of appearing in caches. Another difference between ClearingHouse and ARRG is the bandwidth usage. ClearingHouse requests the complete cache from multiple nodes in each round, but disregards a large part of this information, wasting bandwidth. ARRG in contrast only transfers a small fraction of a single cache.

Node failures are handled in ARRG by the constant refreshing of entries. New entries are constantly added to the system, as old ones are purged. ARRG depends on random chance to purge old entries. Some protocols [81] also take into account the age of entries, but as this adds both complexity and a bias against old entries, ARRG simply replaces randomly selected entries each time a gossip exchange is done.

To address both the non-atomicity and the unreliability issue, a gossip request and its reply in ARRG can be seen as two separate gossips; there is no data dependency between the two. If either the request or the reply gets lost, both the node initiating the gossip and the receiver of the request are in a consistent state, and no information is lost. Because of this decoupling between a gossip request and reply, non-atomicity issues such as race conditions do not occur. ARRG explicitly does not assume the exchange to be an atomic operation.

To make ARRG more robust against connectivity problems nodes are *not* removed from the cache when a gossip attempt to this node fails. This is done because the difference between a node which is unreachable and a node which is defective is undetectable. So, to guarantee that nodes behind a firewall don't get removed from all caches, failed gossip attempts are simply ignored. The random replacement policy will eventually remove invalid entries.

Ignoring failed gossip attempts gives nodes behind a NAT or firewall a better chance of remaining in the cache of other nodes, and eventually getting to the cache of a node which *is* able to reach it. There is still a chance that a node may loose all cache entries to nodes it is able to reach, effectively leaving it with no nodes to contact and possibly removing itself from the network. ARRG contains an additional mechanism to keep this from occurring, our novel *Fallback Cache*. This technique is described in the next section.

Another reason ARRG is more robust against connectivity problems, is the *push-pull* mechanism used to perform a gossip exchange. When a node sends a gossip request containing some entries of its cache, the target of this message replies with some of its own entries. When a message is sent out by a machine behind a NAT or firewall, a reply is usually expected by this firewall or NAT. This makes it possible for a node to receive entries from the target node it sent a request to, even though the requesting node normally is unable to receive messages.

3.3 The Fallback Cache

As described in Sections 3.1 and 3.2, current solutions for the network connectivity problem do not always suffice. As a possible solution we introduce the *Fallback Cache*. This technique adds robustness to an existing gossiping algorithm, without changing in any way the functioning of the algorithm itself. The Fallback Cache acts as a backup for the normal membership cache present in the gossiping algorithm. Each time a successful gossip exchange is done, the target of this gossip is added to the Fallback Cache, thus filling it over time with peers which are reachable *by this node*. Whenever a gossip attempt fails, the Fallback Cache is used to select an entry to gossip with instead of the one selected by the original algorithm. Since this Fallback entry has already been successfully contacted once, there is a high probability that it can be reached again.

Figure 3.2 shows the necessary extra (pseudo) code to add a Fallback Cache to an existing algorithm. Line 2 shows the initialization of the Fallback Cache. The cache itself is a set of node identifiers. We assume the original gossiping algorithm performs a gossip once every T seconds by first selecting a target in a *selectTarget* function, followed by an attempt to contact this node to do the actual gossiping by a *doGossip* method. To use the Fallback mechanism, the *doGossipWithFallback* function starting at line 5 has to be called instead.

The *doGossipWithFallback* function mimics the original algorithm, by initially calling *selectTarget* and *doGossip* in lines 7 and 8. If the gossip attempt is successful, this node is added to the Fallback Cache in line 14 after which the function returns. The cache has a maximum size. If a new entry is added while it has already reached the maximum size, a random entry is removed from the cache after the new entry is added (lines 15–17).

If the attempt is unsuccessful, an entry is selected from the Fallback Cache in line 20, and another gossip attempt is done (line 22) with this new target. This retry is only done once. When it also fails (or when the Fallback Cache is empty) the algorithm gives up, and the next gossip attempt will be done whenever the function is called again (usually after some fixed delay).

Although the original gossiping algorithm may remove entries when a gossip attempt fails, the Fallback Cache will never remove invalid entries. This is done to make the Fallback Cache more robust. A node that is currently not reachable because of network problems, or because it is overloaded with requests, may become reachable again later. If invalid entries are removed from the Fallback Cache

```
1 // extra initialization
2 fallbackCache = new Set();
3
4 // gossip function
5 void doGossipWithFallback() {
6     // call existing algorithm functions
7     target = selectTarget();
8     doGossip(target); // could fail because
9                       // of a timeout or a
10                      // connection problem
11
12    if(successful) {
13        // remember this target
14        fallbackCache.add(target);
15        if(fallbackCache.size()>CACHE_SIZE) {
16            fallbackCache.removeRandomEntry();
17        }
18    } else {
19        // retry with Fallback entry
20        target = fallbackCache.
21                        selectRandomEntry();
22        doGossip(target); // if this fails,
23                          // just ignore it
24    }
25 }
```

Figure 3.2: Fallback Cache pseudo code.

it might become empty, leaving the node with no nodes to contact. Invalid entries will eventually be overwritten by new entries when successful gossip exchanges are performed, thanks to the random replacement policy.

The Fallback mechanism does not interfere with the normal operation of a gossiping algorithm. If no errors occur, the Fallback Cache is never used. Therefore, all the properties present in the original algorithm are retained. This makes it possible to add robustness against connectivity problems without redesigning the rest of the algorithm.

When errors *do* occur, the Fallback mechanism guards against the cache of the node containing only invalid entries. Since the Fallback Cache holds only entries which were valid at the time and location they were added, this provides a much more reliable source of valid entries than the original cache, greatly reducing the risk that a node gets disconnected from the network, and reducing the risk that the network partitions.

The Fallback Cache in itself is not a good source for uniformly random nodes. The update rate is slow at one entry per gossip round, and the mechanism relies on the original gossip cache to provide new entries. The combination of the original

and the Fallback Cache is therefore needed to provide a valid and robust gossiping algorithm. The original cache will contain a random subset of all the nodes in the system, while the Fallback Cache contains a random set of all the *reachable* nodes.

3.4 Perceived Network Size

Ideally, a gossiping algorithm should populate node caches such that the chance of finding an item in a particular cache is both independent of the other items in the cache, and independent of the items in neighboring nodes' caches. We define a gossiping algorithm to be *viable* if it is unlikely to result in a partitioning of the network. The closer an algorithm is to the ideal random case, the faster information is likely to move between any two chosen nodes, and thus the more efficient it is.

Previous analytical studies have mostly measured the degree of randomness in communication of gossiping algorithms by considering properties of the overlay network formed by neighborhood caches. For instance, the eventual occurrence of network partitions [31, 2], the presence of small-worlds like clustering [47, 82], or a widely varying node in-degree [2, 81] all indicate a certain amount of relatedness between cache contents. Such measures, however, require taking a snapshot of the state of all nodes, something which is non trivial in a real network. Voulgaris [80] also examines the randomness of the stream of items received by a single node using the Diehard Benchmark of statistical tests for pseudo-random number generators. These tests, however, only allow protocols to be labeled as random-enough, or not-random-enough. Moreover, running such a benchmark is costly, making it impractical to use in most systems.

For this study we introduce a novel measure of gossiping randomness, the Perceived Network Size, that can be applied at a single node, and that can distinguish differences in the degree of randomness of two algorithms, or in the degree of randomness at different locations in a network.

Consider the inter-arrival times, R, of items from a given node at another node in the network. Given that in a gossiping protocol all nodes enter the same number of items into the system, for a fully random protocol each item received should have a probability of $1/N$ of being from the selected node, where N is the number of nodes in the network. The distribution of the random variable R thus gives an indication of the randomness of the gossiping process. For instance, the average value of R, for a large enough sample size, will be N. Further, for a uniformly random process, a smaller sample size will be needed to get the correct average, than for a process where the distribution of items is skewed. Based on this, we define the *Perceived Network Size*, as the average inter-arrival time of a specific node identifier in a node's cache. In practice, we calculate the Perceived Network Size using the inter-arrival time of all nodes in the network (and aggregating all these separate rates) rather than a single node, to increase accuracy and to speed up data collection. By examining the rate at which Perceived Network Size reaches the correct value as a node receives more information over time, we can compare

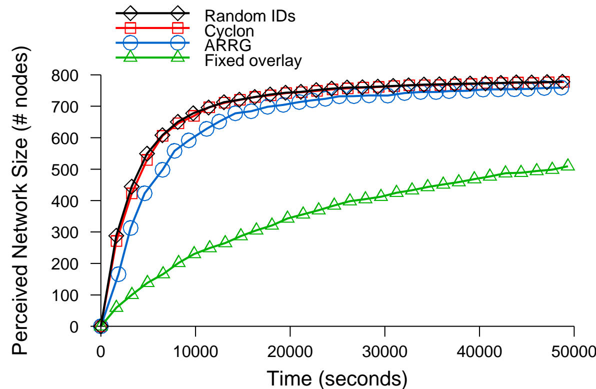


Figure 3.3: Perceived Network Size for four protocols.

the relative randomness of two gossiping processes, or of the same process at different nodes.

Figure 3.3 compares perceived network size versus time for ARRG, Cyclon, a protocol like ARRG but in which nodes' neighbors are fixed (Fixed Overlay), and for a stream of node IDs generated using a random number generator (Random IDs). All of these protocols can be considered to be *viable* in this setup as they all eventually reach the correct perceived network size of 800 nodes. The rate at which this value is reached, however, shows that the protocols in which the neighborhood overlay changes are significantly more *efficient* than the Fixed Overlay protocol. It can also be seen that ARRG is slightly less random than Cyclon, indicating the effect of the more precise age-based item removal used in Cyclon. Cyclon is almost indistinguishable from the "true" random process.

Besides allowing us to accurately compare viable protocols, the Perceived Network Size can also be used to detect the various behaviors considered in earlier studies. Protocols that result in small-worlds like clustering or exhibit widely varying node in-degrees are less efficient, and thus find the correct Perceived Network size more slowly. Network partitions result in a Perceived Network Size that is smaller than expected, as seen later in Figure 3.9. Slowly partitioning networks result in a decreasing Perceived Network Size (Figure 3.12).

3.5 Experiments

To test the effectiveness of ARRG and its Fallback Cache technique we examine a number of different use cases. All experiments where done with Zorilla, which uses gossiping mechanism for resource discovery purposes (see Chapter 2). We evaluate the techniques described in this chapter both with a *real* application on

a *real* system, and using simulations. For the experiments we use the *Distributed ASCI Supercomputer (DAS-3)* [20] , which consists of 5 compute clusters located across the Netherlands.

Zorilla includes implementations for both ARRG and Cyclon using the TCP protocol, and a Fallback Cache for both algorithms. Connections are set up using the SmartSockets library [54]. Both the normal and the Fallback Cache (if present) hold 10 entries. Each gossiping algorithm initiates a gossip exchange with a node from its cache every 10 seconds. During this exchange 3 entries are sent by each node.

To make a fair comparison between the algorithms, we implemented the possibility to run multiple gossiping algorithms with different implementations and settings concurrently. Each running algorithm has its own private data structures, and does not interact with other instances in any way.

When a gossiping algorithm starts, it needs some initial entries to gossip with. To start, or *bootstrap* the gossiping algorithm in Zorilla, or in this case multiple algorithms running concurrently, each node is given one or more other nodes when it starts running. This is usually a single node started before the others. In practice, the node used to bootstrap the system will probably not run permanently. So, to make the experiment as realistic as possible, the bootstrap entry is only available to each gossiping algorithm for a limited time, in this case 100 seconds. If a gossiping algorithm could use this bootstrap indefinitely, it would always be possible to restart an algorithm, making the difference between a failing gossiping algorithm and one performing poorly less distinguishable.

For the experiments on a real system with 80 nodes we use small cache and gossip sizes of 10 and 3 items respectively. For simulations of 8000 nodes, we maintain the same cache-to-gossip-ratio, but speed up the system by using larger values of 100 and 30 items.

3.5.1 The Fully Connected Scenario

For reference, and to determine the overhead of the Fallback Cache technique, we first measure on a network without any connectivity problems. We compare ARRG to ARRG without a Fallback Cache. Figure 3.4 shows the results for the test system. There is no significant difference between ARRG with and without a Fallback Cache. As a Fallback Cache is only used when connectivity problems occur, this is expected. From this graph we can conclude the Fallback Cache causes no overhead. The simulated results are not shown here, as they are identical to the results from the experiment. Also note that the Perceived Network Size of ARRG is converging to 80, the actual network size, thus showing that ARRG is functioning properly in this setting.

3.5.2 The X@Home Scenario

To test the effectiveness of a Fallback Cache in more demanding situations, we continue with some experiments in a *X@Home* setting. In this scenario, a number

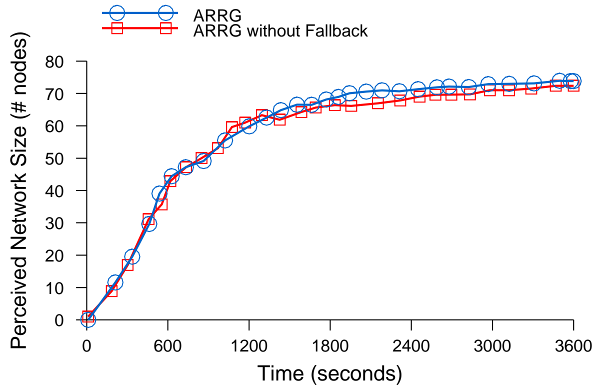


Figure 3.4: ARRG in a fully connected network.

of home users join together to form one big peer-to-peer network. Examples of such systems are the SETI@Home [71] project and file sharing applications.

The main reason for home users to have connectivity problems are NAT systems. As explained in Section 3.1.4, these are used to share a single connection between multiple machines, leaving these machines unreachable from the outside. Not all home users have a NAT system, as they may only have a single machine and do not need to share the connection. Also, some methods [54] exist to allow a machine behind a NAT to be reachable from the outside. In this scenario the machines connected to the peer-to-peer network can therefore be divided into two types. One type which is reachable from other nodes in the system, and one which is not.

As running an experiment with a large number of home machines is impractical, we use a simple technique to achieve the same result on our DAS-3 system. Each node which is in the group of *home* machines does not accept any incoming connections, and is only able to make outgoing connections. This is done at the socket level, so neither the application nor the gossiping algorithm itself are aware of their inability to receive gossip requests.

As a reference, we also include a *retry* version of ARRG. If an attempt to gossip fails, this version, like the Fallback version, attempts a new gossip with a different target. However, this new target is not selected from a special cache, but uses the normal selection mechanism instead. In the case of ARRG, this is simply a random entry from the cache.

This experiment consisted of 64 unreachable home machines and 16 *global* machines which are reachable normally, for a total of 80 nodes. Figure 3.5 shows the performance of ARRG on a global node. We compare ARRG with a version without a Fallback Cache and with a version that uses retries. The graph shows that ARRG without the Fallback Cache is not a viable algorithm. As expected,

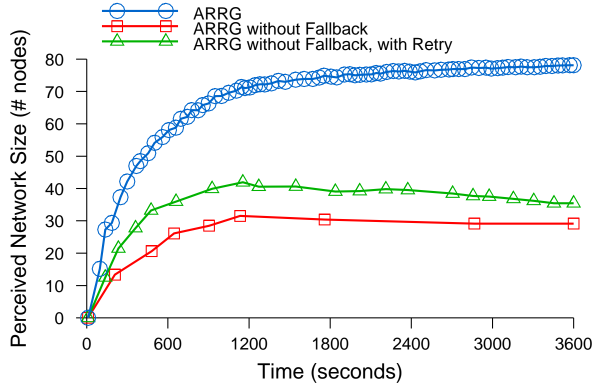


Figure 3.5: ARRG in the X@Home scenario.

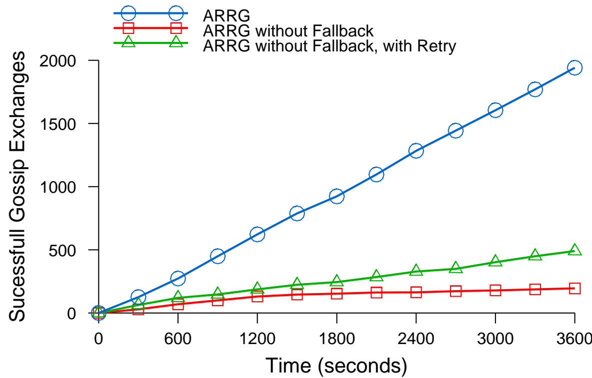


Figure 3.6: Gossip exchanges for ARRG in the X@Home scenario.

it is unable to overcome the connectivity problems present in this scenario. The Perceived Network Size does not reach the actual network size, but only reaches 40, indicating that the gossiping network has partitioned. It also shows that employing a retry strategy, where another target is picked at random when a gossip attempt fails, also does not result in a viable protocol. However, when a Fallback cache is used, ARRG is able to perceive the entire network, showing almost identical performance to the fully connected reference case shown in Figure 3.4.

To clarify the underlying reasons for this difference in performance between the various versions of ARRG, Figure 3.6 shows the number of successful gossip exchanges for each version. These include both gossip exchanges initiated by the

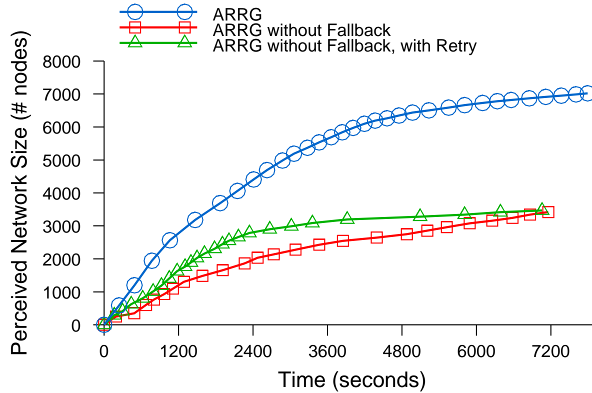


Figure 3.7: ARRG in a 8000 node simulation of the X@Home scenario.

node and incoming requests from other nodes. The graph clearly shows that the Fallback Cache version of the algorithm performs almost an order of magnitude more successful gossip exchanges than the other versions. A large percentage of entries in the cache of each version are home nodes, which causes a gossip attempt to fail when selected. With the Fallback Cache in place, the algorithm will always choose a valid node in the second attempt. In the version without a Fallback Cache and with the retry version, there is no such guarantee, causing gossips to fail frequently.

To test the validity of our findings in a larger environment, we also ran a simulation of the X@Home scenario. The simulation considers 8000 nodes, consisting of 6400 home nodes and 1600 global nodes. To produce an information dissemination rate similar to that in Figure 3.5 we changed several parameters of the algorithm, as discussed in the introduction of this section. The cache size of each node was 100, and 30 entries were exchanged in each gossip. The size of the Fallback Cache was not increased, and remained at 10. Figure 3.7 shows the results of a simulated run of 2 hours. It shows that the performance of ARRG and the Fallback Cache technique are comparable to our real measurements.

3.5.3 The Real-World Distributed System (RWDS) Scenario

The last scenario where we test the Fallback Cache mechanism is the *Real-World Distributed System (RWDS)* scenario, depicted in Figure 3.8. This system consists of multiple separate clusters of machines which are protected by a firewall. This firewall will deny incoming connections from outside the cluster. However, connections between nodes inside a single cluster and outgoing connections are still possible. Each cluster has a single node which resides on the edge of the cluster, a so called *head node*. This node is able to receive incoming connections from

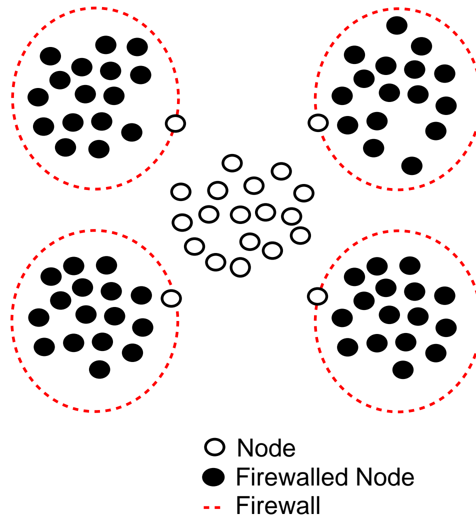


Figure 3.8: RWDS use case Network Model.

