CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2011)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1713

Zorilla: a peer-to-peer middleware for real-world distributed systems

Niels Drost*, †, Rob V. van Nieuwpoort, Jason Maassen, Frank J. Seinstra and Henri E. Bal

Department of Computer Science, VU University, Amsterdam, The Netherlands

SUMMARY

The inherent complex nature of current distributed computing architectures hinders the widespread adoption of these systems for mainstream use. In general, users have access to a highly heterogeneous set of compute resources, which may include clusters, grids, desktop grids, clouds, and other compute platforms. This heterogeneity is especially problematic when running parallel and distributed applications. Software is needed which easily combines as many resources as possible into one coherent computing platform.

In this paper, we introduce Zorilla: peer-to-peer (P2P) middleware that creates a single distributed environment from any available set of compute resources. Zorilla imposes minimal requirements on the resource 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. Zorilla explicitly supports distributed and parallel applications, and allows resources from multiple sites to cooperate in a single computation.

Zorilla makes extensive use of both virtualization and P2P techniques. We will demonstrate how virtualization and P2P combine into a simple design, while enhancing functionality and ease of use. Together, these techniques bring our goal a step closer: transparent, easy use of resources, even on very heterogeneous distributed systems. Copyright © 2011 John Wiley & Sons, Ltd.

Received 9 April 2010; Revised 5 January 2011; Accepted 16 January 2011

KEY WORDS: distributed computing; middleware; peer-to-peer; virtualization

1. INTRODUCTION

When grid computing was introduced over a decade ago, its goal was *efficient and transparent* (*i.e. easy-to-use*) *wall-socket computing over a distributed set of resources* [1]. This goal is sometimes referred to as the *promise of the grid*. Since then, other distributed computing paradigms have been introduced, including desktop grids, volunteer computing, and more recently cloud computing. All these share many of the goals of grid computing, ultimately trying to give end-users access to resources with as little effort as possible. These new distributed computing paradigms have led to a diverse collection of resources available to end-users. In general, users have simultaneous access to many different resources, with different paradigms, available software, access policies, etc. In this paper, we refer to such a heterogeneous set of resources as a *real-world distributed system* (see Figure 1).

Unfortunately, the emergence of real-world distributed systems has made the running of applications complex for end-users. The heterogeneity of these systems makes it difficult to install and

*Correspondence to: Niels Drost, Department of Computer Science, VU University, De Boelelaan 1081A, 1081 HV Amsterdam, The Netherlands.
†E-mail: niels@cs.vu.nl

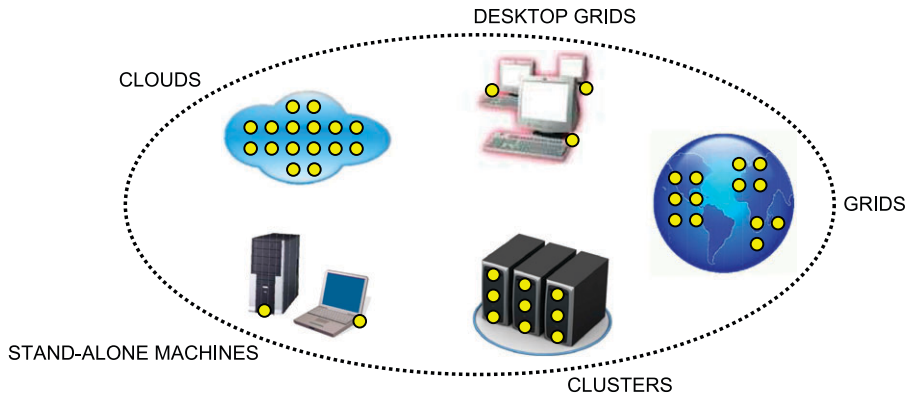

Figure 1. A worst-case real-world distributed system as perceived by end-users, simultaneously comprises clusters, grids, and clouds, as well as several other computing platforms. In general, a real-world distributed system is defined as an *ad hoc* collection of stand-alone compute resources (the yellow dots), each having a local memory, and each capable of communicating via a network protocol stack over a wired or wireless connection. Clusters, grids, and clouds, are administratively and semantically organized subsets of such a system, each provided with their own middleware, programming interfaces, access policies, and protection mechanisms.

run software on multiple resources, as each site requires configuring, compiling and possibly even porting the application to the specific resource. The 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.

Because of the above problems, usage of real-world distributed systems for high-performance computing is currently rather limited [2]. 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, global file systems, 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 [3], multimedia [4], and medical imaging [5]).

