Highly Available and Scalable Grid Services

Guillaume Pierre
VU University Amsterdam
gpierre@cs.vu.nl

Thorsten Schütt
Zuse Institute Berlin
schuett@zib.de

Jörg Domaschka
Ulm University
joerg.domaschka@uni-ulm.de

Massimo Coppola
ISTI - CNR
massimo.coppola@isti.cnr.it

1. INTRODUCTION

Grid computing infrastructures create many new challenges related to data management. Grids are typically deployed at a large scale, and one can only expect this scale to grow even more in terms of number of machines, locations and administrative domains.

Researchers who build Grid computing infrastructures constantly face scalability issues, both at the system-level of the Grid's internals and at the user-level to support complex end-user applications. The XtreemOS project is no exception to this rule: XtreemOS is an E.U. research project that aims at "Building and Promoting a Linux-based Operating System to Support Virtual Organizations for Next Generation Grids" [1].

One of the explicit goals of the project is to address scalability issues by abstracting these issues as much as possible from the internal implementation of the system. This is an ambitious goal, since in the most general case scalability issues must be addressed within each component of a distributed system [6]. We however found that, to a certain extent, it is possible to build scalable abstractions that allow other programmers to build system-level and application-level functionality while giving less thought to scalability issues.

This paper discusses a number of highly available and scalable Grid services that are being built to support this idea. They can be classified in three separate categories: (i) services to store/query structured data in a scalable fashion; (ii) services to communicate in a scalable fashion; (iii) services to (partially) hide the effects of scale. The next sections discuss these three types of services separately, then conclude this paper.

2. STRUCTURED DATA REPOSITORIES

One of the major functions where a Grid infrastructure faces serious scalability issues is job scheduling. Job scheduling consists of identifying a group of machines out of the entire Grid that are collectively suitable and willing to execute a new task. Current Grid scheduling algorithms use a hierarchical structure where each location is represented by a local scheduler, and a meta-scheduler issues scheduling requests to the local schedulers. In a truly scalable system, however, the number of locations may simply prevent one from doing this. Instead, XtreemOS has chosen a three-step approach. First, resources are selected with respect to static job attributes such as the desired CPU family, hardware specifications, required libraries, etc. The list of suitable machines is then potentially refined according to dynamically changing node attributes – their current load, available disk space, etc. Finally, the actual scheduling and negotiation for accessing resources can be issued within a set of pre-selected machines that is orders of magnitudes smaller than the entire Grid. We focus here on the first two steps, and leave the actual resource scheduling for a further discussion.

2.1 Resource Selection based on static attributes

The goal of the resource selection service is to answer queries for a list of machines that match a number of static attributes. Queries may specify a specific value for a supported parameter (e.g., the job must run on AMD64 CPU architectures), a range (e.g., the job requires any amount of memory greater than 1 GB), or none at all (e.g., the job does not depend on any version of library XYZ). To support these queries in a scalable fashion, we organized all Grid nodes in a peer-to-peer overlay dedicated to supporting such queries.

All peer-to-peer approaches to similar problems rely on *delegation*, where compute nodes register their attributes to registry nodes that implement the lookup functionality. We however consider that this approach should be avoided: (i) it creates unnecessary load on the system due to the periodic revalidations of the registered values and the need to check node availability regularly; (ii) it creates inconsistency between the actual and registered attribute values, for example, in the case of a failure of a compute node or its corresponding registry node(s); (iii) it creates imbalanced workloads, requiring extra effort to balance.

We instead proposed an approach where each node represents its own attributes in the overlay. This means for example that a node failure need not be accurately detected by other nodes, and that no specific repair operation is necessary to reconstitute the overlay. We proved using simulations and actual deployments that our approach scales well with the number of nodes as well as the number of supported attributes. For more information we refer the reader to [2].

2.2 Application Directory Service for nodes dynamic attributes

Next to static attributes that barely change over time, nodes may be characterized by volatile attributes such as the current number of jobs they are involved in and the amount of free space in their hard drives (by opposition to the total size of the installed hard drives of the machine). Job submission requests may of course contain requirements regarding these attributes. Similarly, they may place requests about *groups* of nodes that are difficult to address in the resource selection service. For example, one may request that machines are located in different data centers to amximize the job's resilience to failures.

We decided to build this second level of resource selection using a scalable directory service where nodes can register their properties. Additionally, the same scalable directory service is also a useful abstraction to store system-level information regarding jobs, users, administrative domains, and so on.

Our directory service implements its functionality on top of a set of DHT overlays. Currently we support both OverlayWeaver and Scalaris (which is described below). This allows one to selectively benefit from the good properties of different DHTs. For instance, some applications may exploit the Scalaris transactional query semantic, while OverlayWeaver supports higher levels of security thanks to the use of Kademlia protocols.