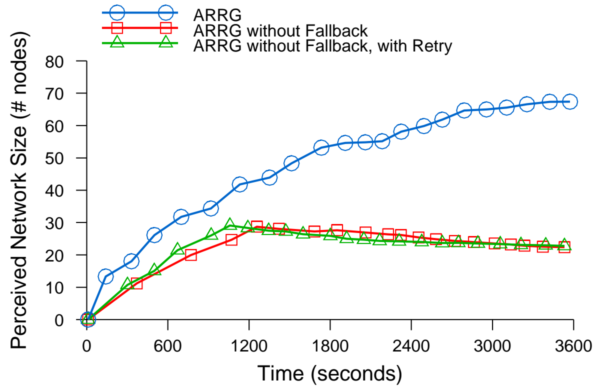


Figure 3.9: ARRG in the RWDS scenario.

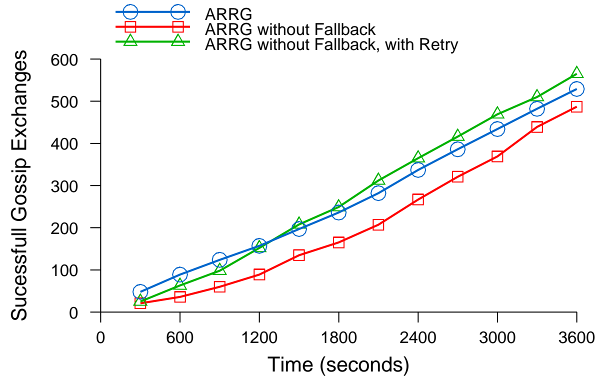


Figure 3.10: Gossip exchanges for ARRG in the RWDS scenario.

the outside, as well as connect to the nodes inside the cluster. We implemented this setup using the SmartSockets library [54], which is able to selectively deny connections if requested using a configuration file. Again, nodes are not aware of these restraints as they are enforced at the socket level. Our setup consisted of 4 clusters with 16 machines and one head node. The system also contained 17 *global* nodes, which were not protected by a firewall. The total number of machines in this system is therefore 85.

Figure 3.9 show the results for this experiment, measured at a global node. As expected, both the retry version of ARRG and the version without the Fallback Cache are not viable in this setting. From 1200 seconds onward, both versions show a decline of the Perceived Network Size. This shows that the gossiping network has partitioned, and nodes are only gossiping with a subset of the network. Analysis of the contents of the gossip caches shows that each node is in fact only communicating with nodes inside its own cluster. The Fallback mechanism, again, is able to compensate for the connectivity problems. The Fallback Cache contains a random subset of all *reachable* nodes, in this case both nodes within a node's own cluster and the global nodes in the system.

Figure 3.10 shows the number of successful gossip exchanges done by each algorithm. Unlike the previous experiment, there is only a small difference between them. In the previous experiment, failure of the protocol was caused by a *lack of reachable nodes*. In the RWDS case, being able to reach nodes is not sufficient. These nodes also need to be distributed correctly. The retry version of ARRG and the version without the Fallback Cache are only gossiping with nodes inside their own cluster, causing the network to partition.

Notice that the performance of the Fallback Cache in the RWDS scenario is slightly less than its performance in the X@Home and fully connected scenarios. This is due to the fact that the cache of the algorithm is not able to determine

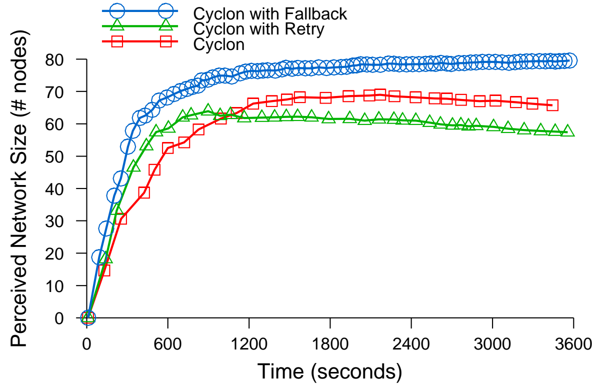


Figure 3.11: Cyclon in the X@Home scenario.

the difference between nodes inside its own cluster and global nodes. With this information, the number of gossips within its own cluster could be limited. This is important because nodes in the local cluster usually have a tendency to contain a less random set of the world than global nodes. Cluster information is available in the gossiping algorithms that use structure such as Astrolabe [67], but this information needs to be supplied manually. We leave automatically determining and exploiting network structure as future work.

3.5.4 Cyclon

To determine the effectiveness of our Fallback mechanism in other algorithms than ARRG, we also tested it with the Cyclon [81] gossiping algorithm. We compared three different versions of Cyclon, the regular Cyclon algorithm, a version which retries once in case of a failure, and a version including our novel Fallback cache. Figure 3.11 shows the result for the X@Home scenario (on a global node). In this case the normal version of Cyclon is not viable. Although it almost reaches the entire network size, the Perceived Network Size quickly starts declining. Over time, the Cyclon network partitions. The same was observed for the retry version. With the addition of the Fallback Cache, Cyclon performs very well in this scenario. The Fallback cache completely compensates for the connectivity problems, and Cyclon is as fast as it was in the ideal situation (See Figure 3.3). Thus, with Fallback Cyclon is viable in the X@Home scenario.

Figure 3.12 shows Cyclon in the RWDS scenario. Normal Cyclon again fails, though much faster this time. It clearly creates a partition of the network, as the Perceived Network Size is converging slowly to the size of a single cluster. Cyclon with retry manages to overcome the network limitations. However, this is most likely due to the aggressive purging by Cyclon. In a slightly modified setting, for

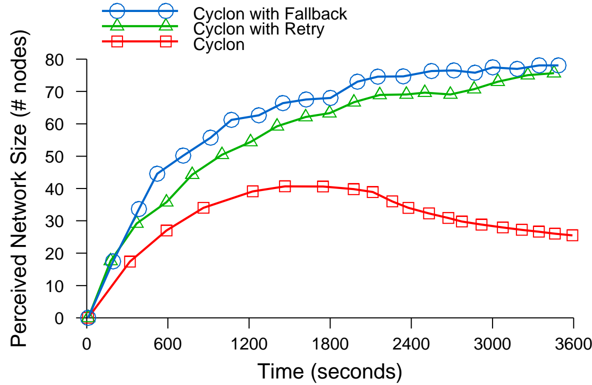


Figure 3.12: Cyclon in the RWDS scenario.

instance with fewer global nodes, the retry version will probably fail as well. The Fallback version of Cyclon again is viable, converging to the full network size.

3.5.5 Pathological Situations

By design, ARRG and its Fallback cache are robust against transient network problems such as congestion or link failure. This is achieved by retaining as much information as possible. Both ARRG's normal and Fallback caches do not remove invalid entries, but only replace entries with new valid entries as they become available. Reduced network performance therefore does not result in the loss of entries. To test this robustness we performed experiments in two pathological cases.

The first system setup is identical to the X@Home case, but in addition, we introduced 50% messages loss. This performance degradation could for instance occur in an environment where the network is congested, or where a poor wireless connection is present. We implemented the message loss at the socket layer. Figure 3.13 shows the results for both ARRG and Cyclon. Several conclusions can be drawn from this graph. First, the ARRG algorithm is able to overcome even this pathological case, without any noticeable performance degradation. Second, the Cyclon algorithm is not viable in this scenario. Even with the addition of a Fallback Cache, Cyclon does not perform well, and is not viable.

The reason for Cyclon's low performance can be found in the manner Cyclon handles failures. When a gossip attempt to a node fails, Cyclon removes its entry. For a gossip exchange to succeed both the request and the reply need to be delivered successfully. Since half of the messages are lost, the failure rate in this scenario is 75%. This causes Cyclon to quickly run out of entries in its cache. Adding a Fallback Cache only partially fixes this problem, as Cyclon is unable to

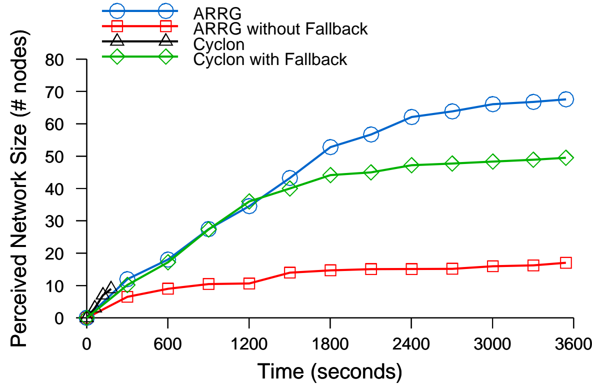


Figure 3.13: ARRG and Cyclon in the X@Home scenario, with 50% message loss.

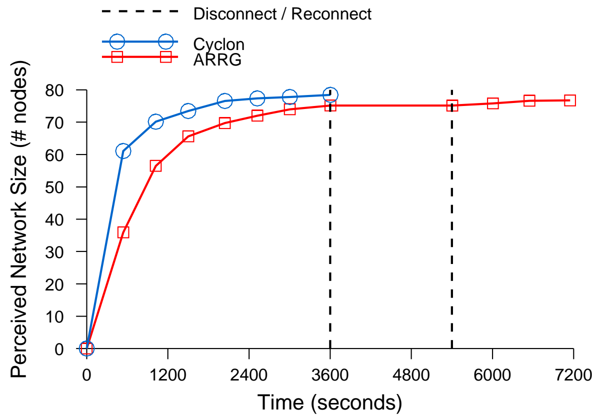


Figure 3.14: ARRG and Cyclon in the Disconnect scenario.

do enough successful exchanges to fill its cache with valid entries, and relies almost solely on the Fallback cache.

In the second test case nodes become disconnected from the network for a time. The results of this test are shown in Figure 3.14. The setup is identical to the Fully connected scenario of Section 3.5.1. After an hour, 16 of the 80 nodes are disconnected from the network, and are neither able to reach the remaining 64 nodes nor the other disconnected nodes. After another half an hour the nodes are reconnected.

	Fully Connected	X@Home	RWDS	Message Loss	Disconnect
Cyclon	+	-	-	-	-
Cyclon (retry)	+	-	+/-	-	-
Cyclon (with Fallback)	+	+	+	+/-	+
ARRG (no Fallback)	+	-	-	-	+
ARRG (retry)	+	-	-	-	+
ARRG	+	+	+	+	+

+ = pass
- = fail
+/- = reduced performance

Table 3.2: Results of experiments for different scenarios and algorithms.

The graph shows ARRG recovers from the network problems, and Cyclon fails to resume gossiping after the network is restored. ARRG is able to resume as the normal and the Fallback cache still contain valid entries, as ARRG does not remove any entries during the disconnect. Cyclon does remove invalid entries, diminishing the number of entries in the Cyclon cache during the time the network is disconnected. At the time the network is restored, Cyclon does not have any entries in its cache left, and is thus not able to resume gossiping. We found that adding an additional Fallback cache makes Cyclon robust against this failure, as the Fallback cache still contains valid entries (not shown).

3.5.6 Experiment Summary

Table 3.2 provides a summary of all experiments performed in this section. It shows that ARRG with a Fallback cache is able to overcome all problems presented. It also shows that the Fallback cache is an invaluable part of ARRG, as without it ARRG does not function properly in all cases. Adding the Fallback cache to Cyclon significantly increases its robustness, making it robust against all problems, though in the scenario where we introduce message loss Cyclon's performance is reduced.

3.6 Conclusions

In this chapter we studied the design and implementation of gossiping algorithms in real-world situations. We addressed the problems with gossiping algorithms in real systems, including connectivity problems, network and node failures, and non-atomicity. We introduced *ARRG*, a new simple and robust gossiping algorithm. The ARRG gossiping algorithm is able to handle all problems we identified by systematically using the simplest, most robust solution available for all required functionality. The *Fallback Cache* technique used in ARRG can also be applied to

any existing gossiping protocol, making it robust against problems such as NATs and firewalls.

We introduced a new metric for the evaluation of gossiping algorithms: *Perceived Network Size*. It is able to clearly characterize the performance of an algorithm, without requiring information from all nodes in the network. We evaluated ARRG, in several real-world scenarios. We showed that ARRG performs well in general, and better than existing algorithms in situations with limited connectivity. In a pathological scenario with a high loss rate and 80% of the nodes behind a NAT system, ARRG still performs well, while traditional gossiping techniques fail.

ARRG is used as the basis of our Zorilla middleware (see Chapter 2). Also, ARRG is used in the distributed implementation of our JEL resource tracking model introduced in Chapter 5. In Chapter 6 we discuss large-scale experiments that include ARRG.

Chapter 4

Flood Scheduling: Simple Locality-Aware Co-allocation*

In this chapter we will examine the design and implementation of the scheduling mechanisms present in Zorilla. Figure 4.1 shows the life cycle of a (parallel) job on Zorilla. For an overview of Zorilla, see Chapter 2.

The scheduler of Zorilla is very much tailored for the *Instant Cloud* nature of Zorilla. It assumes the system is used by relatively small numbers of users concurrently, and that most application are fault-tolerant, and are *malleable*, i.e., able to handle changes in the number of resources used.

The scheduler of Zorilla has a number of requirements. The defining property of Zorilla is that it uses peer-to-peer (P2P) techniques extensively, and that it has no central components. As a first requirement, the scheduler therefore needs to be fully distributed, as it would otherwise compromise the P2P nature of Zorilla.

Zorilla provides support for *distributed supercomputing* applications: high-performance applications which use resources spanning multiple sites simultaneously. Therefore, the second requirement of the scheduler is that it must support *co-allocation* [55], the simultaneous allocation of multiple computational resources, even if these resources are distributed over multiple sites.

Third, the scheduler needs to be *locality aware* [88] in that it should schedule jobs with regard for the distance between machines. If a single computation is executed on resources which are too far apart, the communication cost may become too high. Fourth and last, the scheduling algorithm needs to be robust against changes in the environment.

The scheduler of Zorilla is based on *flooding* (a form of selective broadcast). It floods messages over a P2P overlay network to locate available resources. Flooding

*This chapter is based on our paper published in *Sixth International Workshop on Global and Peer-2-Peer Computing (GP2P 2006)* [25].

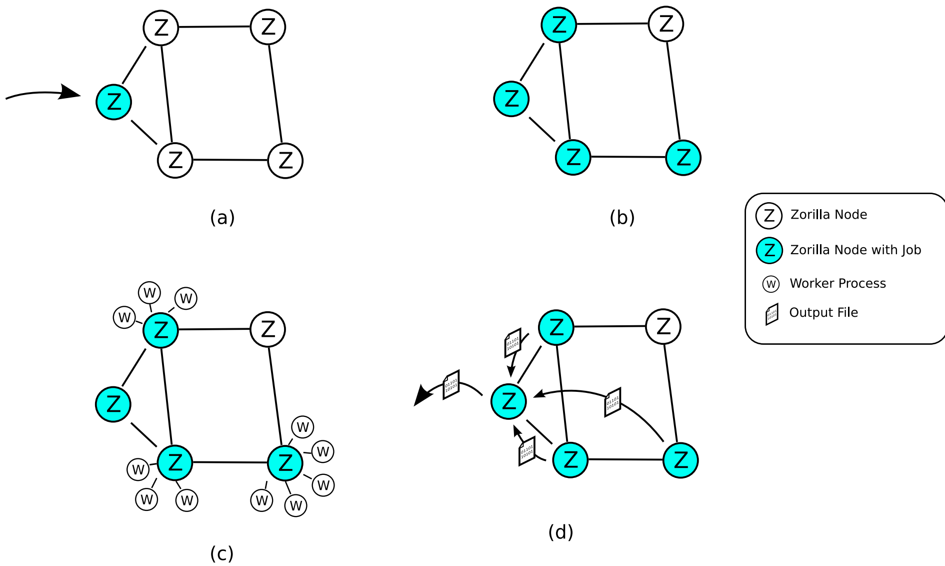


Figure 4.1: Job life cycle in a Zorilla system consisting of 5 nodes connected through an overlay network. (a) A (parallel) job is submitted by the user to a node. (b) The job is disseminated to other nodes. (c) Local schedulers at each node decide to participate in the job, and start one or more Worker processes (e.g. one per processor core available). (d) Output files are copied back to the originating node. (This figure is identical to Figure 2.2, but is repeated here for convenience.)

is done within a certain radius, which influences the number of nodes reached. Zorilla dynamically determines a suitable flood radius. This scheduling algorithm adheres to all the requirements stated above.

This chapter is organized as follows. Section 4.1 describes related work. Section 4.2 describes how scheduling is performed and implemented in Zorilla. In Section 4.3 we perform several experiments to see how efficient Zorilla is in scheduling jobs. Finally, we conclude in Section 4.4.

4.1 Related Work

In this section we will give an overview of other distributed systems and compare them to Zorilla, focusing on scheduling mechanisms.

Zorilla shows similarities to so-called *global computing (GC)* systems such as SETI@home [71], XtremWeb [14] and distributed.net [23]. GC systems use spare cycles of workstations to run primarily master-worker type programs. Workers download jobs from a server and upload results when the job is completed. This server is the main difference with Zorilla. In Zorilla, jobs are not stored on a server. Instead, storage and scheduling of jobs is performed in a decentralized manner. Another difference between Zorilla and GC systems is that few GC systems support supercomputing applications. If they do (e.g. in the case of XtremWeb) a central scheduler is used to perform the necessary co-allocation.

Another type of system related to Zorilla is grid computing with systems such as Globus and Unicore. These systems typically use compute clusters located at different sites. Every site has a scheduler for the jobs local to the site, and it is possible to submit jobs to remote sites. Co-allocation is usually not directly supported but relies on a centralized meta-scheduler instead, for instance Koala [55]. The main difference to Zorilla is this centralized nature of schedulers.

The file search mechanism in the Gnutella P2P file sharing system [38] formed a basis for the resource discovery protocol in Zorilla. In Gnutella, a node sends a search request to its direct neighbors, which in turn send it to all their neighbors. This process continues until a message is forwarded as many times as specified by the user when the search was initiated. The Zorilla search algorithm extends this algorithm in several ways, including dynamically determining a suitable number of times to forward a message, and using the locality awareness of the P2P network to optimize the selection of nodes reached. In Gnutella, a search will reach nodes randomly distributed throughout the network, while in Zorilla nodes are reached that are most likely close to the node where the flood was initiated (see Section 4.2 for more details).

Like Zorilla, the P2P infrastructure present in ProActive [11] strives to use P2P techniques to perform parallel computations. There are several differences though. ProActive primarily supports applications which use an ActiveObject model. Zorilla, on the other hand, supports any application. Another difference lies in the mechanism to discover resources. Like Zorilla, ProActive uses a mechanism derived from the Gnutella system to locate resources, but it extends it differently. For in-

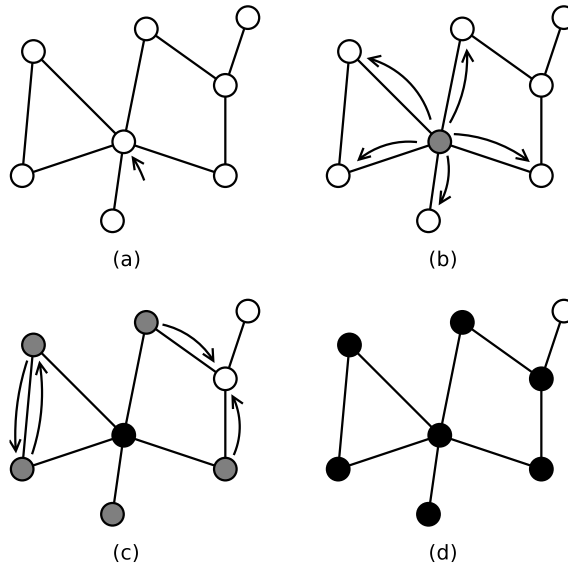


Figure 4.2: Flooding a message with a radius of two.

stance, ProActive has no way of dynamically determining a suitable parameter for the *flood* of the resource request. Also, ProActive is not locality aware, as resource selection is based on the load of a remote node, not the distance to other nodes participating in the computation. This approach leads to performance degradation if this mechanism is used to run supercomputing applications on distributed resources.

4.2 Flood Scheduling

In this section, we will discuss the scheduler of Zorilla. As the scheduling algorithm uses *flooding* to locate resources, we dub this mechanism *flood scheduling*.

When a job is submitted to the Zorilla system, an *advertisement* is created for this job. This advertisement is a request to join the computation of the advertised job. It contains information needed to start the job and any resource requirements the job has.

The advertisement for a job is *flooded* by the origin of that job to other nodes in the Zorilla system. In Figure 4.2, a flood is executed of a message in a Zorilla network consisting of eight nodes. In Figure 4.2(a), the flood is started at the node marked with an arrow. This origin node then sends the message to all its neighbors in the overlay network in Figure 4.2(b). The neighbors of the origin node will, in turn, forward the message to all their neighbors, shown in Figure 4.2(c). In this picture the two nodes on the left forward the message to each other. If a node receives a duplicate message it will be discarded instead of being forwarded.