We argue that the ability to use all available resources transparently and simultaneously will greatly benefit users. Not all users have exclusive access to resources, and in our experience, no matter how powerful a single computer or cluster is, users will always desire more computational power. What is needed is a platform capable of turning *any* (possibly distributed) collection of resources into a single, homogeneous, and easy-to-use system.

This paper investigates middleware specially designed to run parallel and distributed applications 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 the existing grid and cloud middleware, where a more or less stable system is assumed. Also, middleware for real-world distributed systems is installed by users, and not system administrators. As a result, this middleware must be very easy to install, and does 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.

Although a real-world distributed system can comprised of any number of resources, typically, everyday scenarios will most likely be somewhat simpler. As an example use case, 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 [6] resources. Alternatively, the scientist can acquire additional resources

DOI: 10.1002/cpe

by deploying on a number of desktop machines. Unfortunately, it is impossible to predict exactly which types of resources are combined by users. As a result, middleware for real-world distributed systems must support all combinations of resources. In this paper, we will assume the worst-case scenario of *all* possible types of resources, ensuring that the resulting system is applicable in all possible scenarios.

Recently, cloud computing has emerged as a promising new computing platform. 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. Although originally designed to run mostly web servers, cloud computing is now also used as a high-performance computing platform [7, 8]. Virtualization of resources is an efficient way of solving the problem of heterogeneity in distributed systems today.

The use of virtualization allows for a simplified design of middleware for real-world distributed systems. 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 explore using Peer-to-Peer (P2P) techniques in 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 middleware.

In this paper, we introduce Zorilla: our P2P 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 that 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.

Zorilla is a prototype system, explicitly designed for running parallel and distributed 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, Zorilla focuses on this single use-case, and 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 [9, 10], and we consider integrating such systems as the future work.

The contributions of this paper are as follows:

- We establish the requirements of middleware for real-world distributed systems.
- We describe the design and implementation of Zorilla: a new lightweight, easy-to-use Java-based P2P middleware, explicitly designed for parallel and distributed applications.
- 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 middleware designed for real-world distributed systems brings closer the goal of easy-to-use distributed computing on these systems.
- We show the use of Zorilla in a large-scale application, concurrently using a variety of resources, including clusters, grids, desktop grids, and clouds. Although consisting of such a large number of different systems, our system is still able to perform computations efficiently, and handles faults automatically.

The research described in this paper is part of the Ibis project, which strives to make running parallel and distributed applications as easy as possible. Zorilla and other software referred to in this paper can be freely downloaded from the Ibis web site at `http://www.cs.vu.nl/ibis`.

The remainder of this paper is organized as follows. In Section 2 we define the requirements of middleware for real-world distributed systems. Section 3 gives a description of Zorilla, our P2P

DOI: 10.1002/cpe

middleware. Experiments are described in Section 4. We discuss the related work in Section 5. Finally, we conclude in Section 6.

2. REQUIREMENTS

In this section, we discuss the requirements of middleware for real-world distributed systems.

Resource independence: The primary function of a real-world distributed system 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, real-world distributed system middleware must be able to interface with this *local middleware*[‡]. The implementation of real-world distributed system 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 a real-world distributed system, it is difficult to guarantee that such machines are available: resources are not under the control of the user, and reliability is difficult to determine without the detailed knowledge of resources. Therefore, middleware should rely on as little central functionality as possible. Ideally, middleware uses no centralized components, and instead is implemented in a completely decentralized manner.

Malleability: In a real-world distributed system, 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 runtime of the programming model of the application , or in the application itself. Support for application-level fault-tolerance in the middleware can be limited to failure detection and reporting.

Easy deployment: Since a real-world distributed system is created *ad hoc* by end-users, middleware is typically deployed by the user, possibly for the duration of only a single experiment. Therefore, middleware for these systems needs to be as easy to deploy as possible. Complicated installation and setup procedures defeat the purpose of this middleware. 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 [11]. Parallel applications require scheduling of multiple (distributed) resources concurrently, tracking which resources are available [12], and providing reliable communication in the face of firewalls, and other problems [13].

Global file storage: Besides running applications, distributed systems 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, independent of applications, is also useful. Ideally, middleware should

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

DOI: 10.1002/cpe

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, security is important. Middleware must protect resources from users, as well as users from each other. Because of the heterogeneous nature of the resources in a real-world distributed system, 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 of middleware for real-world distributed systems presented above lead us to the conclusion that using existing techniques for implementing this 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 a real-world distributed system. Therefore, our middleware, discussed in the following section, uses a number of alternative approaches for implementing functionality.