Dynamic node attributes are periodically updated into the system. Structured information entered in the system is attached at least to a primary key, which acts as the item'ss key in the DHT. By combining a different choice of the DHT implementation, and different store/query algorithms executed on top of the DHT layer, more complex query semantics like range query and multi-attribute queries are achieved. As separate clients of the directory service are also allowed to transparently share a same DHT, the approach allows a user-defined QoS trade off to be set up with bounded overhead.

Although many of the ideas exploited are relatively classical, combining all of them in a single structure raises a number of interesting research questions. One example is determining the optimal number of replicas that each dynamically changing item stored in the DHT should have, given that different items can be refreshed at different rates and exploiting opportunistic optimizations, thereby incurring different replication costs.

3. SCALABLE COMMUNICATIONS

Another critical feature in a large-scale Grid infrastructure is to allow scalable communications between entities participating in the same task. However, given the expected scale of the Grid and the churn that one must expect in such environments, one should not rely on point-to-point communications to efficiently spread information. We instead focus on the publish-subscribe paradigm, which allows to decouple the information producers from the consumers. A pub/sub system aims at bridging the gap by allowing consumers to register their interests, and by routing information to the corresponding consumers. The main goal here is to provide strong data consistency in the face of node crashes and hefty concurrent read and write accesses.

Our Pub/Sub system, called Scalaris, comprises four layers. At the bottom is a DHT which provides a simple put and get interface to a dictionary-like data structure which is distributed over all participating nodes. In contrast to many other overlays, our implementation stores the keys in lexicographical order. Lexicographical ordering instead of random hashing enables control of data placement which is necessary to support range queries efficiently in the DHT. The DHT provides scalability and fault-tolerance. The second layer implements so-called symmetric replication which guarantees the availability of data even when nodes fail or are unavailable. Symmetric replication divides all nodes into r equivalence classes, and distributes the replica so that the nodes storing the replicas of item belong to different equivalence classes. On top of the replication layer, the third layer implements transaction data accesses, where all read and write operations are performed inside transactions with ACID properties. The transactions allow us to consistently update all replicas belonging to one item and at the same update several items in one atomic operation. This layer is particularly innovative compared to the state of the art in the domain. The transaction framework employs Paxos. The final layer is the pub/sub system itself, which uses the layers below for managing subscribers and topics. The maximum system throughput has been show to scale linearly with the number of nodes. For more information we refer the reader to [8, 7]. Scalaris also won the 1st price (shared) at the IEEE Scalable Computing Challenge 2008.

4. HIDE THE EFFECTS OF SCALE

In a Grid environment, not all services can or should be implemented in a decentralized fashion. For example, one can make a strong case against the full decentralization of a cryptographic key management system. Similarly, some services have too complex functionalities to be able to benefit from a decentralized implementation. Although these services should therefore remain largely centralized, they may suffer from issues related to the scale of the environment they run in. First of all, many of these services should be implemented in a fault-tolerant manner so as to provide continuous service across node or network failures. Second, many services can make use of replication to improve their performance in the presence of heavy workloads. In both cases, the fact that a service physically runs on a collection of nodes needs to be hidden from the users of this service, even in the case of a change in the list of machines collectively providing a service. We first discuss how transparent fault-tolerance can be achieved, then how we can hide the server group composition to its clients.

4.1 Virtual nodes

The goal of virtual nodes is to provide fault-tolerance functionality to service-oriented applications with minimum effort for the service developers. We aim at (i) maximizing the reusability of existing replication-unaware code by (ii) making replication issues as transparent as possible to the service developer. For this reason, virtual nodes are implemented as a middleware layer, akin to an RMI or SOAP container. Instead of instantiating a single server object in a single server machine, virtual nodes create a group of logically equivalent server objects on several machines. The middleware makes sure to maintain the consistency of the replicated server object when incoming requests may modify the internal state.

The only assumption about a service is that it is a passive entity whose execution is triggered only by incoming requests, and that this execution is deterministic. This means in practice that issuing the same request to a consistent set of replicas will leave replicas in another consistent state. Services may have state and may be invoked concurrently. Notably supporting this last feature is a challenge, since the concurrent execution of multiple deterministic requests may not be deterministic itself. We solve this problem by applying deterministic thread scheduling strategies that ensure the determinism of concurrent request invocation.

Virtual nodes allow a programmer to choose a replication policy among several available ones, comprising several flavors of passive and active replication. For more information we refer the reader to [4].

4.2 Distributed servers

One important issue when using virtual nodes or any other system that implements a single service abstraction over a distributed group of machines is to give its user a simple contact address where queries can be sent. This is the goal of Distributed Servers: a distributed server is an abstraction that allows a group of server processes to appear as a single entity, with a single IP address, to its clients. Distributed Servers aim at allowing high-performance client-to-server communication, while being totally transparent to the clients. The only requirement is that the clients support the Mobile IPv6 protocol [5].