A flood has a certain *radius*, to limit the scope of the flood. The origin of a flood sets the *time to live (TTL)* of the message it sends to the radius of the flood. When a node forwards a flood message to its neighbors it decreases the TTL of the message by one. If the TTL reaches zero it is not forwarded anymore. In Figure 4.2 the flood is executed with a radius of two. The neighbors forward the message with a TTL of one in Figure 4.2(c), and these messages are subsequently not forwarded anymore. Figure 4.2(d) shows the situation after the flood has ended. All nodes with a distance of two or less from the origin of the flood have received the message.

When a node receives an advertisement for a job, it checks if all the requirements of that job are met. Requirements are for instance the availability of processing power, memory and disk space. If enough resources are available, the node downloads the input files and executable for the job, and starts one or more workers.

The effect of flooding the advertisement with a certain radius from the origin of a job is that the advertisement is delivered to all nodes close to the origin with a certain maximum distance. This maximum distance is dependent on the radius of the flood. A small radius will only reach nodes very close to the node, a large radius will reach more distant nodes as well, and a very large radius might reach all nodes in the entire system. Since the overlay network used in Zorilla is locality aware, nodes nearby in the overlay network are also nearby (in terms of latency) in the physical network. A flood therefore reaches the nodes physically closest to the origin node.

Zorilla uses the following heuristic algorithm to determine the best (as low as possible) radius for the flood that is used to start a job. First, it calculates an initial low estimate. The heuristic currently used by Zorilla is the base 10 logarithm of the number of nodes required, but any conservatively low estimate is sufficient. The origin of the job then initiates a flood of the advertisement with the estimate radius, causing a number of nodes to respond with a request to join the computation. The origin node then checks to see if enough nodes responded to the advert. If not, it will increase the radius used by one, and send a new flood. As nodes usually have in the order of 10 neighbors each, this increase of the radius by one causes the request to be send to an order of magnitude more nodes than the previous request. Any nodes responding to the new request will be added to the list of nodes in the computation. As long as there are not enough nodes present to perform the computation, it will keep sending floods with increasing radius.

As each flood is done independently, any new nodes which join the network close to the submitting node will automatically receive the request message when the next flood is done. Also, nodes which fail are automatically ignored when a flood is sent as they are no longer part of the network. This makes flood scheduling both flexible and fault-tolerant.

Scheduling in Zorilla meets the four requirements as listed in the introduction of this chapter. First, it uses a *decentralized* scheduling algorithm. Second, the scheduling mechanism supports co-allocation, though in a somewhat limited way. In contrast to most co-allocating schedulers, Zorilla does not schedule nodes on

multiple compute sites explicitly. Instead, it treats all the nodes as one big system, enabling usage of nodes in the entire system. Third, scheduling in Zorilla is locality aware. It is beneficial to select nodes to compute a job from nodes nearby the origin. Zorilla uses the latency between nodes as a measure for distance, and schedules jobs on nodes both near the origin and close to each other. Last, the scheduling algorithm of Zorilla is robust against changes in the environment. New nodes are automatically considered for running a job and nodes leaving the network do not compromise the functioning of the scheduling.

4.2.1 Optimizations

Several optimizations are present in Zorilla to further optimize scheduling. First, if a node is not able to join a computation when it receives an advertisement because of lack of resources, or because it is already running another job, it will not totally discard the advertisement. Instead the advertisement is put in a pending job queue. Periodically, each node will scan its queue and start computing any job which resource requirements are met. To keep jobs from staying in the queue forever, an advertisement comes with an expiration date and is discarded when it expires.

Another optimization is a limitation on the rate at which floods are sent by a node. Since nodes will put the job in their pending job queue if they are not able to start computation, there is a chance some nodes which do not join the computation immediately might do so at a later time. For this reason a node always waits for a certain time before performing another flood. Determining the correct value for this timeout is a problem. If the timeout is too long it will take too long to reach the required number of nodes. If it is too short the radius of the flood will increase too quickly, and will reach nodes far away on the network, while more nearby nodes may still be available.

The solution for the problem of determining the timeout between floods lies in using a flexible timeout. At first, a short timeout is used to reach a reasonable number of nodes quickly. As more floods are send, the timeout increases. The progressively slower rate of sending floods limits the radius and thus the distance to the reached nodes. By default the timeout starts at 1 second and is doubled each time a flood is send, but these values are adjustable if needed.

The last optimization present in the scheduling system of Zorilla is allowing computations to start even when the specified number of nodes is not available yet. Although not all requested resources are there, the limited number of nodes available might still be able to start the computation, if the application supports malleability. This can be done for instance by using JEL (See Chapter 5).

4.3 Experiments

As said, Zorilla uses the locality-awareness of the overlay network to achieve locality awareness of its scheduling. In principle, any locality-aware overlay can be

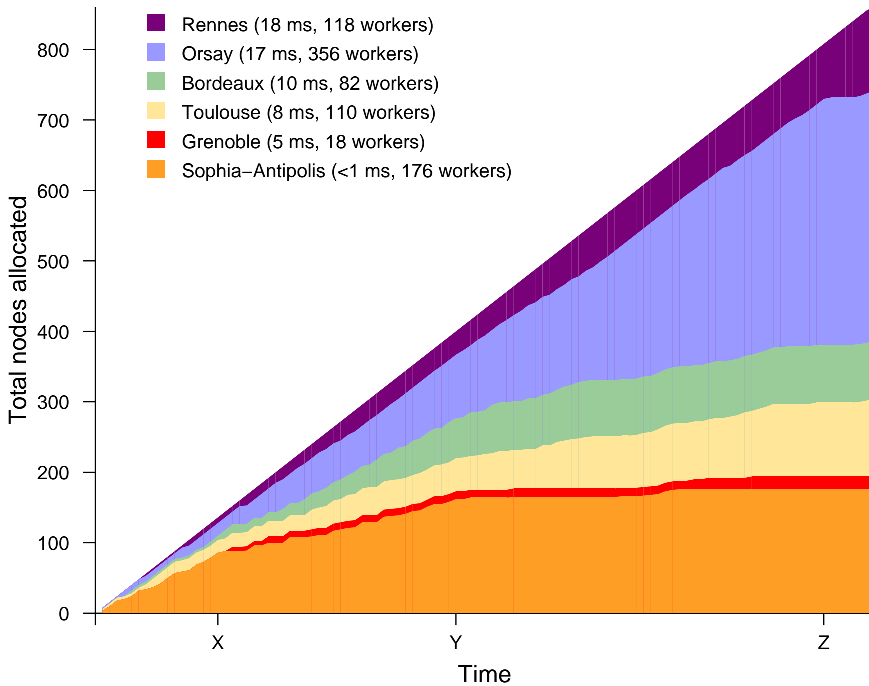


Figure 4.3: Distribution of workers across sites when submitting an increasing number of jobs from a single node.

used for this task. Instead of using the ARRG-based version of Zorilla described in Chapter 2, the experiments done in this chapter where performed with an older version of Zorilla which uses the overlay network of Bamboo [69] to implement flood-scheduling. Bamboo enables nodes to communicate with any other node in the system, and allows new nodes to join the system. It also detects node failures, and automatically adjusts the overlay network to compensate. The Bamboo overlay network is also locality aware.

We tested Zorilla on the Grid5000 [9] testbed. Grid5000 is a system with over a thousand processors distributed over eight sites across France. Most clusters are made up of dual processor AMD Opteron machines connected by a gigabit Ethernet but different processors such as Intel Xeons and IBM PowerPC processors and high speed networks such as Myrinet are also present in the system. Latencies between different sites are 5-20 milliseconds.

To test the scheduling algorithm, we deployed Zorilla on a total of 430 machines on six different sites of Grid5000. The maximum number of workers in this system is 860, as each machine again has two workers. The number of workers on each clusters varied, ranging from 18 in Grenoble, to 356 in Orsay.

After deployment we continuously submitted jobs to the system at one specific node. Each job requested eight workers. As the jobs were set to run forever, the nodes allocated in one job could not be used again in another. Figure 4.3 shows the distribution of nodes over the different sites over time. The sites are sorted in distance to the submitting node, located at the Sophia-Antipolis site. The graph shows that the clusters closer to the submitting job are allocated before clusters which are located further away.

Up until time X, nodes are almost exclusively allocated from the Sophia-Antipolis cluster. From time X to time Y, nodes out of the big cluster in Orsay are allocated as well. Between time Y and time Z, almost all nodes of the Sophia-Antipolis cluster are already allocated, and newly allocated nodes mostly are from Orsay. Finally, from time Z on, the Orsay cluster is also virtually allocated, and nodes of the Rennes cluster are used. The graph shows that the flood scheduler schedules the nodes in a locality-aware fashion.

Although nodes close to the submitting node are *generally* used before far away nodes, the scheduling decisions made are not perfect. Workers are sometimes started on nodes that are not the closest free node. These allocations are a result of the structure of the Bamboo overlay network used in this version of Zorilla. Zorilla floods messages by sending a message to all neighbors of a node. Although Bamboo takes distance into account when selecting neighbors, it cannot *only* use neighbors that are close as this would compromise the functioning of the overlay network. For example, if a network consists of two clusters connected by a very high latency link, selecting only the closest nodes as neighbors would mean no node will ever connect to a node in the other cluster, leading to a partitioning of the network. For this reason the locality awareness of Bamboo cannot be perfect. A node will always have some connections to far away nodes to ensure the functioning of the overlay network.

As a direct result of the measurements in this chapter, we decided to design and implement an overlay network specifically designed to support the locality-aware flood scheduler of Zorilla, eventually leading to the development of ARRG (see Chapter 3), and usage of ARRG in Zorilla. Chapter 2 describes this improved system. Although we do not include any measurements of this later version of Zorilla here, initial findings indicate locality-awareness has indeed improved over the version tested in this chapter. For measurements done with Zorilla *including* ARRG, see Chapter 6.

4.4 Conclusions

In this chapter we have studied the scheduling of supercomputing applications in P2P environments. We introduced *flood scheduling*: a scheduling algorithm based on flooding messages over a P2P network. Flood scheduling is fully decentralized, supports co-allocation and has good fault-tolerance properties.

Using Zorilla, we were able to deploy and run a parallel divide-and-conquer application on 671 processors simultaneously, solving the N-Queens 22 problem in

35 minutes. We used six clusters of the Grid5000 [9] system, located at sites across France. This large scale experiment on a real grid showed that flood scheduling is able to effectively allocate resources to jobs in a locality-aware way across entire grids. For more experiments done with Zorilla, including flood scheduling, see Chapter 6.

Flood scheduling depends on the locality-awareness of the P2P network used. Although the Bamboo based version of Zorilla evaluated in this chapter yields satisfactory results, we expect further improvements from our ARRG based Zorilla.

Chapter 5

JEL: Unified Resource Tracking*

Traditionally, supercomputers and clusters are the main computing environments[†] for running high performance parallel applications. When a job is scheduled and started, it is assigned a number of machines, which it uses until the computation is finished. Thus, the set of resources used for an application in these environments is generally fixed.

In recent years, parallel applications are also run on large-scale grid systems [35], where a single parallel application may use resources across multiple sites simultaneously. Recently, desktop grids [78], and clouds [29], systems are also used for running parallel and distributed applications. Our Zorilla peer-to-peer (P2P) middleware introduced in Chapter 2 allows users to run high-performance distributed supercomputing applications on very dynamic systems with less effort than previously possible. In all such environments, resources may become unavailable at any time, for instance when machines fail or reservations end. Also, new resources may become available after the application has started. As a result, it is no longer possible to assume that resource allocation is static.

To run successfully in these increasingly dynamic environments, applications must be able to handle the inherent problems of these environments. Specifically, applications must incorporate both *malleability* [61], the capability to handle changes in the resources used during a computation, and *fault tolerance*, the capability to continue a computation despite failures. Without mechanisms for malleability and fault-tolerance, the reliable execution of applications on dynamic systems is hard, if not impossible.

---

*This chapter is based on our paper published in *Proceedings of the 17th IEEE International Symposium on High-Performance Distributed Computing (HPDC 2008)* [26] and our paper in *Concurrency and Computation: Practice and Experience* [27].

†We will use the term *environment* for collections of compute resources such as supercomputers, clusters, grids, desktop grids, clouds, peer-to-peer systems, etc., throughout this chapter.

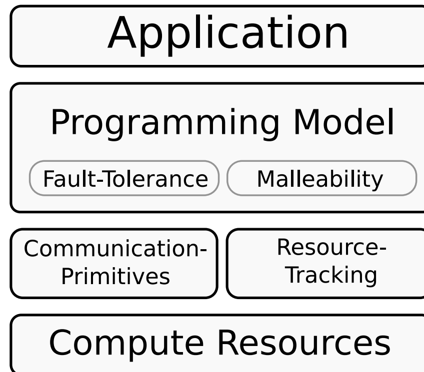


Figure 5.1: Abstract system hierarchy with resource tracking and communication primitives being the central low-level primitives for developing fault-tolerant and malleable programming models and applications.

A first step in creating a malleable and fault-tolerant system is to obtain an accurate and up-to-date view of the resources participating in a computation, and what roles they have. We therefore require some form of signaling whenever changes to the resource set occur. This information can then be used by the application itself, or by the runtime system (RTS) of the application's programming model, to react to these changes. In this chapter we refer to such functionality as *resource tracking*.

An important question is at what level in the software hierarchy resource tracking should be implemented. One option is to implement it in the application itself. However, this requires each application to implement resource tracking separately. Another option is to implement resource tracking in the RTS of the programming model of the application. Unfortunately, this still requires implementing resource tracking for each programming model separately. Also, an implementation of resource tracking designed for use on a grid will be very different from one designed for a P2P environment. Therefore, the resource tracking functionality of each programming model will have to be implemented for each target environment as well. This situation is clearly not ideal.

Based on the observations above, we argue that resource tracking must be an integral part of a system designed for dynamic environments, *in addition* to the low level communication primitives already present in such systems [57, 60, 66]. Figure 5.1 shows the position of resource tracking in a software hierarchy. There, a programming models' RTS uses low-level resource tracking functionality to implement the higher level fault-tolerance and malleability required. This way, resource tracking (indirectly) allows applications to run reliably and efficiently on dynamic systems such as grids and clouds.

In this chapter we propose a general solution for resource tracking: the Join-Elect-Leave (JEL) model. JEL acts as an intermediate layer between programming

models and the environment they run on. Since different environments have different characteristics, using a single implementation is impractical, if not impossible. Instead, several implementations of the JEL API are required, each optimized for a particular environment.

We have implemented JEL efficiently on clusters, grids, P2P systems, and clouds. These different JEL implementations can be used transparently by a range of programming models, in effect providing *unified* resource tracking for parallel and distributed applications across environments.

The contributions of this chapter are as follows.

- We show the need for unified resource tracking models in dynamic environments such as grids, P2P systems, and clouds, and explore the requirements of these models.

- We define JEL: a unified model for tracking resources in dynamic environments. JEL is explicitly designed to be simple yet powerful, scalable, and flexible. The flexibility of JEL allows it to support parallel as well as distributed programming models.

- We show how JEL suits the resource tracking requirements of several programming models. We have implemented 7 different programming models using JEL, ranging from traditional models such as MPI-1 (in the form of MPJ [10]), to Satin [61], a high level divide-and-conquer grid programming model that transparently supports malleability and fault-tolerance.

- We show that JEL is able to function on a range of environments by discussing multiple implementations of JEL. These include a centralized solution for relatively stable environments such as clusters and grids, and a fault-tolerant P2P implementation. In part, these implementations are based on well-known techniques of information dissemination in distributed systems. Notably, JEL can be implemented efficiently in different environments, due to the presence of multiple consistency models.

In previous work we presented the *Ibis Portability Layer (IPL)* [60], a communication library specifically targeted at dynamic systems such as grids. We augmented the IPL with our JEL resource tracking model, leading to a software system which can efficiently run applications on clusters, grids, P2P systems, and clouds.

Using a resource tracking model such as JEL is vital when running applications on our Zorilla middleware, as an instant cloud is inherently dynamic. Therefore, Zorilla explicitly supports JEL (see Chapter 2). The combination of Zorilla and JEL also makes the resulting system highly suited for running malleable applications. The flood-scheduler used in Zorilla (see Chapter 4) explicitly supports malleability, and add or remove resources after the application had been started. For an example of such a scenario, where Zorilla is used in an adaptive application, see [85].

One implementation of our JEL resource tracking model is based on the ARRG gossiping algorithm as discussed in Chapter 3 (see Section 5.4.2). Large-scale experiments on a real-world distributed system with JEL are shown in Chapter 6.

This chapter is structured as follows. Section 5.1 discusses the requirements of a general resource tracking model. Section 5.2 shows one possible model fulfilling these requirements: our Join-Elect-Leave (JEL) model. Section 5.3 explains how JEL is used in several programming models. In Section 5.4 we discuss a (partially) centralized and a fully distributed implementation of JEL. Section 5.5 compares the performance of our implementations, and shows the applicability of JEL in real-world scenarios. As a worst case, we show that JEL is able to support even short-lived applications on large numbers of machines. Section 5.6 discusses related work. Finally, we conclude in Section 5.7.

5.1 Requirements of Resource Tracking models

In this section we explore the requirements of resource tracking in a dynamic system. As said, resource tracking functionality can best be provided at a level between programming models and the computational environment (see Figure 5.1). A programming models' RTS uses this functionality to implement fault-tolerance and malleability. This naturally leads to two sets of requirements for resource tracking: requirements imposed by the programming model above, and requirements resulting from the environment below. We will discuss each in turn.

5.1.1 Programming Model Requirements

For any resource tracking model to be generally applicable, it needs to support multiple programming models, including both parallel and distributed models. Below is a list of requirements covering the needs of most, if not all, parallel and distributed programming models.

List of participants: The most obvious requirement of a resource tracking model is the capability to build up a list of all computational resources participating in a computation. When communicating and cooperating with other participants of a computation, one must know who these other participants are.

Reporting of changes: Simply building a list of participants at start-up is not sufficient. Since resources may be added or removed during the runtime of a computation, a method for updating the current list of participants is also required. This can be done for instance by signaling the programming models' RTS whenever a change occurs.

Fault detection: Not all resources are removed gracefully. Machines may crash, and processes may be terminated unannounced by a scheduling system. For this reason, the resource tracking model also needs to include a failure detection and reporting mechanism.

Role Selection: It is often necessary to select a leader from a set of resources for a specific task. For instance, a primary object may have to be selected in primary-copy replication, or a master may have to be selected in a master-worker application. Therefore, next to keeping track of which resources are present in a computation, a method for determining the roles of these resources is also required.

5.1.2 Environment Requirements

Next to supporting multiple programming models, a generally applicable resource tracking model must also support multiple environments, including clusters, grids, clouds, and P2P systems. We now determine the requirements resulting from the environment in which a resource tracking model is used.

Small, Simple Interface: Different environments may have wildly different characteristics. On cluster systems, the set of resources is usually constant. On grids and clouds resource changes occur, albeit at a low rate. P2P systems, however, are known for their high rate of change. Therefore, different (implementations of) algorithms are needed for efficient resource tracking on different environments. To facilitate the efficient re-targeting of a resource tracking model, its interface must be as small and simple as possible.

Flexible Quality of Service: Even with a small and simple interface, it may not be possible to implement all features of a resource tracking model efficiently on all environments with the same quality of service. For instance, *reliably* tracking *each and every* change to the set of resources in a small-scale cluster system is almost trivial, while in a large-scale P2P environment this is hard to implement efficiently, if possible at all. However, not all programming models require the full functionality of a resource tracking model. Therefore, a resource tracking model should include quality of service features. If the resource tracking model allows for a programming model to specify the required features and their quality of service, a suitable implementation could be selected at runtime. This flexibility would greatly increase the applicability of a resource tracking model.

5.2 The Join-Elect-Leave Model

We will now describe our resource tracking model: Join-Elect-Leave (JEL). JEL fulfills all stated requirements of a resource tracking model. As shown in Figure 5.1, JEL is located at the same layer of the software hierarchy as low-level communication primitives. Applications use a programming model, ideally with support for fault-tolerance and malleability. The programming model's RTS uses JEL for resource tracking, as well as a communication library. In this section we refer to programming models as *users* of JEL.

```
interface JEL {
    void init(Consistency electionConsistency,
              Consistency joinLeaveConsistency);
    void join(String poolName, Identifier identifier);
    void leave();
    void maybeDead(Identifier identifier);
    Identifier elect(String electionName);
    Identifier getElectionResult(String electionName);
}
//interface for notifications, called by JEL
interface JELNotifications {
    void joined(Identifier identifier);
    void left(Identifier identifier);
    void died(Identifier identifier);
}
```

Figure 5.2: JEL API (pseudocode, simplified).