3. ZORILLA

In this section, we describe the design of Zorilla, our *prototype* P2P middleware. We will first give an overview of Zorilla, followed by a more detailed discussion of the selected functionality. The main purpose of Zorilla is to facilitate running parallel and distributed applications remotely on any resource available. We refer to a single instance of an application as a *job*.

The design and implementation of Zorilla is rather different from a typical middleware. Instead of using *bare* resources, it builds on existing infrastructure, and is specifically kept lightweight. Therefore, rather than explaining the design of Zorilla using the typical separation in layers or modules, we will describe Zorilla using the steps required to run a job: discovering resources, scheduling, deploying, and managing the running job.

Zorilla relies heavily on P2P techniques to implement functionality. P2P techniques have proved very successful in the 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 difficult 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). This single overlay network connects all nodes in the entire system, both within a site, as well as in different sites. Each node in the system is completely independent, 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 3 for an overview of the life cycle of a (parallel) job in a Zorilla system.

Zorilla has been explicitly designed to fulfill the requirements we established in Section 2. Table I shows an overview of the requirements, and how Zorilla adheres to these. As said, 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 [15].

DOI: 10.1002/cpe

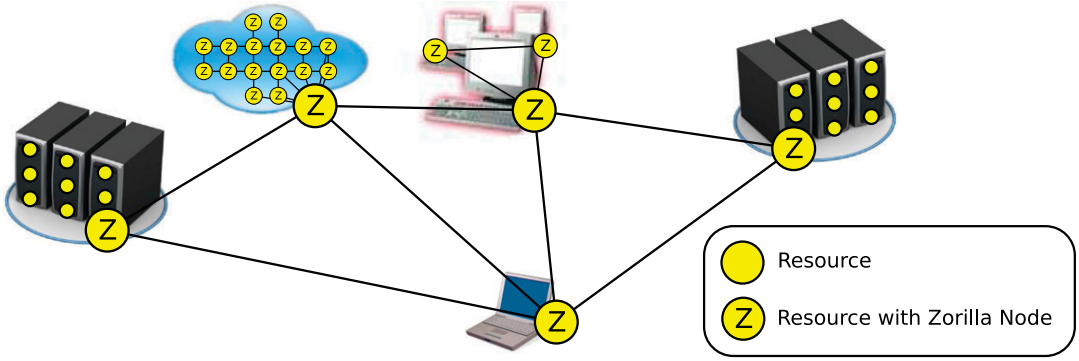

Figure 2. Example of a compute system created by Zorilla. This system consists of two clusters, a desktop grid, a laptop, as well as cloud resources (for instance acquired via Amazon EC2). On the clusters, a Zorilla node is run on the headnode, and Zorilla interacts with the local resources via the localscheduler. 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.

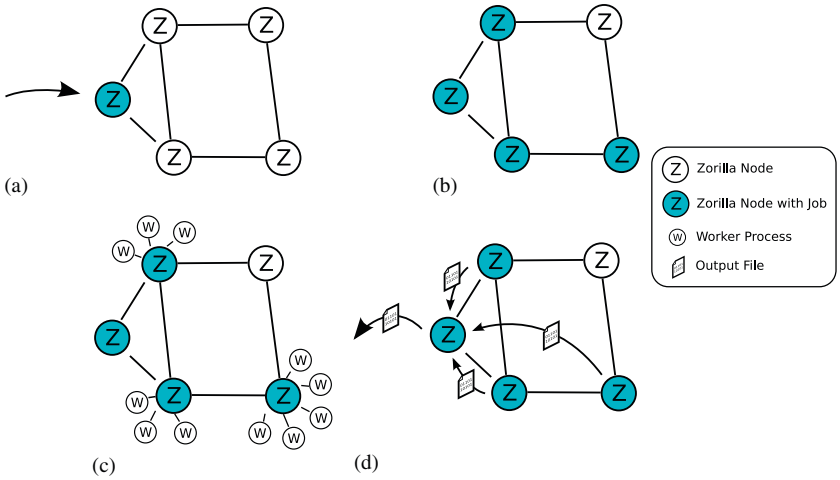

Figure 3. Job life cycle in a Zorilla system consisting of 5 nodes connected through an overlay network: (a) a (parallel) job is submitted by the user to a node; (b) the job is disseminated to other nodes; (c) local schedulers at each node decide to participate in the job, and start one or more Worker processes (e.g. one per processor core available); and (d) output files are copied back to the originating node.

