

A Framework for Self-Optimizing Grids Using P2P Components

Florian Schintke, Thorsten Schütt, Alexander Reinefeld

Zuse Institute Berlin (ZIB)

E-mail: {schintke,schuett,reinefeld}@zib.de

Abstract

We present the framework of a new grid architecture based on the peer-to-peer and the component paradigms. In our architecture, several peer-to-peer components are loosely coupled or even independent from each other. Components affect others indirectly by monitoring the system, recognizing changes caused by other components' actions, and by issuing own actions. This approach is generic and open for future extensions. It allows to design scalable, self-optimizing, and resilient grid systems that have no single point of failure.

A component is a distributed software system that uses a peer-to-peer algorithm to achieve its goal. The peer-to-peer paradigm was invented for large, voluntary PC networks. We apply this concept to the field of self-optimizing grid systems and show how this new architecture can be deployed in the domain of data management. Here, the various management aspects like replica creation, placement, access optimization, synchronization, etc. are cooperatively solved in separate peer-to-peer component layers. All components run concurrently. Each of them optimizes the system with respect to its goals. This separate optimization process performed by the interaction of small, manageable components gives similar or even better results, while being less complex than the holistic approach that tries to optimize the system in one step.

1 Introduction

The peer-to-peer (P2P) approach has become very popular in the last few years [13]. Originally created for the easy sharing of audio/visual data among private PC owners [2, 3, 14, 18], we believe that P2P techniques are now mature enough to allow for a better design of large scale grid systems. Current grid systems, like the Globus toolkit [9] for example, are capable of handling up to hundred servers and several thousands of clients. With a growing number of jobs and larger amounts of data, however, important Globus components like resource discovery, grid schedul-

ing or replica management [1, 15] do not scale. This problem is not only caused by the slow OGSA service calls as demonstrated in [6], but it is a principle flaw of the architecture which contains several bottlenecks.

As a remedy, we advocate to use P2P techniques for implementing scalable grid systems. P2P is based on democratic principles: no hierarchical top-down ruling, but self-organization and autonomic computing in a bottom-up manner. Each peer interacts with its counterparts as equals. A peer may take the role of a client, a server, or both. The global state of the system is generally not necessary to know, which makes P2P systems scalable. The P2P approach has a number of nice features which makes it ideally suited for large grid systems:

- *Simplicity*: Current P2P systems are highly specialized to serve simple tasks like file sharing [3, 14, 18]. Gnutella [2, 5], for example, has only five functions for retrieving distributed files. Such extreme specialization results in simple program code.
- *Redundancy*: The overall task is done by many collaborating peers. When peers fail, others take over. While this does not guarantee fault-tolerance *per se*, it provides basic system resilience.
- *Flatness*: There is no hierarchy and hence no bottlenecks—provided that enough peers are available to do the job. All peers execute the same algorithm.
- *Locality*: Peers have no global view. Their actions are based on their local information horizon. Note that locality is a prerequisite for scalability.

Redundancy plays an important role in large grid systems, because distributed nodes and services may be unavailable, e.g. due to maintenance. Disasters can only be circumvented when peers are able to dynamically reorganize without human intervention.

In this paper, we propose to use P2P techniques for implementing grid systems. Rather than following the monolithic or the toolset approach [4, 8], we advocate to implement grid systems by interacting components, each of them

implemented according to the P2P paradigms. Existing *functional* P2P systems, like those for routing and file access shall be complemented by *self management components* like (data) load balancing and system optimization, which would eventually lead to autonomic computing systems [12].

Our current prototype focuses on data management. Components for replica creation [16] (to provide file availability) and replica synchronization [17] exist. They will be complemented by autonomous monitoring components, which control the system and take the necessary actions to improve the system availability and performance. The JXTA [10] framework could be used as an implementation environment.

In the following Section, we describe the terminology used throughout the rest of this paper. Thereafter we present how complex tasks can be solved by combining loosely coupled P2P systems and we apply our concept to the field of data management in Section 4. Finally we conclude the paper with a brief summary.

2. P2P Systems as Components

In our architectural framework, complex functions are performed by cooperating components, each of them being responsible for one specific aspect or task. A component is by itself a complete P2P system in the traditional sense, running on its own overlay network.

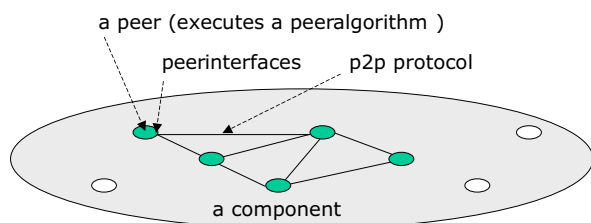


Figure 1. A component performs its tasks by executing identical instances of the peer algorithm on each peer. Peers use a P2P protocol for communicating with each other via peer interfaces. Component interfaces, not shown here, deliver the results to the outside world or to other components.

Figure 1 explains our terminology in more detail. It illustrates one *component*, or alternatively one *P2P system*. In the following, we use the terms component and P2P system synonymously. The participating *peers*, represented by grey nodes, are interconnected by an *overlay network*. Note that the links in the overlay network of a component may

be different from those of other components, and they are usually also different from the underlying physical network topology (refer to Fig. 2 below).