Figure 5.2 shows the JEL API. Next to an initialization function, the API consists of two parts, *Joins and Leaves*, and *Elections*. Together, these fulfill the requirements of parallel and distributed programming models as stated in the previous section.

In general, each machine used in a computation initializes JEL once, and is tracked as a single entity. However, modern machines usually contain multiple processors and/or multiple compute cores per processor. In some cases, it is therefore useful to start multiple processes per machine for a single computation, which then need to be individually tracked. In this chapter, we therefore use the abstract term *node* to refer to a computational resource. Each node represents a single instance in a computation, be it an entire machine, or one processor of that machine.

JEL has been designed to work together with any communication library. The communication library is expected to create a unique identifier containing a contact address for each node in the system. JEL uses this address to identify nodes in the system, allowing a user to contact a node whenever JEL refers to it.

5.2.1 Joins and Leaves

In JEL, the concept of a *pool* is used to denote the collection of resources used in a computation. To keep track of exactly which nodes are participating in a pool, JEL supports *join* notifications. Users are being notified whenever a new node joins a pool. When a node joins a pool, it also is notified of all nodes already present in the pool via the same notifications, given using the *JELNotifications* interface. This is typically done using callbacks, although a polling mechanism can be used instead if callbacks are not supported by a programming language.

JEL also supports nodes leaving a computation, both gracefully and due to failures. If a node notifies JEL that it is leaving the computation, users of the remaining nodes in the pool receive a *leave* notification for this node. If a node does not leave gracefully, but crashes or is killed, the notification will consist of a

died message instead. Implementations of JEL try to detect failing nodes, but the user can also report suspected failures to JEL using the *maybeDead* function.

5.2.2 Elections

It is often necessary to select a leader node from a set of resources for a specific task. To select a single resource from a pool, JEL supports *Elections*. Each election has a unique name. Nodes can nominate themselves by calling the *elect* function with the name of the election as a parameter. The identifier of the winner will be returned. Using the *getElectionResult* function, nodes can retrieve the result without being a candidate.

Elections are not democratic. It is up to the JEL implementation to select a winner from the candidates. For instance, an implementation may simply select the first candidate as the winner. At the user level, all that is known is that *some* candidate will be chosen. When a winner of an election leaves or dies, JEL will automatically select a new winner from the remaining living candidates. This ensures that the election mechanism will function correctly in a malleable pool.

5.2.3 Consistency models

Together, join/leaves and elections fulfill all resource tracking requirements of fault-tolerant and malleable programming models as stated in Section 5.1.1. However, we also require our model to be applicable to a wide range of environments, from clusters to P2P systems. To this end, JEL supports several consistency models for the join/leave notifications and the elections. These can be selected independently when JEL is initialized using the *init* function. Joins/leaves or elections can also be turned off completely, if either part is not used. For examples of situations of when some parts of JEL remain unused, see Section 5.3.

Relaxing the consistency model allows JEL to be used on more dynamic systems such as P2P environments, where implementing strict consistency models cannot be done efficiently, if at all. For example, Section 5.4.2 describes a fully distributed implementation that is robust against failures, under a relaxed consistency model.

JEL offers two consistency models for joins and leaves. The *reliable* consistency model ensures that all notifications arrive in the same order on all nodes. Using reliable joins and leaves, a user can build up a list of all nodes in the pool. As an alternative, JEL also supports *unreliable* joins and leaves, where notifications are delivered on a best effort basis, and may arrive out of order, or not at all.

Similarly, JEL supports multiple consistency models for elections. If *uniform* elections are used, a *single* winner is guaranteed for each election, known at all nodes. Using the *non-uniform* model, an election is only guaranteed to converge to a single winner in unbounded time. The implementation of JEL will try to reach consensus on the winner of an election as soon as possible, but in a large system this may be time-consuming. Before a consensus is reached, different nodes may perceive different winners for a single election. Intuitively, this non-uniform

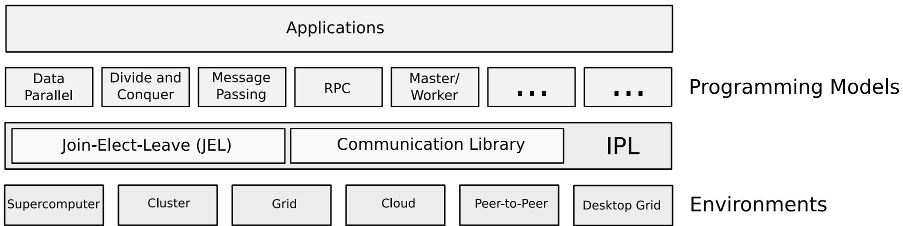


Figure 5.3: Position of JEL in the Ibis grid programming software stack.

Model	Joins and Leave	Elections
Master-Worker	-	Uniform
Divide-and-Conquer (elected master)	Unreliable	Uniform
Divide-and-Conquer (selected master)	Unreliable	Non-Uniform
Message Passing	Reliable	-

Table 5.1: Parts and consistency models of JEL used in the example programming models.

election has a very weak consistency. However, it is still useful in a number of situations (Section 5.3.2 shows an example).

5.3 Applicability of JEL

JEL has been specifically designed to cover the required functionality of a range of programming models found in distributed systems. We have implemented JEL in the *Ibis Portability Layer (IPL)* [60], the communication library of the Ibis project. Figure 5.3 shows the position of JEL in the software stack of the Ibis project. All programming models implemented in the Ibis project use JEL to track resources, notably:

- Satin [61], a divide-and-conquer model
- Java RMI, an object oriented RPC model [83]
- GMI [53], a group method invocation model
- MPJ [10], a Java binding for MPI-1
- RepMI [53], a replicated object model
- Maestro [5], a fault-tolerant and self optimizing dataflow model
- Jorus [5], a user-transparent parallel model for multimedia computing

As JEL is a generic model, it also supports other programming models. In addition to the models listed, we have implemented a number of prototype programming models, including data parallel, master-worker and Bulk Synchronous Parallel (BSP) models. Although our current JEL implementations are implemented using Java, the JEL model itself is not limited to this language. The foremost problem when porting JEL to other programming languages is the possible absence of a callback mechanism. This problem can be solved by using downcalls instead. In addition, parts of current JEL implementations could be reused, for instance by combining the server of the centralized implementation with a client written in another language.

We will now illustrate the expressiveness of JEL by discussing several models in more detail. These programming models use different parts and consistency models of JEL, see Table 5.1 for an overview.

5.3.1 Master-Worker

The first programming model we discuss is the *master-worker* [40] model, which requires a single node to be assigned as the *master*. Since the master controls the application, its identity must be made available to all other (*worker*) nodes. Depending on the application, the number of suitable candidates for the role of master may range from a single node to all participating nodes. For this selection, the master-worker model uses uniform elections.

Since workers do not communicate, the only information a worker needs in a master-worker model is the identity of the master node. So, in this model, joins and leaves are not needed, and can simply be switched off.

5.3.2 Divide-and-Conquer

The second programming model we discuss is divide-and-conquer. As an example of such a system we use Satin [61]. Satin is malleable, can handle failures, and hides many intricacies of the grid from the application programmer. It also completely hides which resources are used. Distribution and load balancing are performed automatically by using random work stealing between nodes. Satin is *cluster-aware*: it exploits the hierarchical nature of grids to optimize load balancing and data transfer. For instance, nodes prefer to steal work from nodes inside their local cluster, as opposed to from remote sites. The Satin programming model requires support from the resource tracking model for adding new nodes, as well as removing running nodes (either gracefully or due to a crash). Satin applies this information to re-execute subtasks if a processor crashes. Also, it dynamically schedules subtasks on new machines that become available during the computation, and it migrates subtasks if machines leave the computation.

Although Satin requires notifications whenever nodes join or leave the computation, these notifications do not need to be completely reliable, nor do they need to be ordered in any way. Satin uses the joins and leaves to build up a list of nodes in the pool. This list is then used to randomly select nodes to steal work from.

As long as each node has a reasonably up-to-date view of who is participating in the application, Satin will continue to work. When the information is out of date or incomplete, the random sampling will be skewed slightly, but in practice the negative impact on performance is small (see Section 5.5.4). Satin therefore uses the *unreliable* consistency of the join and leave notifications.

An election is used to select a special coordinator per cluster. These coordinators are used to optimize the distribution of fault tolerance related data in wide area systems. When multiple coordinators are present, more data will be transferred, which may lead to lower performance. Satin will still function correctly, however. Therefore, the election mechanism used to select the cluster coordinators does not necessarily have to return a unique result, meaning that the *non-uniform* elections of JEL can be used.

When an application is starting, Satin needs to select a master node that starts the main function of the application. This node can be explicitly specified by the user or application, or it can be automatically selected by Satin. The latter requires the *uniform* election mechanism of JEL. If the master node is specified in advance by the user, no election is needed for this functionality.

From the discussion above, we can conclude that the requirements of Satin differ depending on the circumstances. If the user has specified a master node, Satin requires *unreliable* join and leave notifications for the list of nodes, as well as *non-uniform* elections for electing cluster coordinators. If, on the other hand, a master node must be selected by Satin itself, *uniform* elections are an additional requirement.

5.3.3 Message Passing (MPI-1)

The last programming model we discuss is the Message Passing model, in this case represented by the commonly used MPI [57] system. MPI is widely used on clusters and even for multi-site runs on grid systems. We implemented a Java version of MPI-1, MPJ [10]. The MPI model assigns ranks to all nodes. Ranks are integers uniquely identifying a node, assigned from 0 up to the number of nodes in the pool. In addition, users can retrieve the total number of nodes in the system.

Joins and leaves with reliable consistency are guaranteed to arrive in the same order on all nodes. This allows MPI to build up a totally ordered list of nodes, by assigning rank 0 to the first node that joins the pool, rank 1 to the second, etc. Like the master-worker model, MPI does not require all functionality of JEL, as elections are not used.

MPI-1 has very limited support for changes of resources and failures. Applications using this model cannot handle changes to the resources such as nodes leaving or crashing. Using an MPI implemented on top of JEL will not fix this problem. However, some extensions to MPI are possible. For instance, MPI-2 supports new nodes joining the computation, Phoenix [77] adds supports for nodes leaving gracefully, and FT-MPI [33] allows the user to handle faults, by specifying the action to be taken when a node dies. All these extensions to MPI can be implemented using JEL for the required resource tracking capabilities.

5.4 JEL Implementations

It is impractical, if not impossible, to use the same implementation of JEL on clusters, grids, clouds, as well as P2P systems. As these different environments have different characteristics, there are different trade-offs in implementation design. We have explored several alternative designs, and discuss these in this section.

On cluster systems, resources used in a computation are mostly fixed, and do not change much over time. Therefore, our JEL implementation targeted at single cluster environments uses a relatively simple algorithm for tracking resources, based on a central coordinator. This ensures high performance and scalability, and the simple design leads to a more robust, less error prone implementation. This *central* implementation provides reliable joins and leaves and uniform elections. As this implementation uses a central coordinator for tracking resources, these stronger consistency models can be implemented without much effort.

On more dynamic systems such as grids, clouds and desktop grids, the simple implementation design used on clusters is not sufficient. As the number of machines in the system increases, so does the number of failures. Moreover, any change to the set of resources needs to be disseminated to a larger set of machines, possibly with high network latencies. Thus, these environments require a more *scalable* implementation of JEL. We used a number of techniques to decrease the effort required and amount of data transferred by the central coordinator, at the cost of an increased complexity of the implementation. As the resource tracking still uses a central coordinator, the stronger consistency models for joins, leaves and elections of JEL are still available.

Lastly, we implemented JEL on P2P environments. By definition, it is not possible to use centralized components in P2P systems. Therefore, our P2P implementation of JEL is fully distributed. Using Lamport clocks [49] and a distributed election algorithm [42] it is possible to implement strong consistency models in a fully distributed manner. However, these algorithms are prohibitively difficult to implement. Therefore, our P2P implementation only provides unreliable joins and leaves and non-uniform elections, making it extremely simple, robust and scalable. We leave implementing a P2P version of JEL with strong consistency models as future work.

As said, we have augmented our Ibis Portability Layer (IPL) [60] with JEL. The IPL is a low level message-based communication library implemented in Java, with support for streaming and efficient serialization of objects. All functionality of JEL is exported in the IPL's *Registry*. JEL is implemented in the IPL as a separate thread of the Java process. Notifications are passed to the programming models' RTS or application using a callback mechanism.

5.4.1 Centralized JEL Implementation

Our centralized JEL implementation uses a single server to keep track of the state of the pool. Using a centralized server makes it possible to implement stronger

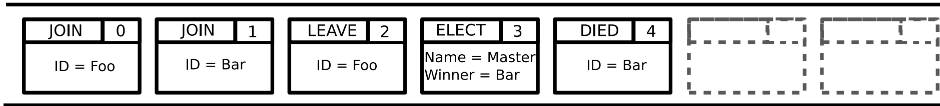


Figure 5.4: Example of an event stream.

consistency models. However, it also introduces a single point of failure, and a potential performance bottleneck.

The server has three functions. First, it handles requests of nodes participating in the computation. For example, a node may signal that it has joined the computation, is leaving, or is running for an election. By design, these requests require very little communication or computation.

Second, the server tracks the current resources in the pool. It keeps a list of all nodes and elections, and detects failed nodes. Our current implementation is based on a *leasing* mechanism, where nodes are required to periodically contact the server. If a node has had no contact with the server for a certain number of seconds, it sends a so-called *heartbeat* to the server. If it fails to do so, the server will try to connect to the node, to see if the node is still functional. If the server cannot reach the node, this node is declared *dead*, and removed from the pool.

Third, the server disseminates all changes of the state of the pool to the nodes. The nodes use these updates to generate join, leave, died, and election notifications for the application. If there are many nodes, the dissemination may require a significant amount of communication and lead to performance problems. To alleviate these problems we use a simple yet effective technique. Any changes to the state of the pool are mapped to *events*. These events have a *unique sequence number*, and are *totally ordered*. An event represents a node joining, a node leaving, a node dying, or an election result.

A series of state changes to a sequence of events can now be perceived as a *stream* of events. Dissemination of this stream can be optimized using well-known techniques such as broadcast trees or gossiping. Figure 5.4 shows an example of a stream of events. In this case, two nodes join, one leaves, one is elected master, and then dies. This stream of events thus results in an empty pool.

We have experimented with four different methods of disseminating the event stream: a simple serial send, serial send with peer bootstrap, a broadcast tree, and gossiping. The different mechanisms and their implementations are described below.

5.4.1.1 Serial Send

In our first dissemination technique, the central server forwards all events occurring in the pool to each node individually. Such a serial send approach is straightforward to implement, and is very robust. It may lead to performance problems though, as a large amount of data may have to be sent by the server. To optimize network usage, the server sends to multiple nodes concurrently.

In this implementation, a large part of the communication performed by the server consists of sending a list of all nodes to a new, joining node (the so-called *bootstrap* data). If many nodes join a computation at the same time, this may cause the server to become overloaded.

5.4.1.2 Peer Bootstrap

As an optimization of the serial send technique, we implemented *peer bootstrapping*, where joining nodes use other nodes (their *peers*) to obtain the necessary bootstrap data. When a node joins, the server sends it a small list of randomly chosen nodes in the pool. The joining node then tries to obtain the bootstrap data from the nodes in this list. If, for some reason, none of the nodes in the list can be reached, the joining node uses the server as a backup source of bootstrap data. This approach guarantees that the bootstrap process will succeed eventually.

5.4.1.3 Broadcast tree

A more efficient way of disseminating the stream of events from the server to all nodes is a broadcast tree. Broadcast trees limit the load on the server by using the nodes themselves to forward data. Broadcast trees also have disadvantages, as the tree itself is a distributed data structure that needs to be managed. This requires significant effort, and makes broadcast trees less robust than serial send.

Our broadcast implementation uses a binomial tree structure with the server as the root of the tree, which is also commonly used in MPI implementations [48]. To minimize the overhead of managing the tree, we use the data stream being broadcast to manage the tree. Since this stream includes totally ordered notifications of all joining and leaving nodes, we can use it to construct the broadcast tree at each node.

To increase the robustness of our broadcast implementation, we implemented *fallback* information dissemination. Periodically, the server directly connects to each node in the pool, and sends it any events it did not receive yet. This fallback mechanism guarantees the functioning of the system, regardless of the number, and type, of failures occurring. Also, it causes very little overhead if there are no failures.

5.4.1.4 Gossiping

A fourth alternative for disseminating the events of a pool to all its nodes is the use of *gossiping* techniques. Gossiping works on the basis of periodic information exchanges (gossips) between peers (nodes). Gossiping is robust, easy to implement and has low resource requirements. See Chapter 3 for more information on gossiping techniques.

In the gossiping dissemination, all nodes record the event stream. Periodically, a node contacts one of its peers. The event stream of those two nodes are then merged by sending any missing events from one peer to the other. To reduce memory usage old events are eventually purged from the system.

Although the nodes exchange events amongst themselves, the pool is still managed by the central server. The server still acts as a contact point for nodes that want to join, leave, or run for an election. Also the server creates all events, determines the ordering of events, detects failing nodes, etc.

To seed the pool of nodes with data, the server periodically contacts a random node, and sends it any new events. The nodes will then distribute these new events amongst themselves using gossiping. When the nodes gossip at a fixed interval, the events travel through the system at an exponential rate. The dissemination process thus requires a time that is logarithmically proportional to the pool size.

To speed up the dissemination of the events to all nodes, we implemented an *adaptive* gossiping interval at the server. Instead of waiting a fixed time between sending events to nodes, we calculate the interval based on the size of the pool by dividing the standard interval by the base 2 logarithm of the pool size. Thus, events are seeded at a speed proportionally to the pool size. The dissemination speed of events becomes approximately constant, at the expense of an increase in communication load on the server.

Since gossip targets are selected randomly, there is no guarantee that all nodes will receive all events. To ensure reliability, we use the same fallback dissemination technique we used in the broadcast tree implementation. Periodically, the server contacts all nodes and sends them any events they do not have.

5.4.2 Distributed JEL Implementation

Although the performance problems of the centralized implementation are largely solved by using broadcast trees and gossiping techniques, the server component is still a central point of failure, and not suitable for usage in P2P systems. As an alternative, we created a fully distributed implementation of JEL using P2P techniques. It has no central components, so failures of individual nodes do not lead to a failure of the entire system.

Our implementation is based on our ARRG gossiping algorithm (see Chapter 3). ARRG is resilient against failures, and can handle network connectivity problems such as firewalls and NATs. Each node in the system has a unique identifier in the form of a UUID [50], which is generated locally at startup. ARRG needs the address of an existing node at startup to bootstrap, so this must be provided. This address is used as an initial contact point in the pool. ARRG provides a so-called *peer sampling service* [46], guaranteeing a random sampling of the entire pool even if failures and network problems occur.

On top of ARRG, we use another gossiping algorithm to exchange data on nodes and elections. Periodically, a node connects to a random node (provided by ARRG) and exchanges information on other nodes and elections. It sends a random subset of the nodes and elections it knows and includes information on itself. It then receives a number of members and elections from the peer node, and merges these with its own state. Over time, nodes build up a list of nodes and elections in the pool.

If a node wants to leave the computation, it sends out this information to a number of nodes in the system. Eventually, this information will reach all nodes. Since a crashed node cannot send a notification to the other nodes indicating it has died, a distributed failure detection mechanism is needed.

The failure detection mechanism uses a *witness* system. A timeout is kept in every entry on a node, indicating the last time this node has successfully been contacted. Whenever the timeout expires, a node is suspected of having died. Nodes with expired entries in their node list try to contact these suspects. If this fails, they add themselves as a witness to this node's demise. The witness list is part of the gossiped information. If a sufficient number of nodes declare that a node has died, it is pronounced dead.

Besides joins and leaves, the distributed implementation also supports elections. Because of the difficulties of implementing distributed election algorithms [42], and the lack of guarantees even when using the more advanced algorithms, we only support the non-uniform election consistency model. In this model, an election converges to a single winner. Before that time, nodes may not agree on the winner of that election.

Election results are gossiped. When a node needs the result of a unknown election, it simply declares itself as the winner. If a conflict arises when merging two different election results, one of the two winners is selected deterministically (the node with the numerically lowest UUID wins). Over time, only a single winner remains in the system.

As a consequence of the aforementioned design, the distributed implementation of JEL is fault tolerant in many aspects. First, the extensive use of gossiping techniques inherently leads to fault tolerance. The ARRG protocol adds further tolerance against failures, for example by using a fallback cache containing previously successful contacts [24]. Most importantly, the distributed implementation lacks *any* centralized components, providing fully distributed implementations of all required functionality instead.