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 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. Moreover, since Zorilla is deployed *ad hoc*, it is not possible to change the middleware in any way.

We use a generic API to interface to resources, in this case the JavaGAT [14]. The JavaGAT allows 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 [16]. 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*.

DOI: 10.1002/cpe

Table I. Design overview of Zorilla.

Requirement	Approach		Solution in Zorilla
	P2P	Virtualization	
Resource independence		X	JVM, VirtualBox
Middleware independence		X	JavaGAT [14]
Decentralization	X		P2P implementations of functionality
Malleability	X		Replacement resources allocated if needed
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, SmartSockets [13], JEL [12]
Global file storage	X		Per-job files only
Security		X	Sandboxing of applications

Listed are the requirements, the approach used to address the issue (be it virtualization or P2P), and how this affects the design of zorilla.

The P2P design of Zorilla allows it to fulfill a number of the requirements of Table I. 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 set up 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. Another benefit of using both P2P and virtualization is that it allows Zorilla to *support parallel applications*. Zorilla explicitly allows parallel and distributed applications by supporting applications which span multiple resources, and optimizing scheduling of these resources (see Section 3.1). Besides scheduling, parallel applications are also supported by offering reliable communication by way of SmartSockets (see Section 3.2), and resource tracking in the form of our JEL model (see Section 3.3).

Zorilla supports files when running applications. Executables, virtual machine images, input files, and output 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 the 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 difficult to implement a reliable authentication system using only P2P techniques, one alternative is to integrate support for the Grid Security Infrastructure (GSI) [17] also used in Globus into Zorilla.

The main goal of our research is to make running applications on real-world distributed systems as easy as possible. Therefore, the impact of our software on applications should be as little as possible. Table II lists the impact Zorilla has on applications. It shows that applications are mostly unaffected. The biggest impact for applications is the fact that Zorilla can provide applications, or the runtime of the programming model of the application, with information regarding failures, and changes to the resources available. To make full use of the features of Zorilla, the application or runtime of its programming model must support reacting to this information provided by the system.

DOI: 10.1002/cpe

Table II. Impact of the design of Zorilla on applications.

Requirement	Impact on applications
Resource independence	None, any application can be run
Middleware independence	None
Decentralization	None
Malleability	Applications preferable need to be able to handle changes in resources
System-level fault-tolerance	Programming models and applications can detect and react to faults
Easy deployment	None
Parallel application support	Applications can now run on large scale, dynamic systems
Global file storage	Applications can use input and output files normally
Security	Applications cannot read and write to anywhere but the sandbox provided

3.1. 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 1 GB of main memory. However, machines with over 16 GB 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 mentioned, 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 is also important. A parallel application may send and receive a large amount of data during its lifetime, hence high bandwidth connections between the resources are required. Also, most parallel applications are sensitive to the latency between resources used.

DOI: 10.1002/cpe

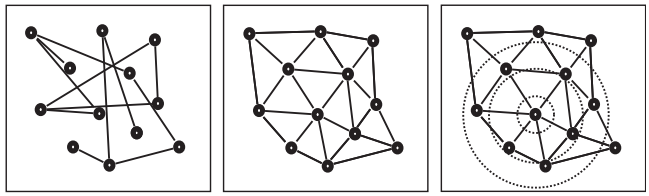

Figure 4. Resource discovery in Zorilla: (a) random overlay created for resource discovery; (b) neighbor connections created; and (c) flood scheduling (iterative ring search) performed using neighbors to schedule resources.

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 4). First, a P2P overlay network consisting of all Zorilla nodes is built up. Second, the P2P overlay is 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 (a previous version of) our system see [18].

Zorilla's overlay network is based on the ARRG [19] *gossiping* algorithm. ARRG provides a *peer sampling service* [20] 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 about 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 4(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 [21] synthetic coordinate system. Vivaldi assigns coordinates in a Cartesian space to each node of a P2P overlay. Coordinates are assigned to reflect the round trip latency between nodes. Given two Vivaldi coordinates, the distance between these 2 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 4(b)).

Once a suitable P2P network and neighbor list is built, this is then used for the actual allocation of resources to jobs. When a job is submitted at a node, Zorilla's *flood scheduling* [18] algorithm 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 4(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 [22] 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 optimize 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

DOI: 10.1002/cpe

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.

3.2. 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, if possible. As a fallback, the submitting node is used when the file is not present at any peer.

