Summary

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.