5.5 Evaluation

To evaluate the performance and scalability of our JEL implementations, we performed several experiments. These include low-level and application-level tests on multiple environments. In particular, we want to assess how much performance is sacrificed to gain the robustness of a fully distributed implementation, as we expect this implementation to have the lowest performance. Exact quantification of performance differences between implementations, however, is hard — if not impossible. As shown below, performance results are highly dependent on the characteristics of the underlying hardware. Furthermore, the impact on application performance, in turn, is dependent on the programming model used. For example, MPI can not proceed until all nodes have joined, while Satin starts as soon as a resource is available. All experiments were performed multiple times. Numbers shown are taken from a single representative experiment. Experiments

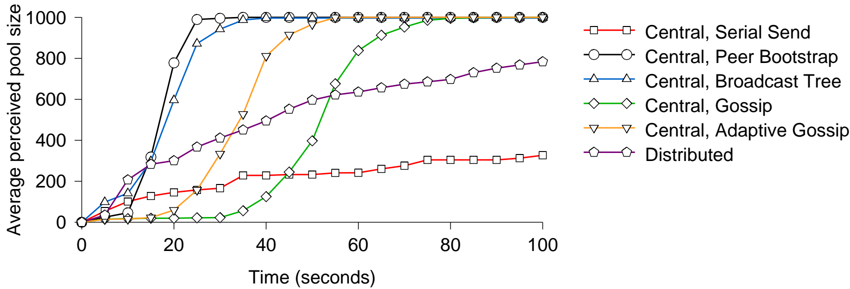


Figure 5.5: 1000 nodes Join test (DAS-2).

described in this chapter are all small- to medium-scale. For large-scale experiments, see Chapter 6.

5.5.1 Low level benchmark: Join test

The first experiment is a low-level stress test using a large number of nodes. We ran the experiment on two different clusters. The purpose of the experiment is to determine the performance of our JEL implementations under different network conditions. In the experiment, all nodes join a single pool and, after a predetermined time, leave again. As a performance metric, we use the *average perceived pool size*. To determine this metric, we keep track of the pool size at all nodes. Ideally, this number is equal to the actual pool size. However, if a node has not received all notifications, the perceived pool size will be smaller. We then calculate the average perceived pool size over all nodes in the system. The average is expected to increase over time, eventually becoming equal to the actual pool size. This indicates that all nodes have received all notifications. The shorter the stabilization time, the better.

This experiment was done on our DAS-2 [19] and DAS-3 [20] clusters. The DAS-2 cluster consists of 72 dual processor Pentium III machines, with 2Gb Myrinet interconnect. The DAS-3 cluster consists of 85 dual-CPU dual-core Opteron machines, with 10Gb Myrinet.

Since neither the DAS-2 nor DAS-3 have a sufficiently large number of machines to stress test our implementation, we started multiple nodes per machine. As neither our JEL implementations or the benchmark are CPU bound, the sharing of CPU resources does not influence our measurements. The nodes do share the network bandwidth though. However, all implementations of JEL are affected equally, so the relative results of all tested implementations remain valid. The server of the centralized implementation of JEL is started on the front-end machine of the cluster.

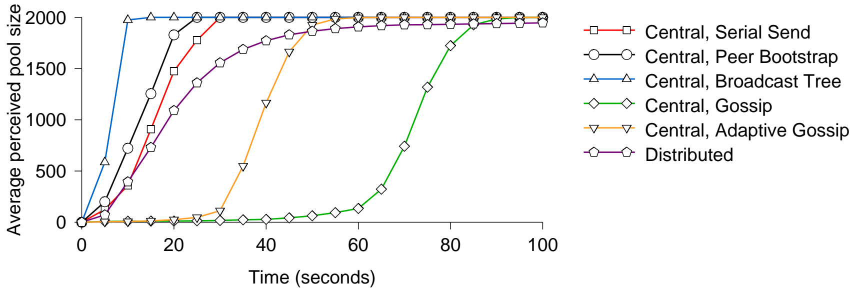


Figure 5.6: 2000 nodes Join test (DAS-3).

5.5.1.1 DAS-2

Figure 5.5 shows the performance of JEL on the DAS-2 system. We started 10 nodes per processor core on 50 dual processor machines, for a total of 1000 nodes. Due to the sharing of network resources, all nodes, as well as the frontend running the server, have an effective bandwidth of about 100Mbit/s.

For convenience, we only show the first 100 seconds of the experiment, when all nodes are joining. The graph shows that the serial send dissemination suffers from a lack of network bandwidth, and is the lowest performing implementation.

The peer bootstrap and broadcast tree techniques perform equally well on this system. This is not surprising, as the broadcast tree and peer bootstrap techniques utilize all nodes to increase throughput. As the graph shows, adaptive gossip dissemination is faster than the normal central gossip version, as it adapts its speed to the pool size.

While not shown in the graph, the fully distributed implementation is also converging to the size of the pool, albeit slower than most versions of the centralized implementation. The slow speed is caused by an overload of the bootstrap service, which receives 1000 gossip requests within a few milliseconds when all the nodes start. This is an artifact of this artificial test that causes all the nodes to start simultaneously. In a P2P environment this is unlikely to occur. Multiple instances of the bootstrap service would solve this problem. Still, the performance of the distributed implementation is acceptable, especially considering the high robustness of this implementation.

5.5.1.2 DAS-3

Next, we examine the performance of the same benchmark on the newer DAS-3 system (see Figure 5.6). As a faster network is available on this machine, congestion of the network is less likely. Since the DAS-3 cluster has more processor cores, we increased the number of nodes to 2000, resulting in 250Mbit/s of bandwidth per node. The frontend of our DAS-3 cluster has 10Gbit/s of bandwidth. Performance on the DAS-3 increases significantly compared to the DAS-2, mostly because of

Implementation	Dissemination	Server (MB)	Node Average (MB)
Central	Serial Send	1521.47	0.76
	Peer Bootstrap	677.23	0.45
	Broadcast Tree	5.57	1.32
	Gossip	9.83	0.49
	Adaptive Gossip	40.36	0.57
Distributed	Gossip	n.a.	25.37

Table 5.2: Total data transferred in Join test with 2000 nodes on the DAS-3.

the faster network. The serial send and gossip techniques no longer suffer from network congestion at the server or bootstrap service. As a result, performance increases dramatically for both. Also, the graph shows that the performance of the broadcast tree is now significantly better than any other dissemination technique.

Performance of the central implementation with gossiping is influenced by the larger size of the pool. It takes considerably longer to disseminate the information to all nodes. As before, the adaptive gossiping manages to adapt, and reaches the total pool size significantly faster.

From our low level benchmark on both the DAS-2 and DAS-3 we conclude that it is possible to implement JEL such that it is able to scale to a large number of nodes. Also, a number of different implementation designs are possible for JEL, all leading to reasonable performance.

5.5.2 Network bandwidth usage

To investigate the cost of using JEL, we recorded the total data transferred by both the server and the clients in the previous experiment. Table 5.2 shows the total traffic generated by the experiment on DAS-3, after all the nodes have joined and left the pool.

Using the serial send version, the server transferred over 1500 MB in the 10 minute experiment. Using peer bootstrap already halves the traffic needed at the server. However, the broadcast tree dissemination uses less than 5 MB of server traffic to accomplish the same result. It does this by using the nodes of the system, leading to a slightly higher traffic at the nodes (1.32 MB instead of 0.76 MB).

From this experiment we conclude that the dissemination techniques significantly increase the scalability of our implementation. Also, the broadcast tree implementation is very suited for low bandwidth environments. For the distributed implementation, the average traffic per node is 25 MB, an acceptable cost for having a fully distributed implementation.

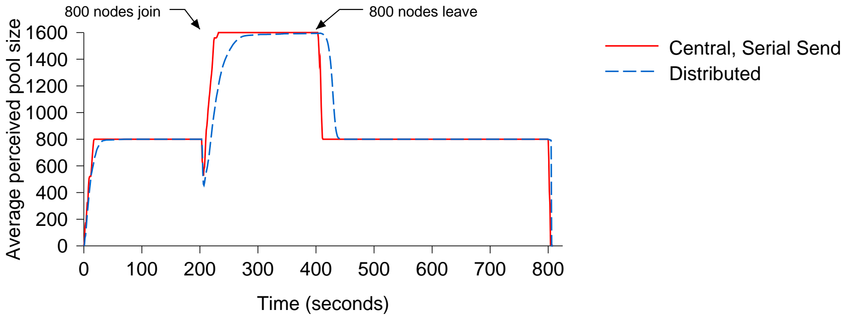


Figure 5.7: Join/Leave test run on 4 clusters across the DAS-3 grid. Half of the nodes only start after 200 seconds, and leave after 400 seconds.

5.5.3 Low level benchmark in a dynamic environment

We now test the performance of JEL in a dynamic environment, namely the DAS-3 grid. Besides the cluster at the VU used in the previous tests, the DAS-3 system consists of 4 more clusters across the Netherlands. For this test we started our Join benchmark on two clusters (800 nodes), and add two clusters later, for a total of 1600 nodes. Finally, two clusters also leave, either gracefully, or by *crashing*.

Results of the test when the nodes leave gracefully are shown in Figure 5.7. We tested both the central implementation of JEL and the distributed implementation. For the central implementation we have selected the serial send dissemination technique, which performs average on DAS-3 (see Figure 5.6). On the scale of the graph of Figure 5.7 results obtained for the other techniques are indistinguishable.

Figure 5.7 shows that both implementations are able to track the entire pool. As said, the pool size starts at 800 nodes, and increases to 1600 nodes 200 seconds into the experiment. The *dip* in the graph at 200 seconds is an artifact of the metric used: At the moment 800 extra nodes are started, these nodes have a perceived pool size of 0. Thus, the average over all nodes in the pool halves. As in the previous test, the central implementation is faster than the distributed implementation. After 400 seconds, two of the four clusters (800 of the 1600 nodes) leave the pool. The graph shows that JEL correctly handles nodes leaving, with both implementations processing the leaves shortly.

As said, we also tested with the nodes crashing by forcibly terminating the node's process. The results can be seen in Figure 5.8. When nodes crash instead of leaving, it takes longer for JEL to detect these nodes have died. This delay is due to the timeout mechanism in both implementations. A node is only declared dead if it cannot be reached for a certain time (a configuration property of the implementations, in this instance set to 120 seconds). Thus, nodes are declared dead with a delay after crashing. The central implementation of JEL has a slightly longer delay, as it tries to contact the faulty nodes one more time after the timeout

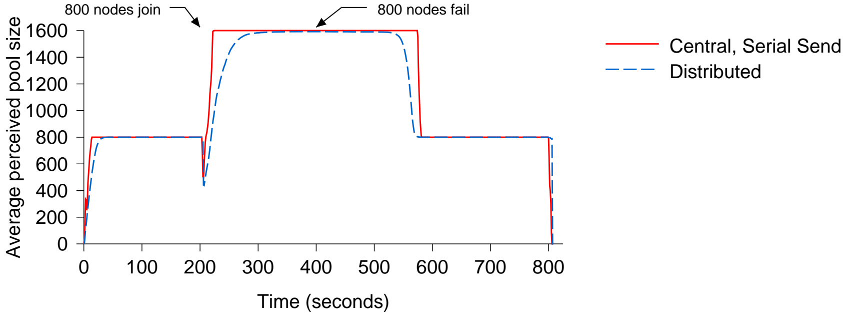


Figure 5.8: Join/Fail test run on 4 clusters across the DAS-3 grid. Half of the nodes only start after 200 seconds, and crash after 400 seconds.

Implementation	Dissemination	Run time Small	Run time Large	Join Time
Central	Serial Send	71.7	408.0	18.2
	Peer Bootstrap	70.5	406.1	17.2
	Broadcast Tree	66.4	402.9	10.6
	Gossip	67.7	426.6	14.6
	Adaptive Gossip	67.5	426.4	11.1
Distributed	Gossip	82.3	462.4	14.1

Table 5.3: Gene sequencing application on 256 cores of the DAS-3. Listed are total runtime (in seconds) of the application for two problem sizes and time (in seconds) until all nodes have joined fully (average perceived pool size is equal to the actual pool size). Runtime includes the join time.

expires. From this benchmark we conclude that JEL is able to function well in dynamic systems, with both leaving and failing nodes.

5.5.4 Satin Gene Sequencing Application

To test the performance of our JEL implementations in a real world setting, we used 256 cores of our DAS-3 cluster to run a gene sequencing application implemented in Satin [61]. Pairwise sequence alignment is a bioinformatics application where DNA sequences are compared with each other to identify similarities and differences. We run a large number of instances of the well-known Smith-Waterman [72] algorithm in parallel using Satin's divide-and-conquer programming style. The resulting application achieves excellent performance (93 %efficiency on 256 processors).

Table 5.3 lists the performance of the application for various JEL implementations, and two different problem sizes. We specifically chose to include a small

problem on a large number of cores to show that our JEL implementations are also suitable for short-running applications where the overhead of resource tracking is relatively large. In this very small problem, the application only ran for little over a minute. The table shows similar performance for all versions of JEL. Moreover, the relative difference is even smaller in the large problem size. An exception are the implementations based on gossiping techniques. The periodic gossiping causes a small but constant amount of network traffic. Unfortunately, the load balancing mechanism of Satin is very sensitive to this increase in network load. Though the distributed implementation lacks the guaranteed delivery of notifications present in the central implementation, Satin is able to perform the gene sequencing calculations with only minor delay. This is an important result, given Satin's transparent support for malleability and fault-tolerance, as explained in Section 5.3.2.

To give an impression of the overhead caused by JEL, we also list the *join time*, the amount of time from the start of the application it takes for the average perceived pool size to reach the actual pool size, i.e. the time JEL needs to notify all nodes of all joins. The join time of an application is independent of the runtime of the application, and mainly influenced by the number of nodes, JEL implementation, and resources used. Therefore, we only list the join time once, for both problem sizes. The performance of the various JEL implementations is in line with the low-level benchmark results, with the broadcast tree implementation being the fastest. Our gene sequencing experiment shows that our model and implementations are able to handle even these short running applications.

5.6 Related Work

Other projects have investigated supporting malleability and fault tolerance in various environments, and resource tracking in these systems. However, most of these projects focus on a single programming model, and a single target environment.

One area of active research for supporting applications on more dynamic environments is the MPI standard. As said, the MPI-1 standard does not have support for nodes joining or leaving the computations. To alleviate this problem the follow-up MPI-2 [57] standard also supports changes to the nodes in a system. A process may *spawn* new instances of itself, or connect to a different running set of MPI-2 processes. A very basic naming service is also available.

Although it is possible to add new processes to an MPI application, the resource tracking capabilities of MPI-2 are very limited by design and a MPI implementation is not required to handle node failures. Also, notifications of changes such as machines joining, leaving or crashing are not available. Thus, resource tracking of MPI-2 is very limited, unlike our generic JEL model.

One MPI derivative that does offer explicit support for fault-tolerance is FT-MPI [33]. FT-MPI extends the MPI standard with functionality to *recover* the MPI library and run-time environment after a node fails. In FT-MPI, an application can specify if failed nodes must be simply removed (leaving gaps in the ranks

used), replaced with new nodes, or if the groups and communicators of MPI must be *shrunk* so that no gap remains. Recovering the application must still be done by the application itself.

FT-MPI relies on the underlying system to detect failures and notify it of these failures. The reference implementation of FT-MPI uses HARNESS [7], a distributed virtual machine with explicit support for adding and removing hosts from the virtual machine, as well as failure detection. HARNESS shares much of the same goals as JEL, and is able to overcome many of the same problems JEL tries to solve. However, HARNESS focuses on a smaller set of applications and environments than JEL. HARNESS does not explicitly support distributed applications, as JEL does. Also, HARNESS does not offer the flexibility to select the concurrency model required by the application, hindering the possibility for more loosely coupled implementations of the model, such as the P2P implementation of JEL.

Other projects have investigated supporting dynamic systems. One example is Phoenix [77], where an MPI-like message passing model is used. This model is extended with support for *virtual nodes*, which are dynamically mapped to *physical nodes*, the actual machines in the system. GridSolve [87] is a system for using resources in a grid based on a *client-agent-server* architecture. The “View Synchrony” [3] shared data model also supports nodes joining, leaving and failing. Again, all these programming models focus on resource tracking for a single model, not the generic resource tracking functionality offered by JEL. All models mentioned can be implemented using the functionality of JEL.

Although all our current JEL implementations use gossiping and broadcast trees as a means for information dissemination, other techniques exist. One example is the publish-subscribe model [30]. Despite the fact that information dissemination is an important part of JEL, our model offers much more functionality to provide a full solution for the resource tracking problem. Most importantly, further functionality includes the *active* creation and gathering of information regarding (local) changes in the resource set.

All current implementations of JEL are build from the ground up, with little external dependencies. However, JEL implementations could in principal interface with external systems, for instance Grid Information Services (GIS [17]). These systems can be used both for acquiring (monitoring) data, as well as disseminating the resulting information. One key difference between JEL and current monitoring systems is the fact that JEL tracks resources of *applications*, not *systems*. An application crashing usually does not cause the entire system to cease functioning. Sole reliance of system monitoring data will therefore not detect application-level errors.

5.7 Conclusions

With the transition from static cluster systems to dynamic environments such as grids, clusters, clouds, and P2P systems, fault-tolerance and malleability are

now essential features for applications running in these environments. This is especially so for an application running on an instant cloud as created by Zorilla (see Chapter 2), as these are inherently dynamic systems. A first step in creating a fault-tolerant and malleable system is *resource tracking*: the capability to track exactly which resources are part of a computation, and what roles they have. Resource tracking is an essential feature in any dynamic environment, and should be implemented on the same level of the software hierarchy as communication primitives.

In this chapter we presented JEL: a unified model for tracking resources. JEL is explicitly designed to be scalable and flexible. Although the JEL model is simple, it supports both traditional programming models such as MPI, and flexible grid oriented models like Satin. JEL allows programming models such as Satin to implement both malleability and fault-tolerance. With JEL as a common layer for resource tracking, the development of programming models is simplified considerably. In the Ibis project, we developed a number of programming models using JEL, and we continue to add models regularly.

JEL can be used on environments ranging from clusters to highly dynamic P2P environments. We described several implementations of JEL, including a centralized implementation that can be combined with decentralized dissemination techniques, resulting in high performance, yet with low resource usage at the central server. Furthermore, we described several dissemination techniques that can be used with JEL. These include a broadcast tree and gossiping based techniques. In addition, we showed that JEL can be implemented in a fully distributed manner, using the ARRG gossiping algorithm discussed in Chapter 3 as a basis. This distributed implementation efficiently supports flexible programming models such as Satin, and increases fault-tolerance compared to a centralized implementation.

JEL is especially useful in combination with the flood-scheduling present in Zorilla (see Chapter 4), as this scheduler explicitly supports malleability, and may thus add or remove resources after the application had been started. For an example of such a scenario, see [85].

There is no single resource tracking model implementation that serves all purposes perfectly. Depending on the circumstances and requirements of the programming model and application a different implementation is appropriate. In a reliable cluster environment, a centralized implementation performs best. If applications are run on low bandwidth networks, the broadcast tree dissemination technique has the benefit of using very little bandwidth. In a hostile environment, such as desktop grids or P2P systems, a fully distributed implementation is robust against failures. JEL explicitly supports different algorithms and implementations, making it applicable in a large number of environments.

We evaluated JEL in several real-world scenarios. The scenarios include starting 2000 instances of an application, and wide area tests with new machines joining, and resources failing. Further experiments are discussed in detail in the next chapter.

Chapter 6

Large Scale Experiments*

In this chapter we will perform experiments that show the feasibility of real-world distributed systems for high-performance computing. Also, our experiments will demonstrate the algorithms and techniques described in this thesis in a larger, more dynamic, system.

All experiments in this chapter are performed using *IbisDeploy*. IbisDeploy is an easy-to-use library for running applications on distributed resources, and includes an intuitive GUI (see Figure 6.1). IbisDeploy is designed specifically for deploying applications written in Java that use the IPL [60] to communicate. IbisDeploy automatically uploads program codes, libraries, and input files, and downloads output files when the application is done. It also deploys a Smart-Sockets [54] overlay network which is capable of overcoming many connectivity problems present in real-world distributed systems such as Firewalls and NATs.

By default, IbisDeploy uses the JavaGAT to implement the required functionality, and users manage resources manually, by specifying exactly which resources to use for every job. Optionally, IbisDeploy can use Zorilla to deploy and run applications instead. This has the benefit of using functionality present in Zorilla such as discovery of resources and co-allocation provided by the flood-scheduling mechanism, and automatic recovery of failures by allocating replacement resources.