Although using peers to transfer input files, executables and VM images greatly improve scalability, there can still be a significant amount of traffic and time required to get all files to all nodes, especially if large input files, or a large VM image is required to launch jobs. In the latter case, reducing the size of the VM image can help, for instance by using a minimal distribution. As the future work, we plan to further reduce this problem by integrating a global, persistent filesystem in Zorilla. Then, input files and VM images can be stored permanently in this filesystem, reused for each job, and cached at nodes locally.

While running a job, it is important that all nodes used to run this job can communicate reliably. 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 [13] overlay network. This overlay network is used by Zorilla to route traffic over, if needed. Besides Zorilla itself, this overlay can also be used by applications. This ensures reliable communication between all resources used, regardless of NAT and Firewalls.

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 [15]. For Java, this is simply done by invoking the *java* command with the right parameters. For OVF, this image is first imported to the local VirtualBox environment, and subsequently started.

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 3.1. 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.

3.3. 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*,

DOI: 10.1002/cpe

running, or *finished*. Zorilla 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 a real-world distributed system 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 [23], or our Join-Elect-Leave (JEL) [12] model. 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 difficult to do efficiently and reliably, and regard it as the future work.

4. EXPERIMENTS

After discussing the design of Zorilla, we will now illustrate its use by running a number of experiments. We will focus on high-level experiments showing the functioning of Zorilla as a whole. Various subsystems and techniques used in Zorilla are evaluated in other work, including JavaGAT [14], SmartSockets [13], JEL [12], ARRG [19], and the scheduling subsystem of Zorilla [18] These papers also evaluate the scalability of the various techniques.

For the experiments in this paper we use the Distributed ASCI Supercomputer 3 (DAS-3), a five-cluster distributed system located in The Netherlands. We use an additional cluster in Chicago, an Amazon EC2 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 the introduction. See Table III for an overview of all resources used.

On the headnode of each cluster used in the experiment, a Zorilla node is started. This node is then responsible for managing the resources in that cluster. 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 cloud, the desktop grid, and the stand-alone machine

Table III. Sites used in the world-wide experiment.

Location	Country	Type	Middleware	Nodes	Cores
VU University, Amsterdam	The Netherlands	Grid (DAS-3)	SGE	16	64
University of Amsterdam			Globus	16	64
Leiden University			prun	8	16
MultimediaN, Amsterdam			prun	16	32
EVL, Chicago	U.S.A.	Cluster	SSH	16	16
VU University, Amsterdam	The Netherlands	Desktop Grid	—	2	4
Amazon EC2	U.S.A.	Cloud	—	8	8
VU University, Amsterdam	The Netherlands	Desktop	—	1	1
			Total	83	205

All sites run Zorilla on the front-end. 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.