A distributed server address is simply an IPv6 address provided for the network service by other name resolution techniques such as Domain Name System (DNS). For example using the distributed server address, a client first connects to a contact node. Subsequently, a client connection may be transparently handed off – the server endpoint of the connection may be transferred – to different servers to effect load balancing or for client-specific processing. With the use of Mobile IPv6 (MIPv6) route optimization, this handoff provides a direct connection to the new server with corresponding network efficiencies. Further data are not routed through the contact node. This ability to migrate client connections without modification of the client depends on the implementation of Mobile IPv6, which typically supports client mobility. However, Distributed Servers inverts client mobility to provide the appearance of server mobility.

Distributed servers allows to handoff connections to other server nodes at the time a new TCP connection is established, but also at later stages of an existing connection. For example, a server node willing to shutdown for repairs may handoff all the connections it is currently serving to other nodes so that its shutdown does not impact the client experience with the service. For more information we refer the reader to [9, 3].

4.3 Integrating virtual nodes with distributed servers

Virtual nodes and distributed servers are highly complementary to each other: virtual nodes provide fault-tolerant replication that is mostly transparent to the service developer, but lacks an access method that makes fault-tolerance transparent to the clients; distributed servers provide a solution for making the service replication transparent to the client. Although each service has its own utility when used in isolation, we see that merging both systems would in principle allow one to build fault-tolerant replicated services where the complexity of replication would be transparent to both the service developer, and to the client-side application. Such a system could thus rely on open access protocols such as HTTP/SOAP.

Merging these two different services into a single whole is however not a trivial task, mainly for two reasons. First, although both services rely on some membership protocol to give a consistent view of currently available replicas to all parties, the two services have different requirements: virtual nodes need to be conservative before declaring a node as failed, because the cost of a false positive is potentially very high; on the other hand, distributed servers should detect node failures much more aggressively to present a continuously available service to the clients.

Second, the semantic of failure recovery in virtual nodes and distributed servers is not exactly the same, which implies that certain corner cases of node failures cannot be recovered transparently to the clients. For example, in the case a server node fails while processing a request, a backup server node can recover the client connection to the crashed server but may lack sufficient information about the request to restart it transparently. In this case a proper exception reply can be sent to the client (instead of letting the dangling connection reach a timeout value if we would not recover the connection). We assume that on receipt of an exception, the request can be quickly retried by the client.

5. CONCLUSION

As any large-scale distributed system, a Grid computing platform needs to explicitly address a number of scalability and high availability issues. Instead of addressing each such issue separately in an ad hoc fashion, the XtreemOS project has chosen to treat scalability and availability as first class objects by building generic abstractions that allow other parts of the system to benefit from good properties with no significant increase of complexity. We first build peer-to-peer overlays to handle scalable and reliable structured data management, in particular to support scalable job scheduling within the Grid. Then, we build generic functionality that can be used by any other system-level or application-level programmer willing to benefit from the said properties.

Two services have already been made publicly available: the resource selection service, as part of the first release of the XtreemOS system [10], and Scalaris as an independent peer-to-peer distributed key-value store [7]. Other services are expected to be part of the next release of XtreemOS, and to contribute availability and scalability properties to this new operating system for grid computing.

6. REFERENCES

[1] T. Cortes, C. Franke, Y. Jégou, T. Kielmann, B. Matthews, C. Morin, L. P. Prieto, and A. Reinefeld. XtreemOS: a vision for a Grid operating system. Technical Report #4, XtreemOS, May 2008. `http://www.xtreemos.org/publications/techreports/xtreemos-visionpaper-1.pdf`.

[2] P. Costa, J. Napper, G. Pierre, and M. van Steen. Autonomous resource selection for decentralized utility computing. In *Proc. 29th Intl Conference on Distributed Computing Systems (ICDCS)*, June 2009.

[3] Distributed servers demonstration. `http://www.cs.vu.nl/~gpierre/mipv6/`.

[4] J. Domaschka, H. P. Reiser, and F. J. Hauck. Towards generic and middleware-independent support for replicated, distributed objects. In *Proc. 1st workshop on Middleware-application interaction*, 2007.

[5] D. Johnson, C. Perkins, and J. Arkko. Mobility Support in IPv6. RFC 3775, June 2004.

[6] B. C. Neuman. Scale in distributed systems. In *Readings in Distributed Computing Systems*, pages 463–489. IEEE Computer Society Press, 1994.

[7] Scalaris – a distributed transactional key-value store. `http://code.google.com/p/scalaris/`.

[8] T. M. Shafaat, M. Moser, A. Ghodsi, T. Schütt, S. Haridi, and A. Reinefeld. Key-based consistency and availability in structured overlay networks. In *Proc. Intl. ICST Conference on Scalable Information Systems*, June 2008.

[9] M. Szymaniak, G. Pierre, M. Simons-Nikolova, and M. van Steen. Enabling service adaptability with versatile anycast. *Concurrency and Computation: Practice and Experience*, 19(13):1837–1863, Sept. 2007.

[10] Xtreemos-1.0. `http://www.xtreemos.eu/software/downloading`.