Figure 6.2 shows an overview of the software developed for this thesis, as well as selected other software that is part of the Ibis project. On the left, Ibis-Deploy is shown using Zorilla (see Chapter 2) to deploy applications. Zorilla, in turn, uses the flood-scheduler described in Chapter 4 to schedule resources. The flood-scheduler uses an overlay based on the ARRG gossiping algorithm described in Chapter 3 to discover resources. As described in Chapter 2, Zorilla used the JavaGAT to deploy applications, and supports many different middlewares. In addition to deploying applications, Zorilla also offers support for the JEL resource tracking model, by creating and managing the necessary support processes.

---

*This chapter contains portions of papers published in *Concurrency and Computation: Practice and Experience* [27] and *IEEE Computer* [6], as well as our paper submitted for publication [28].

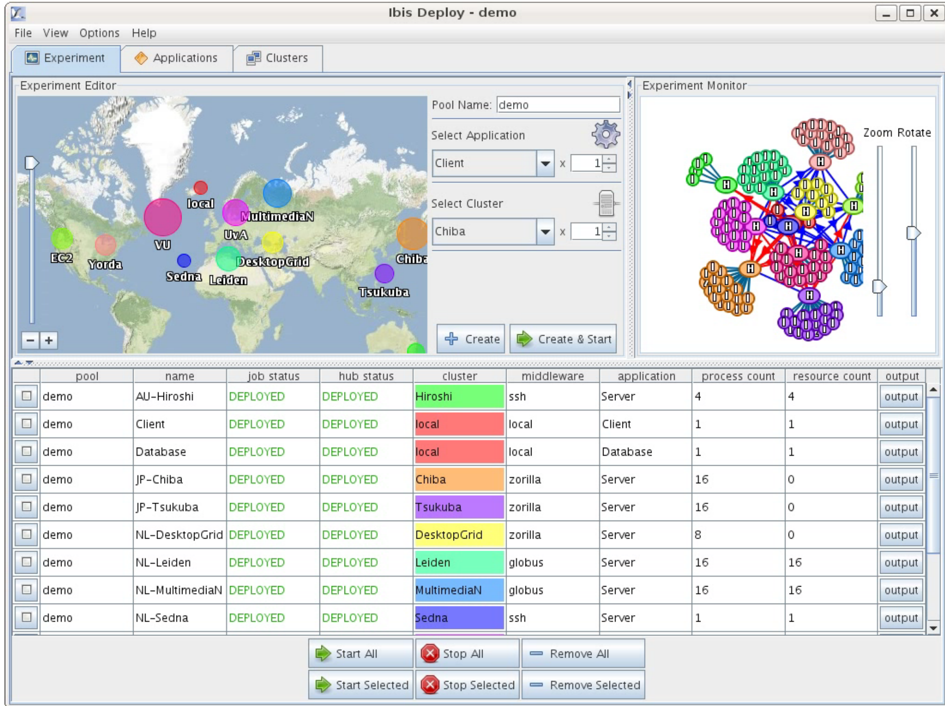


Figure 6.1: The IbisDeploy GUI allows loading of applications and resources (top middle), and keeping track of running processes (bottom half). Top left shows a world map of the locations of available resources; top right shows the SmartSockets network consisting of hubs and compute nodes.

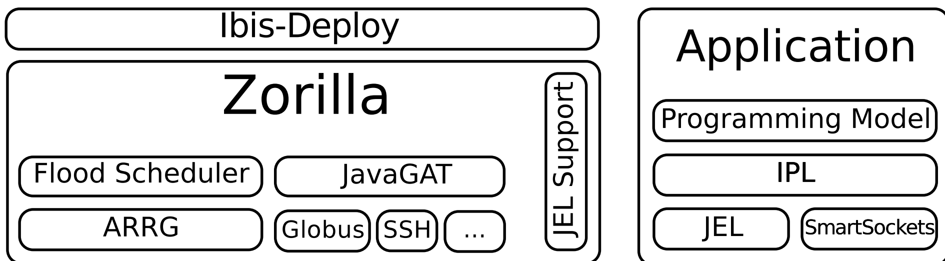


Figure 6.2: Simplified Ibis Software Stack. In this configuration, Ibis-Deploy is using Zorilla to run applications. Alternatively (and not shown), Ibis-Deploy can also use the JavaGAT directly.

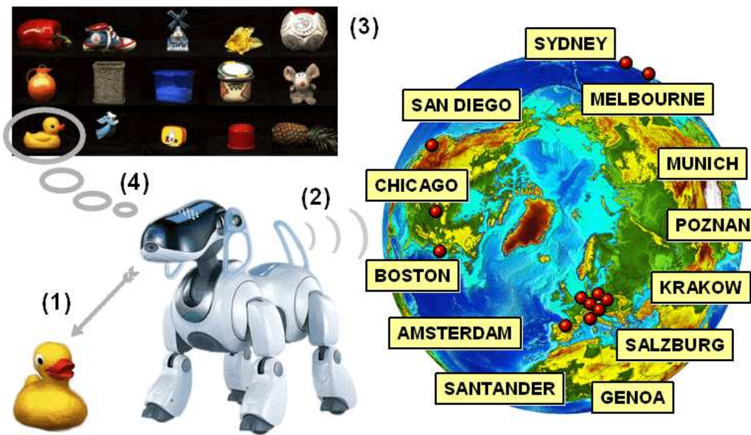


Figure 6.3: Multimedia content analysis application. (1) An object is held in front of a camera (for instance, part of a robot dog). (2) The system processes video frames on available resources. (3) The system compares the computed *feature vectors* to a database of previously learned objects. (4) The most similar object is returned. A video of this application is available at `http://www.cs.vu.nl/ibis/demos.html`.

The right side of Figure 6.2 shows a typical application deployed with Ibis-Deploy. Applications are written in some programming model, which in turn uses the IPL. The IPL then uses JEL to track resources, and SmartSockets to Communicate.

For our experiments, we use the Distributed ASCI Supercomputer 3 (DAS-3) [20], a five cluster distributed system located in The Netherlands. We also use the InTrigger [45] grid system in Japan, clusters in the USA and Australia, an Amazon EC2 [29] cloud system (USA, East Region), as well as a desktop grid and a single stand-alone machine (both Amsterdam, The Netherlands). Together, these machines comprise a real-world distributed system, as described in Chapter 1.

The rest of this chapter is organized as follows. In Section 6.1 we deploy a multimedia content analysis application on a large scale system, including both Zorilla and non-Zorilla resources. In Section 6.2 we deploy a high performance application (implementing a Go solver) on a world-wide scale, showing the viability of a real-world distributed system for compute intensive applications. In Section 6.3 we create an instant cloud from a large collection of heterogeneous resources using Zorilla. Finally, in Section 6.4 we give some indication of the actual real-world usability of our software and techniques, by describing a number of competitions we participated in.

6.1 Multimedia Content Analysis

As a first experiment we deploy a multimedia content analysis (MCA) application on our world-wide system. This application performs real-time recognition of every-day objects (see Figure 6.3). Images produced by a camera are processed by an advanced algorithm developed in the MultimediaN project [70]. This algorithm extracts *feature vectors* from the video data, which describe local properties like color and shape. To recognize an object, its feature vectors are compared to ones that have been stored earlier and annotated with a name. The name of the feature vector closest to the one currently seen is returned as a result. As this a compute intensive problem with (soft) real-time constrains, the analysis is performed on a large distributed system. In this case, the application is a data-parallel application, using a single thread per compute node. However, this application serves only as an example, and usage of upcoming techniques such as multi-core and many-core architectures such as GPUs could be integrated as well. This would allow for more compute-intensive MCA kernels.

A single video frame is processed by the data-parallel application (a *multimedia server*) running on a single site. Calculations over consecutive frames are distributed over different sites in a task-parallel manner. Because of this design, communication is mostly done within sites, and wide-area traffic is limited to input video frames.

The content analysis application is interactive, with a client application (See Figure 6.4) used to show the recognition result, as well as to teach the system new objects. The application also shows a list of available servers, and allows a user to explicitly control which servers are used. In line with this high level of control, selection of resources is done explicitly by the user in IbisDeploy. This experiment is supported by a video presentation, which is available at http://www.cs.vu.nl/ibis/demos.html. As shown in the video, we use IbisDeploy to start a client on a local machine, and to deploy four data-parallel multimedia servers, each on a different DAS-3 cluster (using 64 machines in total).

The use of a single multimedia server results in a processing rate of approximately 1.2 frames per second. The simultaneous use of 2 and 4 clusters leads to linear speedups at the client side with 2.5 and 5 frames/sec respectively. By adding additional clusters, the Amazon EC2 cloud, the local desktop grid, and the local stand-alone machine, we obtain a world-wide set of machines.

As said, resource selection is done explicitly by the user, and IbisDeploy uses the JavaGAT to run the application on each resource. Although Zorilla is not used on all resources in this experiment to create one, large, instant cloud, it is used on some of these resources. For instance, the desktop grid that is part of our world-wide system relies solely on Zorilla to manage resources, and the Amazon EC2 resources run Zorilla to distribute jobs and files.

To illustrate our fault-tolerance mechanisms, the video also shows an experiment where an entire multimedia server crashes. The JEL resource tracking system notices this crash, and signals the application. The client then removes the crashed server from the list of available servers. The application continues to run, as the

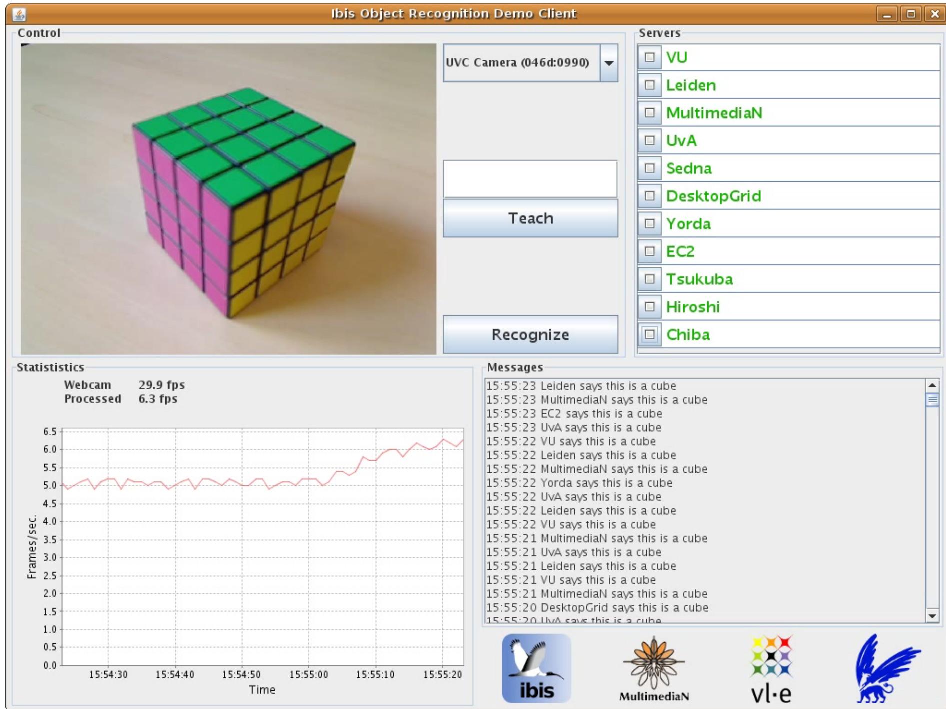


Figure 6.4: Screenshot of the Multimedia Content Analysis application. The top half of the window consists of the video stream (left), and interface for teaching new objects and recognizing object(center) and a list of available servers (right). The bottom half consists of a performance graph showing processed frames per second (left), and output log (right).

Location	Country	Type	Nodes	Cores	Efficiency
VU University, A'dam	NL	Grid (DAS-3)	32	128	97.3%
University of A'dam			16	64	96.5%
Delft University			32	64	94.0%
Leiden University			16	32	96.7%
Nat. Inst. of Inf., Chiba	JP	Grid (InTrigger)	8	16	84.0%
University of Tsukuba			8	64	81.1%
VU University, A'dam	NL	Desktop Grid	16	17	98.0%
Amazon EC2	USA	Cloud	16	16	93.2%
		Total	176	401	94.4%

Table 6.1: Sites used in the Divide-and-conquer GO experiment. Efficiency is calculated as the difference between total runtime of the application process, and time spent computing. Overhead includes joining and leaving, as well as application communication for load balancing, returning results, etc.

client simply keeps on forwarding video frames to all remaining servers. Lastly, the video shows the multimedia servers being accessed from a smartphone.

Our object recognition experiment shows that real-world distributed systems are a viable platform for running high-performance applications. The use of virtualization (in this case the Java virtual machine), the SmartSockets library, and our JEL resource tracking model, allows our application to efficiently use resources of a world-wide dynamic system. Also, this experiment shows Zorilla is useful for running parallel applications on *bare* resources which do not have any middleware installed.

6.2 Divide-and-conquer GO

Next, we use our world-wide set of resources to run a non-interactive application which uses the wide-area network more intensively. The application is an implementation of *First Capture Go*, a variant of the *Go* board game where a win is completed by capturing a single stone. Our application determines the optimal move for a given player, given any board position. It uses a simple brute-force algorithm for determining the solution, trying all possible moves recursively using a divide-and-conquer algorithm. Since the entire space needs to be searched to calculate the optimal answer, our application does not suffer from search overhead. Our Go application is implemented in Java, with many of the techniques used inspired by the Satin programming model [61]. It is implemented using the IPL communication library [60], which in turn uses JEL to track the resources available, and the SmartSockets library [54] to communicate between resources. Our application is highly malleable and fault-tolerant, automatically uses any new resources added, and continues computations even if resources are removed or fail.

Table 6.1 shows an overview of the sites used. We used a total of 176 machines, with a total of 401 cores. Figure 6.5 shows the communication structure of the ex-

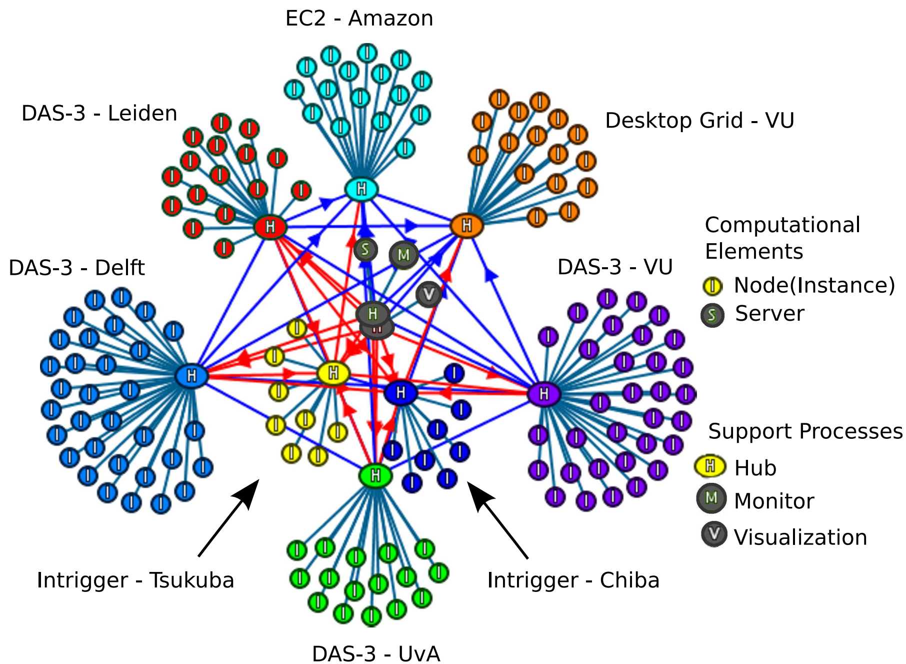


Figure 6.5: Communication structure of the world wide divide-and-conquer experiment. Nodes in this graph represent processes, edges represent connections. The experiment contains both nodes performing the computation, as well as a number of support processes which allow communication to pass through firewalls, monitor the communication, and produce this image. Each color represents a different location. Note that direct communication channels between individual nodes (even between separate clusters) are not shown, to maintain legibility of the graph.

periment. The graph shown is produced by the visualization of the SmartSockets library. In the graph, each site is represented by a different color. Next to the compute nodes themselves (called *Instances* in the graph), and the central server, a number of support processes is used. These support processes allow communication to pass through firewalls, monitor the communication, and produce the visualization shown. The support processes run on the frontend machines of the sites used.

Our world wide system finishes the capture Go application in 35 minutes. We measured the efficiency of the machine as the ration between the total time spent computing and the total runtime of the application. Overhead includes joining and leaving, as well as time spent communicating with other nodes to load balance the application, return results, etc. Efficiency of the nodes ranges from 79.8% to 99.1%. The low efficiency on some nodes is due to the severely limited connectivity of these nodes: the nodes of the InTrigger grid in Japan can only communicate with the outside world through an SSH tunnel, with a bandwidth of only 1Mbit/s and a latency of over 250ms to the DAS-3. Even with some nodes having a somewhat diminished efficiency, the average efficiency over all nodes in the world-wide experiment is excellent, at 94.4%.

Our experiment confirms that our system, including JEL, is suitable for running applications on a large scale and on a wide range of systems, including desktop grids and clouds concurrently, even when a significant amount of wide-area traffic is required.

6.3 Instant Cloud

Next, we use Zorilla to turn our entire collection of resources into one large instant-cloud and perform experiments. For this reason, we set IbisDeploy to use Zorilla, and let it discover and manage all resources. A Zorilla node is started on the headnode of each site used in the experiment. This node is then responsible for managing the resources available at that site. To show that Zorilla is capable of accessing resources using multiple middlewares, we use different ways of accessing the resources, including Globus, SGE and SSH. On the EC2 cloud, the desktop grid, and the stand-alone machine Zorilla is started on each machine individually. After startup, all Zorilla nodes form one large distributed system. Within this distributed system, our ARRG gossiping algorithm is used to create a locality-aware overlay network. The flood-scheduler of Zorilla uses this overlay network to allocate resources (see Figure 6.2).

We deployed the Go application on the entire instant cloud by submitting it to the Zorilla node on the local desktop machine. See Table 6.2 for an overview of all nodes used, and Figure 6.6 for the SmartSockets overlay network visualization. Zorilla deployed the application on 83 nodes, with over 200 cores. The applications achieved 87% efficiency overall, ranging from 72% on the poorly connected EVL cluster in Chicago, to over 90% on machines in the DAS system. This ex-

Location	Country	Type	Middleware	Nodes	Cores
VU University, A'dam	NL	Grid (DAS-3)	SGE	16	64
University of A'dam			Globus	16	64
Leiden University			prun	8	16
MultimediaN, A'dam			prun	16	32
EVL, Chicago	USA	Cluster	SSH	16	16
VU University, A'dam	NL	Desktop Grid	-	2	4
Amazon EC2	USA	Cloud	-	8	8
VU University, A'dam	NL	Desktop	-	1	1
			Total	83	205

Table 6.2: Sites used in the Instant Cloud experiment. All sites run Zorilla on the frontend. In some cases, Zorilla interfaces with existing middleware at the site to allocate resources. Middleware used includes Sun Grid Engine (SGE), Globus, and the custom *prun* scheduling interface available on the DAS-3. In the last three sites in the list, the resources themselves also run Zorilla nodes, and no other middleware is necessary.

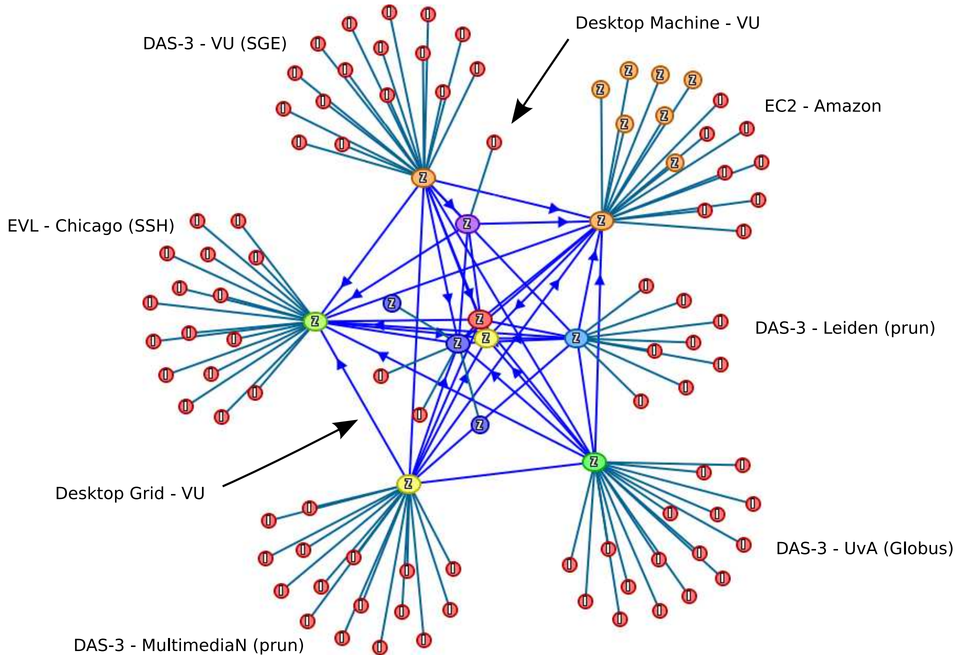


Figure 6.6: Resources used in the Zorilla experiment. This visualization shows all nodes, and the SmartSockets overlay network between them. A node marked *Z* represents a Zorilla node, running on either a frontend or a resource. A node marked *I* represents an instance of the Go application. As in Figure 6.5, only the SmartSockets overlay is shown, and links between individual nodes are removed from the graph for legibility.