DOI: 10.1002/cpe

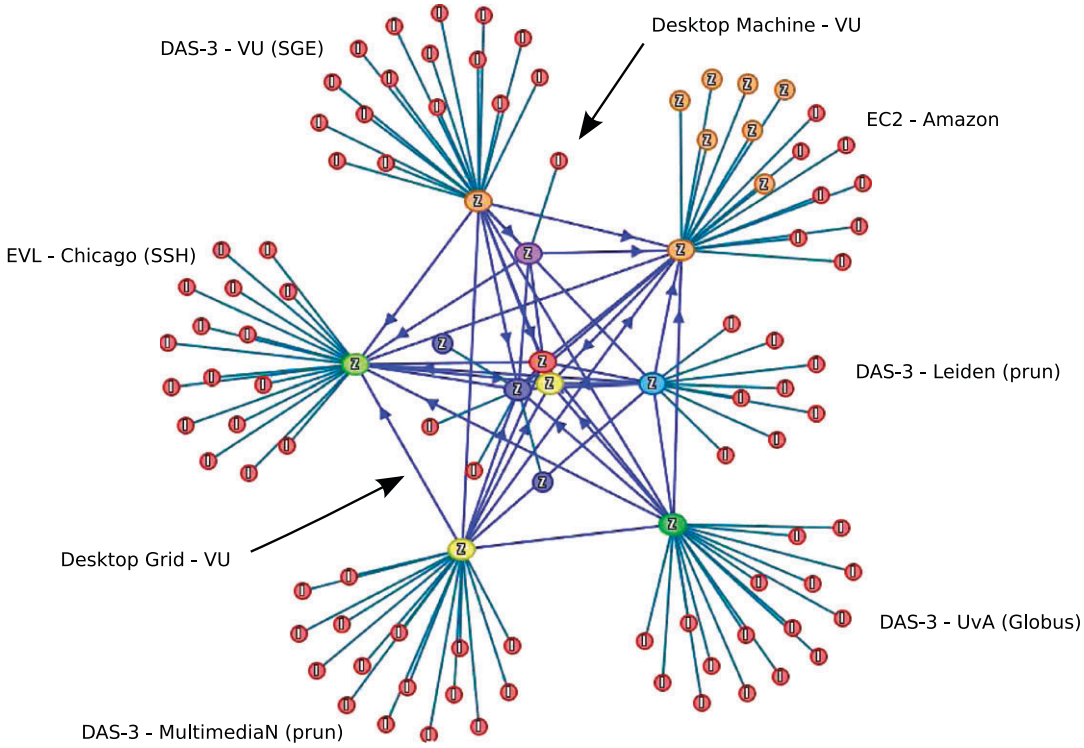

Figure 5. Resources used in the world-wide experiment. This visualization shows all nodes, and the SmartSockets overlay network between them. A node marked *Z* represents a Zorilla node, running on either a front-end or a resource. A node marked *I* represents an instance of the Go application.

Zorilla is started on each machine individually. After startup, all Zorilla nodes form one large distributed system.

As an application we used 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. 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 [24] programming model. It is implemented using the IPL [25] communication library, which in turn uses JEL to track the resources available, and SmartSockets 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.

As a first experiment, we deployed the Go application on the entire instant cloud by submitting it to the Zorilla node on the local desktop machine. See Figure 5 for an overview of all nodes used, and the SmartSockets overlay network. Zorilla deployed the application on 83 nodes, with over 200 cores. The applications achieved 87% efficiency overall, ranging from 72% on the poorly connected cluster in Chicago, to over 90% on machines in the DAS system. This experiment 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 Zorilla prefers close-by resources 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, the number of nodes used in the computation drops from 40 to 29.

DOI: 10.1002/cpe

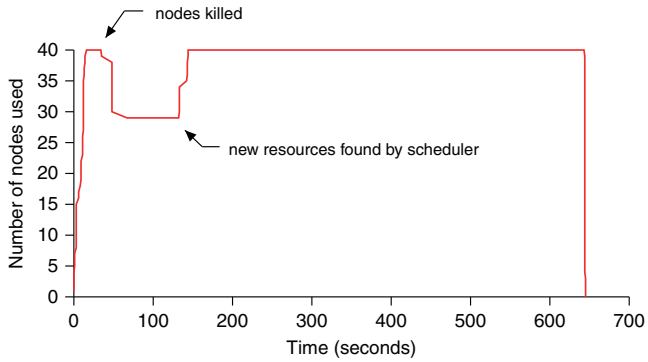

Figure 6. Fail test, where a job requiring 40 nodes was submitted. Zorilla automatically compensates when nodes fail by acquiring new resources.

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 dramatically increases the efficiency of the application, resulting in an average efficiency of 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.

5. RELATED WORK

There are a number of projects that share at least some of the goals and techniques of Zorilla. One type of related system is *Wide Area Overlays of virtual Workstations*, or WOW [26, 27], which creates a single system out of independent resources, best described as a *virtual cluster*. This cluster is created using VMware virtual machines, and an overlay network between all nodes. Applications do not need to be aware that the system they run on is actually a virtual machine. The drawback of this approach is that all traffic is routed over the overlay network, limiting performance. In Zorilla, traffic is only routed using the overlay network if needed, while direct connections are used if possible, leading to *native* network speeds. Also, Zorilla supports the more lightweight Java Virtual Machine rather than only VMWare. Because of these limitations, WOWs are described as a platform for high throughput computing. Zorilla, on the other hand, explicitly supports high performance, parallel applications.

ProActive [28] is another system that, like Zorilla, strives to use Java and P2P techniques [29] to run high-performance computations on distributed systems. However, Proactive primarily supports applications that 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.

Also related Zorilla are cloud middleware, including Amazon EC2 [6], Eucalyptus [30], and Globus Nimbus [31]. All these middleware are designed to turn a number of machines into a single coherent system. One difference to Zorilla is the fact that these middleware assume that no other middleware is present, while zorilla can also run on top of other middleware. Also, these middleware all have centralized components, while zorilla is complete decentralized, and these systems are assumed to be installed (semi) permanently by system administrators, while Zorilla

DOI: 10.1002/cpe

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 targeted to run 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 [31], OpenNebula [32], and InterGrid [33].

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 [34–36] 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.

Another approach to creating a system spanning multiple resources is used in the InterGrid [33] project. Here, gateways are installed that 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 set up in advance by system administrators.

6. CONCLUSIONS AND FUTURE WORK

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 paper, we suggest a possible solution for these problems: middleware explicitly designed for real-world distributed systems. We established the requirements of such a middleware, including fault-tolerance, platform independence, and support for parallel applications.

We introduce Zorilla, a prototype P2P middleware designed for creating a single, coherent system out of any available resources, including stand-alone machines, clusters, grids, and clouds, used concurrently, and running parallel and distributed 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.

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 comprising grids, clusters, clouds, desktop grids, and other systems.

Although we have shown Zorilla to be highly useful, it is not yet complete. The future work includes adding a global filesystem for long-term storage and security in the form of GSI.

ACKNOWLEDGEMENTS

This work was carried out in the context of the Virtual Laboratory for e-Science project (www.vl-e.nl). This project is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W) and is part of the ICT innovation program of the Ministry of Economic Affairs (EZ). This work has been co-funded by the Netherlands Organization for Scientific Research (NWO) grant 612.060.214 (Ibis: a Java-based grid programming environment).

We would like to thank the people of the Electronic Visualization Laboratory for access to their systems, and thank Amazon for providing academic EC2 credits. We kindly thank Ceriel Jacobs for all his help. We also like to thank the anonymous reviewers for their insightful and constructive comments.

DOI: 10.1002/cpe

REFERENCES

1. Foster I, Kesselman C, Tuecke S. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications* 2001; **15**(3):200–222.
2. Butler D. The petaflop challenge. *Nature* 2007; **448**:6–7.
3. Gualandris A, Zwart SP, Tirado-Ramos A. Performance analysis of direct *n*-body algorithms for astrophysical simulations on distributed systems. *Parallel Computing* 2007; **33**:159–173.
4. Seinstra FJ, Geusebroek J-M, Koelma D, Snoek CGM, Worring M, Smeulders AWM. High-performance distributed video content analysis with parallel-Horus. *IEEE Multimedia* 2007; **14**(4):64–75.
5. Montagnat J, Breton V, Magnin IE. Using grid technologies to face medical image analysis challenges. *Proceedings of the Third International Symposium on Cluster Computing and the Grid CCGRID '03*. IEEE Computer Society: Washington, DC, U.S.A., 2003; 588.
6. Amazon ec2 website. Available at: http://aws.amazon.com/ec2 [5 January 2011].
7. Huang W, Liu J, Abali B, Panda DK. A case for high performance computing with virtual machines. *ICS '06*: *Proceedings of the 20th Annual International Conference on Supercomputing*. ACM: New York, NY, U.S.A., 2006; 125–134.
8. Penguin computing on demand website. Available at: http://www.penguincomputing.com/POD [5 January 2011].
9. Muthitacharoen A, Morris R, Gil TM, Chen B. Ivy: A read/write peer-to-peer file system. *SIGOPS Operating Systems Review* 2002; **36**:31–44.
10. Stribling J, Sovran Y, Zhang I, Pretzer X, Li J, Kaashoek MF, Morris R. Flexible, wide-area storage for distributed systems with wheelFS. *Proceedings of the Sixth USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association: Dummy, Berkeley, CA, U.S.A., 2009; 43–58.
11. Bal HE, Drost N, Kemp R, Maassen J, van Nieuwpoort RV, van Reeuwijk C, Seinstra FJ. Ibis real-world problem solving using real-world grids. *IPDPS '09*: *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*. IEEE Computer Society: Washington, DC, U.S.A., 2009; 1–8.
12. Drost N, van Nieuwpoort RV, Maassen J, Seinstra F, Bal HE. JEL: Unified resource tracking for parallel and distributed applications. *Concurrency and Computation*: *Practice and Experience* 2010; **23**(1):17–37.
13. Maassen J, Bal HE. SmartSockets: Solving the connectivity problems in grid computing. *HPDC '07*: *Proceedings of the 16th International Symposium on High Performance Distributed Computing*. ACM: New York, NY, U.S.A., 2007; 1–10.
14. Nieuwpoort R, Kielmann T, Bal HE. User-friendly and reliable grid computing based on imperfect middleware. *SC '07*: *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. ACM: New York, NY, U.S.A., 2007; 1–11.
15. Virtualbox website. Available at: http://www.virtualbox.org/ [5 January 2011].
16. Goodale T, Jha S, Kaiser H, Kielmann T, Kleijer P, von Laszewski G, Lee C, Merzky A, Rajic H, Shalf J. SAGA: A simple API for grid applications, high-level application programming on the grid. *Computational Methods in Science and Technology* 2006; **12**(1):7–20.
17. Foster I, Kesselman C, Tsudik G, Tuecke S. A security architecture for computational grids. *CCS '98*: *Proceedings of the Fifth ACM Conference on Computer and Communications Security*. ACM: New York, NY, U.S.A., 1998; 83–92.
18. Drost N, van Nieuwpoort RV, Bal HE. Simple locality-aware co-allocation in peer-to-peer supercomputing. *Sixth International Workshop on Global and Peer-2-Peer Computing* (*GP2P 2006*), Singapore, May 2006.
19. Drost N, Ogston E, van Nieuwpoort RV, Bal HE. ARRG: Real-world gossiping. *Proceedings of the 16th IEEE International Symposium on High-performance Distributed Computing* (*HPDC*), Monterey, CA, U.S.A., June 2007.
20. Jelasity M, Guerraoui R, Kermarrec A-M, van Steen M. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. *Middleware '04*: *Proceedings of the Fifth ACM/IFIP/USENIX International Conference on Middleware*. Springer: New York, NY, U.S.A., 2004; 79–98.
21. Dabek F, Cox R, Kaashoek F, Morris R. Vivaldi: A decentralized network coordinate system. *SIGCOMM Computer Communication Review* 2004; **34**:15–26.
22. Gnutella. The gnutella protocol specification. Available at: http://rfc-gnutella.sourceforge.net [5 January 2011].
23. Fagg GE, Gabriel E, Bosilca G, Angskun T, Chen Z, Pjesivac-Grbovic J, London K, Dongarra JJ. Extending the MPI specification for process fault tolerance on high performance computing systems. *Proceedings of ICS'04*, June 2004.
24. Nieuwpoort R, Wrzesinska G, Jacobs CJ, Bal HE. Satin: A high-level and efficient grid programming model. *ACM Transactions on Programming Languages and Systems* (*TOPLAS*) 2010; **32**(3):1–39.
25. Nieuwpoort R, Maassen J, Wrzesińska G, Hofman RFH, Jacobs CJH, Kielmann T, Bal HE. Ibis: A flexible and efficient Java-based grid programming environment. *Concurrency and Computation*: *Practice and Experience* 2005; **17**(7–8):1079–1107.
26. Ganguly A, Agrawal A, Boykin P, Figueiredo R. Wow: Self-organizing wide area overlay networks of virtual workstations. *Journal of Grid Computing* 2007; **5**:151–172. DOI: 10.1007/s10723-007-9076-6.
27. Wolinsky D, Figueiredo R. Simplifying resource sharing in voluntary grid computing with the grid appliance. *IEEE International Symposium on Parallel and Distributed Processing*, *2008* (*IPDPS 2008*), April 2008; 1–8.
28. Baduel L, Baude F, Caromel D, Contes A, Huet F, Morel M, Quilici R. Programming, deploying, composing, for the grid. *Grid Computing*: *Software Environments and Tools*. Springer: Berlin, 2006.