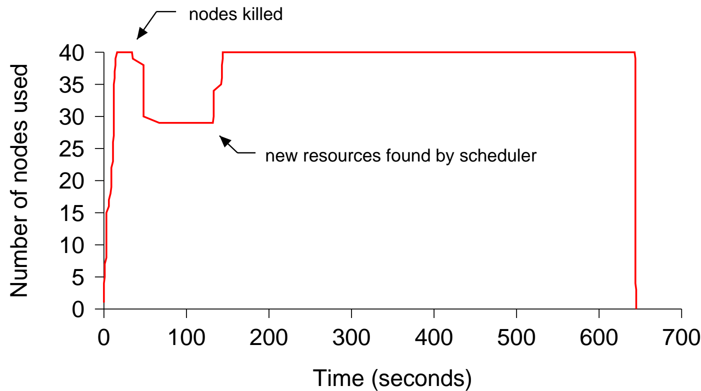


Figure 6.7: Fail test, where a job requiring 40 nodes was submitted. Zorilla automatically compensates when nodes fail by acquiring new resources.

periment shows that Zorilla is able to efficiently combine resources of a multitude of computing platforms, with different middlewares.

We also tested the ability of Zorilla to detect and respond to failures. Using the same distributed system as used in the previous experiment, we deployed the Go application on 40 nodes. As the flood-scheduler of Zorilla is locality-aware, it acquires mostly local resources (the stand alone machine, desktop grid, and local cluster), as well as some resources from other sites in The Netherlands. To simulate a resource failing, we manually killed all jobs running on our local cluster, totaling 11 nodes. As shown in Figure 6.7, the number of nodes used in the computation drops from 40 to 29. As we requested Zorilla to run the application on 40 nodes, it starts searching for additional resources. After a while, these resources are found and added to the computation. Since most close-by resources are already in use, some of the Amazon EC2 cloud resources are acquired. Subsequently, the number of resources used increases to 40 again.

Using JEL, the application is notified of the resources failing, as well as the new resources being available. The application responds by re-computing all lost results, and automatically starts using the new resources when they become available. In contrast to the previous experiment, the application does not use the poorly connected EVL resources. This significantly increases the efficiency of the application from 87% to 95%. This experiment shows that Zorilla is able to automatically acquire new resources in the face of failures, and optimizes resource acquirement for parallel and distributed applications. In addition, we have showed an example of the ARRG-based flood scheduler discovering resources in a locality-aware fashion, and show that JEL is able to support applications on a dynamic (faulty) system.

6.4 Competitions

The software produced by the Ibis project (which includes Zorilla, JEL, and other algorithms and techniques described in this thesis as its core components) also has been put to the test in a number of international competitions [5].

The first competition we participated in was SCALE 2008, or the First IEEE International Scalable Computing Challenge, held in conjunction with CCGrid 2008 (Lyon, France). Our submission consisted of the multimedia application described in Section 6.1, which is able to recognize objects in images taken from a video stream. These images are sent to a real-world distributed system for processing, and the resulting image descriptions are used to search for objects in a database. In our application, JEL is used to keep track of precisely which resources are available for processing images. Also, Zorilla (including both ARRG and the flood-scheduler) is used to manage and deploy the application on a number of resources used in the experiment.

The second competition was DACH 2008, or the First International Data Analysis Challenge for Finding Supernovae, held in conjunction with Cluster/Grid 2008 (Tsukuba, Japan) Here, the goal was to find 'supernova candidates' in a large distributed database of telescope images. We used JEL in our submission to keep track of all the available resources.

The DACH challenge consisted of two categories: a *Basic Category* where the objective was to search the entire database as fast as possible, and a *Fault-Tolerant* category, where next to speed, fault tolerance was also measured by purposely killing over 30% of the nodes in the computation. Especially in the Fault-Tolerant category, JEL was vital for the successful completion of the application.

The third and last competition was the Semantic Web Challenge 2008, held in conjunction with the Semantic Web Conference (ISWC 2008), Karlsruhe, Germany. Here, we participated in the *Billion Triples Track*. In this challenge, the primary goal was to demonstrate scalability of Semantic Web architectures and applications. The participants were given a collection of one billion triples (single facts) and had to demonstrate added value from the implicit semantics in the data. This could involve browsing and visualization of the data, could include inferencing to add implicit information to the dataset, etc. Our submission consisted of MaRVIN [64]: a self-organizing distributed reasoner that can handle these large amounts of data. Marvin is build using the IPL, which in turn uses JEL for resource tracking.

Using our software, we have won first prize in both SCALE 2008 and DACH 2008, and third price at the Semantic Web Challenge, in a field of international competitors. Moreover, we won both the Basic and the Fault-Tolerant categories at DACH 2008. These prizes [5] show that our software and techniques are very effective in many real-world scenarios, including dynamic systems with failing nodes.

Chapter 7

Summary and Conclusions

In this thesis, we have investigated *how to run distributed supercomputing applications on very heterogeneous, dynamic, systems.* We used the term *real-world distributed system* for the type of systems users have access to and need to (though not necessarily want to) use to run their high-performance applications. We explicitly take into account distributed supercomputing applications, in which resources from multiple sites cooperate in a single high-performance distributed computation.

In our research, we focus on how to get an existing distributed supercomputing application to run on available resources. Besides resource discovery, scheduling, and managing resources, we also investigate tracking exactly which resources are available in a computation. Throughout this thesis we use Zorilla, our prototype P2P middleware, as a research platform.

In Chapter 2, we investigated middleware for real-world distributed systems. The emergence of these systems has made running high-performance and large-scale applications a challenge for end-users. Real-world distributed systems are heterogeneous, faulty, and constantly changing. We suggest a possible solution for these problems: instant cloud middleware. We established the requirements of such a middleware, consisting mainly of the capability to overcome all limitations of real-world distributed systems. Requirements include fault-tolerance, platform independence, and support for parallel applications.

We introduced Zorilla, a prototype P2P middleware designed for creating an instant cloud out of any available resources used concurrently, including stand-alone machines, clusters, grids, and clouds. Zorilla explicitly supports running distributed supercomputing applications on the resulting system. Zorilla uses a combination of Virtualization and P2P techniques to implement all functionality, resulting in a simple, effective, and robust system. For instance, the flood-scheduling system in Zorilla makes use of the fact that resources are virtualized, allowing for a simple yet effective resource discovery mechanism based on P2P techniques.

In Chapter 3, we studied the design and implementation of gossiping algorithms in real-world situations. We addressed the problems with gossiping algorithms in real systems, including connectivity problems, network and node failures, and non-atomicity. We introduced *ARRG*, a new simple and robust gossiping algorithm. The ARRG gossiping algorithm is able to handle all problems we identified by systematically using the simplest, most robust solution available for all required functionality. The *Fallback Cache* technique used in ARRG can also be applied to any existing gossiping protocol, making it robust against problems such as NATs and firewalls.

We introduced a new metric for the evaluation of gossiping algorithms: *Perceived Network Size*. It is able to clearly characterize the performance of an algorithm, without requiring information from all nodes in the network. We evaluated ARRG, in several real-world scenarios. We showed that ARRG performs well in general, and better than existing algorithms in situations with limited connectivity. In a pathological scenario with a high loss rate and 80% of the nodes behind a NAT system, ARRG still performs well, while traditional techniques fail.

In Chapter 4, we have studied the scheduling of supercomputing applications in P2P environments. We introduced *flood scheduling*: a scheduling algorithm based on flooding messages over a P2P network. Flood scheduling is fully decentralized, supports co-allocation and has good fault-tolerance properties. Flood scheduling depends on the locality-awareness of the P2P network used.

Using Zorilla, we were able to deploy and run a parallel divide-and-conquer application on 671 processors simultaneously, solving the N-Queens 22 problem in 35 minutes. We used six clusters of the Grid5000 [9] system, located at sites across France. This large scale experiment on a real grid showed that flood scheduling is able to effectively allocate resources to jobs in a locality-aware way across entire grids.

In Chapter 5, we have studied resource tracking mechanisms. With the transition from static cluster systems to dynamic environments such as grids, clusters, clouds, and P2P systems, fault-tolerance and malleability are now essential features for applications running in these environments. This is especially so for an application running on an instant cloud as created by Zorilla, as these are inherently dynamic systems. A first step in creating a fault-tolerant and malleable system is *resource tracking*: the capability to track exactly which resources are part of a computation, and what roles they have. Resource tracking is an essential feature in any dynamic environment, and should be implemented on the same level of the software hierarchy as communication primitives.

We introduced JEL: a unified model for tracking resources. JEL is explicitly designed to be scalable and flexible. Although the JEL model is simple, it supports both traditional programming models such as MPI, and flexible grid oriented models like Satin. JEL allows programming models such as Satin to implement both malleability and fault-tolerance. With JEL as a common layer for resource tracking, the development of programming models is simplified considerably.

JEL can be used on environments ranging from clusters to highly dynamic P2P environments. We described several implementations of JEL, including a

centralized implementation that can be combined with decentralized dissemination techniques, resulting in high performance, yet with low resource usage at the central server. In addition, we showed that JEL can be implemented in a fully distributed manner, using the ARRG gossiping algorithm as a basis. This distributed implementation efficiently supports flexible programming models such as Satin, and increases fault-tolerance compared to a centralized implementation.

We evaluated JEL in several real-world scenarios. The scenarios include starting 2000 instances of an application, and wide area tests with new machines joining, and resources failing.

In Chapter 6, we performed several experiments that showed the feasibility of real-world distributed systems for high-performance computing. We covered all the software and techniques developed for this thesis, and demonstrated these in large-scale dynamic systems. Using Zorilla, we ran a world-wide experiment, showing how Zorilla can tie together a large number of resources into one coherent system. Moreover, we have shown that these resources can be used efficiently, even when faults occur. Zorilla allows users to transparently use large numbers of resources, even on very heterogeneous distributed systems comprised of grids, clusters, clouds, desktop grids, and other systems. We also show the real-world applicability of our research, by describing a number of awards won in international competitions with our software.

Zorilla, and the techniques it incorporates described in this thesis, greatly enhance the applicability of real-world distributed systems for everyday users. Instead of limiting usage of these systems to a single site at a time, it is now possible to routinely use large numbers of resources, possibly distributed across the globe. Moreover, the work described in this thesis, combined with the complementary research done in the Ibis project, allows users to do this transparently, and with little effort. Instead of constantly managing files, jobs, and resources, users can now focus on the actual computations performed with their application.

Although we have shown Zorilla to be highly useful, it is far from complete. We plan to add more functionality to Zorilla to increase the number of use-cases it supports. Extensions include improved security, to enable Zorilla to support applications which use sensitive information such as medical data. Also, recent years have shown a dramatic increase in the amount of data used in computations, leading to a need for more advanced data storage and managing capabilities.

Looking ahead, we predict that real-world distributed systems will be used more and more by average users. The current trend of increasing parallelism in even a single machine will also change the parallel computing landscape considerably. Multi-core and many-core architectures like GPUs will need to be incorporated in current and future systems to keep up with growing user demands. On the other hand, the breakthrough of parallelism on every desktop also provides an opportunity, as more programmers will be familiar with parallel concepts. We also predict a renewed interest for parallel programming models, hiding as much of the (growing) complexity as possible.

Bibliography

[1] H. Abbes and C. Crin. A decentralized and fault-tolerant desktop grid system for distributed applications. *Concurr. Comput. : Pract. Exper.*, 22(3):261–277, 2010.

[2] A. Allavena, A. Demers, and J. E. Hopcroft. Correctness of a gossip based membership protocol. In *PODC '05: Proc. of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 292–301, New York, NY, USA, 2005. ACM Press.

[3] O. Babaoğlu, A. Bartoli, and G. Dini. Enriched view synchrony: A programming paradigm for partitionable asynchronous distributed systems. *IEEE Trans. Comput.*, 46(6):642–658, 1997.

[4] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. Programming, Deploying, Composing, for the Grid. In *Grid Computing: Software Environments and Tools*. Springer-Verlag, Jan. 2006.

[5] H. E. Bal, N. Drost, R. Kemp, J. Maassen, R. V. van Nieuwpoort, C. van Reeuwijk, and F. J. Seinstra. Ibis: Real-world problem solving using real-world grids. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[6] H. E. Bal, J. Maassen, R. V. van Nieuwpoort, N. Drost, N. Palmer, G. Wrzesinska, T. Kielmann, K. van Reeuwijk, F. J. Seinstra, C. Jacobs, R. Kemp, and K. Verstoep. Real-world distributed computing with ibis. *IEEE Computer*, 43(8):54–62, 2010.

[7] M. Beck, J. J. Dongarra, G. E. Fagg, G. A. Geist, P. Gray, J. Kohl, M. Migliardi, K. Moore, T. Moore, P. Papadopoulous, S. L. Scott, and V. Sunderam. Harness: a next generation distributed virtual machine. *Future Generation Computer Systems*, 15(5-6):571–582, 1999.

[8] K. P. Birman, R. v. Renesse, and W. Vogels. Navigating in the Storm: Using Astrolabe to Adaptively Configure Web Services and Their Clients. *Cluster Computing*, 9(2):127–139, 2006.

[9] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *Int. J. High Perform. Comput. Appl.*, 20(4):481–494, 2006.

[10] M. Bornemann, R. V. van Nieuwpoort, and T. Kielmann. MPJ/Ibis: a flexible and efficient message passing platform for Java. In *Proceedings of PVM/MPI'05*, Sorrento, Italy, September 2005.

[11] J. Bustos-Jimenez, D. Caromel, A. di Costanzo, M. Leyton, and J. M. Piquer. Balancing active objects on a peer to peer infrastructure. In *Proc. of the XXV International Conference of the Chilean Computer Science Society (SCCC 2005)*, Valdivia, Chile, November 2005.

[12] D. Butler. The Petaflop Challenge. *Nature*, 448:6–7, 2007.

[13] A. R. Butt, R. Zhang, and Y. C. Hu. A self-organizing flock of condors. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 42, Washington, DC, USA, 2003. IEEE Computer Society.

[14] F. Cappello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Néri, and O. Lodygensky. Computing on large scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid. *Future Generation Computer Systems*, 21(3):417–437, March 2005.

[15] D. Caromel, A. d. Costanzo, and C. Mathieu. Peer-to-peer for computational grids: mixing clusters and desktop machines. *Parallel Computing*, 33(4-5):275–288, 2007.

[16] A. Chandra and J. Weissman. Nebulas: Using distributed voluntary resources to build clouds. In *HotCloud '09*, 2009.

[17] K. Czajkowski, C. Kesselman, S. Fitzgerald, and I. Foster. Grid information services for distributed resource sharing. *High-Performance Distributed Computing, International Symposium on*, 0:0181, 2001.

[18] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: a decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.*, 34:15–26, August 2004.

[19] DAS-2 website. `http://www.cs.vu.nl/das2`.

[20] DAS-3 website. `http://www.cs.vu.nl/das3`.

[21] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *PODC '87: Proc. of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, New York, NY, USA, 1987. ACM Press.

[22] A. di Costanzo, M. D. de Assuncao, and R. Buyya. Harnessing cloud technologies for a virtualized distributed computing infrastructure. *IEEE Internet Computing*, 13(5):24–33, 2009.

[23] Distributed.net. `http://distributed.net`.

[24] N. Drost, E. Ogston, R. V. van Nieuwpoort, and H. E. Bal. ARRG: Real-World Gossiping. In *Proceedings of The 16th IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, Monterey, CA, USA, June 2007.

[25] N. Drost, R. V. van Nieuwpoort, and H. E. Bal. Simple locality-aware co-allocation in peer-to-peer supercomputing. In *Sixth International Workshop on Global and Peer-2-Peer Computing (GP2P 2006)*, Singapore, May 2006.

[26] N. Drost, R. V. van Nieuwpoort, J. Maassen, and H. E. Bal. Resource tracking in parallel and distributed applications. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 221–222, New York, NY, USA, 2008. ACM.

[27] N. Drost, R. V. van Nieuwpoort, J. Maassen, F. Seinstra, and H. E. Bal. JEL: Unified resource tracking for parallel and distributed applications. *Concurr. Comput. : Pract. Exper.*, 2010. Accepted for publication.

[28] N. Drost, R. V. van Nieuwpoort, J. Maassen, F. J. Seinstra, and H. E. Bal. Zorilla: Instant cloud computing. 2010. Submitted for publication.

[29] Amazon ec2 website. `http://aws.amazon.com/ec2`.

[30] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[31] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, 2003.

[32] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulie. From epidemics to distributed computing. *IEEE Computer*, 37(5):60–67, 2004.

[33] G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London, and J. J. Dongarra. Extending the MPI specification for process fault tolerance on high performance computing systems. In *Proceedings of ICS'04*, June 2004.

[34] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*, pages 83–92, New York, NY, USA, 1998. ACM.

[35] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, 2001.

[36] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulie. Peer-to-peer membership management for gossip-based protocols. *IEEE Trans. Comput.*, 52(2):139–149, 2003.

[37] A. Ganguly, A. Agrawal, P. Boykin, and R. Figueiredo. Wow: Self-organizing wide area overlay networks of virtual workstations. *Journal of Grid Computing*, 5:151–172, 2007. 10.1007/s10723-007-9076-6.

[38] Gnutella. The gnutella protocol specification. `http://rfc-gnutella.sourceforge.net`.

[39] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. von Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf. SAGA: A Simple API for Grid Applications, High-Level Application Programming on the Grid. *Computational Methods in Science and Technology*, 12(1):7–20, 2006.

[40] J.-P. Goux, S. Kulkarni, M. Yoder, and J. Linderoth. An enabling framework for master-worker applications on the computational grid. In *HPDC '00: Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing*, page 43, Washington, DC, USA, 2000. IEEE Computer Society.

[41] A. Gualandris, S. P. Zwart, and A. Tirado-Ramos. Performance analysis of direct n-body algorithms for astrophysical simulations on distributed systems. *Parallel Comput.*, 33:159–173, April 2007.

[42] I. Gupta, R. v. Renesse, and K. P. Birman. A probabilistically correct leader election protocol for large groups. In *DISC '00: Proceedings of the 14th International Conference on Distributed Computing*, pages 89–103, London, UK, 2000. Springer-Verlag.

[43] W. Huang, J. Liu, B. Abali, and D. K. Panda. A case for high performance computing with virtual machines. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 125–134, New York, NY, USA, 2006. ACM.

[44] Ibis website. `http://www.cs.vu.nl/ibis`.

[45] Intrigger website. `http://www.intrigger.jp`.

[46] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 79–98, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[47] M. Jelasity, W. Kowalczyk, and M. van Steen. Newscast computing. Technical report, Vrije Universiteit Amsterdam, Department of Computer Science, 2003.

[48] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 131–140, New York, NY, USA, 1999. ACM.

[49] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[50] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122 (Proposed Standard), July 2005.

[51] M.-J. Lin and K. Marzullo. Directional gossip: Gossip in a wide area network. In *EDCC-3: Proc. of the Third European Dependable Computing Conference on Dependable Computing*, pages 364–379, London, UK, 1999. Springer-Verlag.

[52] M.-J. Lin, K. Marzullo, and S. Masini. Gossip versus deterministically constrained flooding on small networks. In *DISC '00: Proc. of the 14th International Conference on Distributed Computing*, pages 253–267, London, UK, 2000. Springer-Verlag.

[53] J. Maassen. *Method Invocation Based Communication Models for Parallel Programming in Java.* PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands, June 2003.

[54] J. Maassen and H. E. Bal. SmartSockets: solving the connectivity problems in grid computing. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 1–10, New York, NY, USA, 2007. ACM.