DOI: 10.1002/cpe

29. Caromel D, Costanzo Ad, Mathieu C. Peer-to-peer for computational grids: Mixing clusters and desktop machines. *Parallel Computing* 2007; **33**(4–5):275–288.
30. Nurmi D, Wolski R, Grzegorczyk C, Obertelli G, Soman S, Youseff L, Zagorodnov D. The Eucalyptus open-source cloud-computing system. *CCGRID '09*: *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE Computer Society: Washington, DC, U.S.A., 2009; 124–131.
31. Nimbus science cloud. Available at: http://www.nimbusproject.org [5 January 2011].
32. Sotomayor B, Montero RS, Llorente IM, Foster I. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing* 2009; **13**(5):14–22.
33. di Costanzo A, de Assuncao MD, Buyya R. Harnessing cloud technologies for a virtualized distributed computing infrastructure. *IEEE Internet Computing* 2009; **13**(5):24–33.
34. Abbes H, Crin C. A decentralized and fault-tolerant desktop grid system for distributed applications. *Concurrency and Computation*: *Practice and Experience* 2010; **22**(3):261–277.
35. Butt AR, Zhang R, Hu YC. A self-organizing flock of condors. *SC '03*: *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society: Washington, DC, U.S.A., 2003; 42.
36. Chandra A, Weissman J. Nebulas: Using distributed voluntary resources to build clouds. *HotCloud '09*, San Diego, CA, 2009.

DOI: 10.1002/cpe