[55] H. Mohamed and D. Epema. Koala: a co-allocating grid scheduler. *Concurr. Comput. : Pract. Exper.*, 20(16):1851–1876, 2008.

[56] J. Montagnat, V. Breton, and I. E. Magnin. Using grid technologies to face medical image analysis challenges. In *Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, CCGRID '03, pages 588–, Washington, DC, USA, 2003. IEEE Computer Society.

[57] MPI forum website. `http://www.mpi-forum.org`.

[58] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: a read/write peer-to-peer file system. *SIGOPS Oper. Syst. Rev.*, 36:31–44, December 2002.

[59] R. Nieuwpoort, T. Kielmann, and H. E. Bal. User-friendly and reliable grid computing based on imperfect middleware. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–11, New York, NY, USA, 2007. ACM.

[60] R. Nieuwpoort, J. Maassen, G. Wrzesińska, R. F. H. Hofman, C. J. H. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a flexible and efficient java-based grid programming environment. *Concurr. Comput. : Pract. Exper.*, 17(7-8):1079–1107, 2005.

[61] R. Nieuwpoort, G. Wrzesinska, C. J. Jacobs, and H. E.Bal. Satin: a high-level and efficient grid programming model. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(3), 2010.

[62] Nimbus science cloud. `http://www.nimbusproject.org`.

[63] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud-computing system. In *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[64] E. Oren, S. Kotoulas, G. Anadiotis, R. Siebes, A. ten Teije, and F. van Harmelen. Marvin: A platform for large-scale analysis of semantic web data. In *Proceeding of WebSci'09: Society On-Line*, Athens, Greece, March 2009.

[65] Penguin computing on demand website. `http://www.penguincomputing.com/POD/Penguin_On_Demand`.

[66] J. Postel. Transmission Control Protocol. RFC 793 (Standard), Sept. 1981. Updated by RFCs 1122, 3168.

[67] R. v. Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.

[68] R. v. Renesse, Y. Minsky, and M. Hayden. A gossip-based failure detection service. In *Middleware'98, IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 55–70, England, September 1998.

[69] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proc. of the USENIX Annual Technical Conference*, pages 127–140, Boston, MA, USA, June 2004.

[70] F. J. Seinstra, J.-M. Geusebroek, D. Koelma, C. G. M. Snoek, M. Worring, and A. W. M. Smeulders. High-Performance Distributed Video Content Analysis with Parallel-Horus. *IEEE Multimedia*, 14(4):64–75, 2007.

[71] Seti@home project website. `http://setiathome.ssl.berkeley.edu`.

[72] T. Smith and M. Watherman. Identification of common molecular subsequences. *Journal of Molecular biology*, 147, 1981.

[73] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing*, 13(5):14–22, 2009.

[74] A. Stavrou, D. Rubenstein, and S. Sahu. A lightweight, robust p2p system to handle flash crowds. In *ICNP '02: Proc. of the 10th IEEE International Conference on Network Protocols*, pages 226–235, Washington, DC, USA, 2002. IEEE Computer Society.

[75] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris. Flexible, wide-area storage for distributed systems with wheelfs. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 43–58, Berkeley, CA, USA, 2009. USENIX Association.

[76] G. Tan, S. A. Jarvis, X. Chen, and D. P. Spooner. Performance analysis and improvement of overlay construction for peer-to-peer live streaming. *Simulation*, 82(2):93–106, 2006.

[77] K. Taura, K. Kaneda, T. Endo, and A. Yonezawa. Phoenix: a parallel programming model for accommodating dynamically joining/leaving resources. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 216–229, New York, NY, USA, 2003. ACM.

[78] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurr. Comput. : Pract. Exper.*, 17(2-4):323–356, 2005.

[79] Virtualbox website. `http://www.virtualbox.org/`.

[80] S. Voulgaris. *Epidemic-Based Self-Organization in Peer-to-Peer Systems.* PhD thesis, Vrije University Amsterdam, 2006.

[81] S. Voulgaris, D. Gavidia, and M. Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, June 2005.

[82] S. Voulgaris, M. Jelasity, and M. van Steen. A robust and scalable peer-to-peer gossiping protocol. In *Proc. of the 2nd International Workshop on Agents and Peer-to-Peer Computing (AP2PC03)*, Melbourne, Australia, July 2003.

[83] J. Waldo. Remote procedure calls and java remote method invocation. *IEEE Concurrency*, 6(3):5–7, 1998.

[84] Wikipedia. Striped polecat. `http://en.wikipedia.org/w/index.php?title=Striped_Polecat&oldid=385699699`, 2010.

[85] G. Wrzesinska, J. Maassen, and H. E. Bal. Self-adaptive applications on the grid. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*, San Jose, CA, USA, March 2007.

[86] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. In *Proc. of the International Conference on Distributed Computing Systems*, pages 5–14, Vienna, Austria, July 2002.

[87] A. YarKhan, J. Dongarra, and K. Seymour. Gridsolve: The evolution of network enabled solver. In *Proceedings of IFIP WoCo9*, Prescott, AZ, USA, July 2006.

[88] B. Y. Zhao, A. Joseph, and J. Kubiatowicz. Locality aware mechanisms for large-scale networks. In *Proc. of Workshop on Future Directions in Distributed Comp*, June 2002.

Samenvatting

In dit proefschrift, getiteld *gedistribueerde supercomputer applicaties in de echte wereld*, hebben we onderzocht hoe gedistribueerde supercomputer applicaties gedraaid kunnen worden op zeer heterogene, dynamische systemen. We gebruikten de term real-world gedistribueerd systeem voor het type van systemen waar gebruikers toegang toe hebben en dat deze gebruikers (vaak tegen wil en dank) moeten gebruiken om applicaties te draaien die zeer veel rekenkracht vragen, zogenaamde *high-performance*-applicaties. We hebben expliciet rekening gehouden met gedistribueerde supercomputer-toepassingen, waarin computers op meerdere locaties samenwerken in één gedistribueerde berekening.

In ons onderzoek richtten we ons op het draaien van bestaande gedistribueerde supercomputer applicaties op alle beschikbare machines. Naast het ontdekken van beschikbare machines, scheduling en het managen van gebruikte computers, onderzochten we ook het bijhouden van welke computers precies beschikbaar zijn in een berekening. Het onderzoek in dit proefschrift maakt gebruik van Zorilla, ons prototype Peer-to-Peer(P2P) middleware systeem, als onderzoeksplatform.

In Hoofdstuk 2 hebben we middleware voor real-world gedistribueerde systemen onderzocht. De opkomst van deze systemen heeft het draaien van high-performance toepassingen op grote schaal problematisch gemaakt voor eindgebruikers. Real-world gedistribueerde systemen zijn heterogeen, gaan vaak stuk en veranderen voortdurend. Wij suggereren een mogelijke oplossing voor deze problemen: instant cloud middleware. We stellen de vereisten voor een dergelijke middleware vast. Deze bestaan voornamelijk uit het overwinnen van alle beperkingen van real-world gedistribueerde systemen, zoals het tolereren van defecten, platformonafhankelijkheid en ondersteuning voor parallelle applicaties.

We introduceerden Zorilla, een prototype P2P middleware ontworpen voor het creëren van instant cloud uit alle mogelijke beschikbare systemen, met inbegrip van PC's, clusters, grids en clouds. Zorilla heeft expliciet ondersteuning voor het draaien van gedistribueerde supercomputer-toepassingen. Zorilla gebruikt een combinatie van virtualisatie en P2P-technieken om alle functionaliteit te implementeren, wat resulteert in een eenvoudig, efficiënt en robuust systeem. Het flood-scheduling systeem in Zorilla maakt bijvoorbeeld gebruik van het feit dat de machines gevirtualiseerd zijn, waardoor een simpel maar effectief algoritme, gebaseerd op P2P technieken gebruikt kan worden om nieuwe machines te ontdekken.

In Hoofdstuk 3 hebben we het ontwerp en de implementatie van *gossiping*(roddel) algoritmen in realistische situaties bestudeerd. We hebben de problemen aangepakt die optreden bij het gebruik van gossiping algoritmen in realistische situaties, zoals connectiviteitsproblemen en netwerk- en computerstoringen. We introduceerden ARRG, een eenvoudig en robuust gossiping algoritme. Het ARRG gossiping algoritme lost alle problemen die we geïdentificeerd hebben op, door systematisch gebruik van de eenvoudigste, meest robuuste oplossing beschikbaar. De Fallback Cache techniek die gebruikt wordt in ARRG kan ook worden toegepast op alle bestaande gossiping algoritmen, waardoor deze robuuster worden voor problemen zoals NAT-systemen en firewalls.

We evalueerden ARRG in verschillende realistische scenario's. We toonden aan dat ARRG over het algemeen goed presteert en beter presteert dan bestaande algoritmen in situaties met beperkte connectiviteit. In een pathologisch scenario met veel storingen en 80% van de machines achter een NAT-systeem, presteerde ARRG nog steeds goed, terwijl de traditionele gossiping-technieken falen.

In Hoofdstuk 4 hebben we de scheduling van supercomputing-toepassingen in P2P omgevingen onderzocht. We introduceerden *flood scheduling*: een scheduling algoritme gebaseerd op een vloedgolf van berichten over een netwerk. Flood scheduling is volledig gedecentraliseerd, ondersteunt co-allocatie en is foutbestendig.

Met behulp van Zorilla waren we in staat een parallelle *verdeel-en-heers* toepassing op 671 processoren tegelijkertijd te draaien. Deze applicatie loste het 22-Koninginnen probleem op in 35 minuten. We gebruikten zes clusters van het Grid5000 systeem op verschillende locaties in Frankrijk. Dit experiment toont aan dat flood scheduling in staat is om effectief machines te alloceren in een grootschalig systeem.

In Hoofdstuk 5 hebben we zogenaamde *resource tracking*-mechanismen onderzocht. Met de overgang van statische clustersystemen naar meer dynamische omgevingen zoals grids, clusters, clouds en P2P systemen, zijn foutbestendigheid en flexibiliteit nu van essentieel belang voor applicaties die draaien in deze omgevingen. Dit is extra van toepassing op een applicatie die draait op een instant cloud zoals gecreëerd door Zorilla, aangezien dit inherent dynamische systemen zijn. Een eerste stap in het creëren van een storingsongevoelig en flexibel systeem is resource tracking: de mogelijkheid om precies bij te houden welke machines deel uitmaken van een berekening en welke rol ze hebben. Resource tracking is een essentieel onderdeel van elke dynamische omgeving.

We introduceerden JEL: een resource tracking model voor het bijhouden van beschikbare machines in een gedistribueerde applicatie. JEL is expliciet ontworpen om zowel schaalbaar als flexibel te zijn. Hoewel het JEL model erg eenvoudig is, ondersteunt het zowel traditionele programmeermodellen zoals MPI, als flexibele grid programmeermodellen zoals Satin. JEL stelt programmeermodellen zoals Satin in staat zowel flexibel als foutbestendig te zijn. Met JEL als een gemeenschappelijke laag voor resource tracking is het ontwikkelen van programmeermodellen aanzienlijk vereenvoudigd.

JEL kan gebruikt worden op omgevingen variërend van clusters tot zeer dynamische P2P-omgevingen. We beschreven verschillende implementaties van JEL.

Een daarvan is een gecentraliseerde implementatie die kan worden gecombineerd met decentrale verspreidingstechnieken, wat resulteert in hoge prestaties en een lage belasting van de centrale server. Bovendien toonden we aan dat JEL kan worden geïmplementeerd op een volledig gedistribueerde wijze met gebruik van het ARRG gossiping algoritme als basis. Deze gedistribueerde implementatie ondersteunt efficiënt flexibele programmeermodellen zoals Satin en heeft een betere foutbestendigheid dan de gecentraliseerde versie.

We evalueerden JEL in verschillende realistische scenario's, waaronder het starten van een applicatie op 2000 machines en situaties waarin er constant veranderingen zijn aan de beschikbare machines. Ook testten wij JEL in een scenario waar er storingen optreden.

In Hoofdstuk 6 hebben we een aantal experimenten uitgevoerd die de haalbaarheid testen van het gebruik van real-world gedistribueerde systemen voor high-performance toepassingen. We testten alle software en technieken ontwikkeld voor dit proefschrift en toonden aan dat deze goed functioneren in grootschalige dynamische systemen. Met behulp van Zorilla deden we een wereldwijd experiment waarmee we aantoonden dat Zorilla een groot aantal machines tot één groot systeem kan smeden. Bovendien hebben we aangetoond dat deze machines efficint gebruikt kunnen worden, zelfs wanneer er storingen optreden. Zorilla stelt gebruikers in staat om grote aantallen machines op een makkelijk manier te gebruiken, zelfs in zeer heterogene gedistribueerde systemen bestaande uit grids, clusters, clouds, desktop grids, en andere systemen.

We hebben ook de toepasbaarheid van ons onderzoek aangetoond met de beschrijving van een aantal prijzen gewonnen in internationale wedstrijden met onze software. De technieken beschreven in dit proefschrift vergroten de toepasbaarheid van real-world gedistribueerde systemen voor de dagelijkse gebruikers zeer. Het is nu mogelijk om routinematig een groot aantal machines tegelijk te gebruiken, eventueel verdeeld over de hele wereld. Het werk beschreven in dit proefschrift, gecombineerd met het complementair onderzoek gedaan in het Ibis-project, stelt gebruikers in staat om dit tevens op een gemakkelijke wijze te doen.

Hoewel we hebben laten zien dat Zorilla zeer nuttig is, is het verre van compleet. We zijn van plan om meer functionaliteit toe te voegen aan Zorilla, zoals verbeterde beveiliging. De afgelopen jaren hebben we een dramatische toename gezien van de hoeveelheid data die gebruikt wordt in berekeningen. Hierdoor is het noodzakelijk geworden meer geavanceerde data-opslag en -beheer toe te voegen aan Zorilla.

In de toekomst zullen real-world gedistribueerde systemen meer en meer gebruikt worden door eindgebruikers. De huidige trend van toenemende parallellisme in een enkele machine zal ook het programmeerlandschap danig veranderen. Multicore machines en GPU's zullen moeten worden opgenomen in de huidige en toekomstige systemen om te voldoen aan de eisen van eindgebruikers. De doorbraak van parallellisme op elke PC biedt ook een kans, omdat er meer programmeurs zullen zijn met kennis van parallelle concepten. Wij voorspellen ook een hernieuwde belangstelling voor parallelle programmeermodellen, met name modellen die zoveel mogelijk verbergen van de (toenemende) complexiteit.

Curriculum Vitae

Personal Data

Name:	Niels Drost
Place of birth:	Alkmaar, The Netherlands
Date of birth:	October 18, 1978
Nationality:	Dutch
Current Address:	Department of Computer Science Faculty of Sciences, VU University De Boelelaan 1081, 1081HV Amsterdam, The Netherlands niels@cs.vu.nl http://www.cs.vu.nl/∼niels/

Education

M.Sc.: Dept. of Computer Science, Faculty of Sciences,
VU University, Amsterdam 2004
VWO (pre-university secondary school), Alkmaar 1998
HAVO/MBO (higher level secondary school), Alkmaar 1997
MAVO (medium level secondary school), Heerhugowaard 1995

Grants and Prizes

First Prize Winner: J. Urbani, S. Kotoulas, J. Maassen, **N. Drost**, F.J. Seinstra, F. van Harmelen, and H.E. Bal. "WebPIE: A Web-Scale Parallel Inference Engine". In *3rd IEEE International Scalable Computing Challenge (SCALE 2010)*, held in conjunction with the *10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2010)* Melbourne, Victoria, Australia, May 17-20, 2010.

Third Prize Winner: G. Anadiotis, S. Kotoulas, E. Oren, R. Siebes, F. van Harmelen, **N. Drost**, R. Kemp, J. Maassen, F.J. Seinstra, and H.E. Bal. "MaRVIN: a distributed platform for massive RDF inference". *Semantic Web Challenge 2008 (Billion Triples Track), held in conjunction with the 7th International Semantic Web Conference (ISWC 2008)*, Karlsruhe, Germany, October 26-30, 2008.

First Prize Winner: F.J. Seinstra, **N. Drost**, R. Kemp, J. Maassen, R.V. van Nieuwpoort, K. Verstoep, and H.E. Bal. "Scalable Wall-Socket Multimedia Grid Computing". *1st IEEE International Scalable Computing Challenge (SCALE2008), held in conjunction with the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008),* Lyon, France, May 2008.

Best Paper Nomination: N. Drost, E. Ogston, R.V. van Nieuwpoort, and H.E. Bal. "ARRG: Real-World Gossiping". In *Proceedings of the 16th IEEE International Symposium on High-Performance Distributed Computing (HPDC 2007)*, Monterey, CA, USA, June 2007.

Peer-reviewed Journal Publications

N. Drost, R.V. van Nieuwpoort, J. Maassen, F.J. Seinstra, and H.E. Bal. "Zorilla: Instant Cloud Computing". Submitted for publication.

H.E. Bal, J. Maassen, R.V. van Nieuwpoort, **N. Drost**, N. Palmer, G. Wrzesinska, T. Kielmann, K. van Reeuwijk, F.J. Seinstra, C. Jacobs, R. Kemp, and K. Verstoep. "Real-World Distributed Computing with Ibis". IEEE Computer, 43(8): 54-62, August 2010.

N. Drost, R.V. van Nieuwpoort, J. Maassen, F.J. Seinstra, and H.E. Bal. "JEL: Unified Resource Tracking for Parallel and Distributed Applications". *Concurrency and Computation: Practice and Experience*, to appear, 2010.

Peer-reviewed Conference Publications

T. van Kessel, **N. Drost**, and F.J. Seinstra. "User Transparent Task Parallel Multimedia Content Analysis". In *Proceedings of the 16th International Euro-Par Conference (Euro-Par 2010)*, Naples, Italy, August 2010.

R. Kemp, N. Palmer, T. Kielmann, F.J. Seinstra, **N. Drost**, J. Maassen, and H.E. Bal. "eyeDentify: Multimedia Cyber Foraging from a Smartphone". *IEEE International Symposium on Multimedia (ISM2009)*, San Diego, California, December 2009.

H.E. Bal, **N. Drost**, R. Kemp, J.Maassen, Rob V. van Nieuwpoort, C. van Reeuwijk, and F.J. Seinstra. "Ibis: Real-world problem solving using Real-World Grids". In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009), HPGC workshop*, Rome, Italy, May 2009.

N. Drost, R.V. van Nieuwpoort, J. Maassen, and H.E. Bal. "Resource Tracking in Parallel and Distributed Applications". In *Proceedings of the 17th IEEE International Symposium on High-Performance Distributed Computing (HPDC 2008)*, Boston, MA, USA, June 2008.

H.E. Bal, J. Maassen, R.V. van Nieuwpoort, T. Kielmann, **N. Drost**, C. Jacobs, F.J. Seinstra, R. Kemp, and K. Verstoep. "The Ibis Project: Simplifying Grid Programming and Deployment". European Projects Showcase, special track at *the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008)*, Lyon, France, 19-22 May, 2008.

N. Drost, E. Ogston, R.V. van Nieuwpoort, and H.E. Bal. "Gossiping in the real world". *13th Annual Conference of the Advanced School for Computing and Imaging (ASCI 2007)*, Heijen, The Netherlands, June 2007.

N. Drost, E. Ogston, R.V. van Nieuwpoort, and H.E. Bal. "ARRG: Real-World Gossiping". In *Proceedings of the 16th IEEE International Symposium on High-Performance Distributed Computing (HPDC 2007)*, Monterey, CA, USA, June 2007.

N. Drost, R.V. van Nieuwpoort, and H.E. Bal. "Simple Locality-Aware Co-Allocation in Peer-to-Peer Supercomputing". *12th Annual Conference of the Advanced School for Computing and Imaging (ASCI 2006)*, Lommel, Belgium, June 2006.

N. Drost, R.V. van Nieuwpoort, and H.E. Bal. "Simple Locality-Aware Co-Allocation in Peer-to-Peer Supercomputing". In *Sixth International Workshop on Global and Peer-2-Peer Computing (GP2P 2006)*, Singapore, May 2006.

Refereed Book Chapters

F.J. Seinstra, J. Maassen, R.V. van Nieuwpoort, **N. Drost**, T. van Kessel, B. van Werkhoven, J. Urbani, C. Jacobs, T. Kielmann, and H.E. Bal "Jungle Computing: Distributed Supercomputing beyond Clusters, Grids, and Clouds" In: M. Cafaro and G. Aloisio, editors, *Grids, Clouds and Virtualization*, pp. 167-197, Springer-Verlag, 2011.

View publication stats