Components deliver their results to the outside world or to other components via their *component interface*. This interface, not shown in Fig. 1, is defined by an API. Note that components, whose interfaces comply to the OGSA standard [7], for example, can be integrated into existing grid environments.

Each peer of a component executes independently the same instance of the *peer algorithm*. Peers communicate with each other via a *P2P protocol* through *peer interfaces*. Peers have their own view on the system, called the *visibility horizon*. The visibility horizon is usually limited to the neighbors in the overlay network.

In summary, a peer consists of the following three parts:

- (i) A *peer interface* for internal communication between peers.
- (ii) A *peer algorithm* that represents the 'intelligence' of the system, ensuring that all peers work together in a coordinated way.
- (iii) A *component interface* that allows to interact with a component from the outside world.

We distinguish two kinds of components, *functional components* and *self-management components*. The former provide services to consumers (users and other components) and the latter keep the system in a stable state. The various components are typically independent from each other – especially those for self-management. Components can be arranged to higher functional units to perform more complex operations.

In the following we describe the two component types in more detail.

2.1. Functional Components

Functional components are invoked either by the user or by other functional components. Most contemporary P2P systems, like CAN, Chord, Gnutella, and Oceanio consist of only one functional component. In contrast to our approach they solve relatively simple tasks and they cannot be combined to perform more complex functions.

A typical example of a functional component is a file sharing environment that uses an information service for registering files and their corresponding locations. In this example the parts of each peer work as follows:

- (i) *Peer Interface*: Each peer provides interfaces to store, delete and query entries. It also maintains connections to neighboring peers.

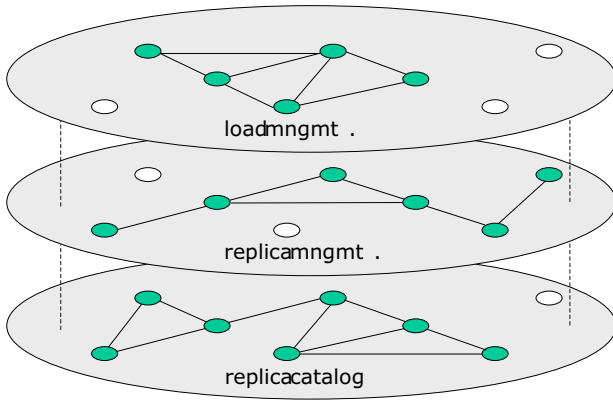


Figure 2. Three P2P components for distributed data management. Nodes (white and grey) represent servers. The underlying physical network is not shown. The components' peers (grey nodes) are linked to each other via their respective overlay network. The servers represented by white nodes do not run peers.

- (ii) *Peer Algorithm:* Gnutella uses the flooding for file queries, while CAN and Chord use routing tables and thereby can find the responsible peers in $O(\log(n))$.
- (iii) *Component Interface:* The component interface allows users or other components to use the system similarly to a client-server architecture or single application. The component interface invokes the peer algorithm according to the specific request.

2.2. Self-Management Components

The optimization and self-management components take care of the health and performance of other components. Each self-management component babysits just one other component rather than trying to optimize multiple components. As described in the next paragraph, sometimes different strategies are used to reach the same optimization goal. The strategies may complement each other, thereby leading to an improved overall result.

To guarantee a given file availability with unreliable file servers, for example, one component could apply an analytical model [16] while another might use a statistical approach with Markov models or Kalman [11] filters. The latter could detect patterns like 'on weekend systems are less well maintained' and create more replicas in advance. Having both components would lead to redundant and therefore more reliable systems.

Components with contradicting strategies may cause thrashing effects: A 'space saver' and an 'availability maximizer', for example, which do not know about the existence of their counterparts, might fight against each other. But properly designed, both components could coexist by voting on what to do best.

3. Complex Services Using P2P Components

Compared to the hierarchical architectures of current grid systems like Globus [9] for example, the P2P approach is more flexible, scalable, and even easier to design. We simply take the techniques of P2P systems and apply them to distributed servers in the grid. This way single points of failure are avoided and scalability to larger systems is guaranteed. In contrast to the volunteer P2P networks, we do not cope with sporadically available nodes, but assume more reliable systems. Even so the servers have only a limited view of the system and the global status is generally not known.

Components build on each other to provide complex services. For data-aware scheduling, for example, the job management and data management components may be combined as shown in Fig. 3.

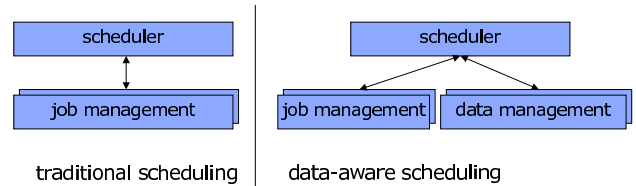


Figure 3. Data-aware scheduling combines job and data management components.

The job management component provides means for job submission, cancellation and execution. It also returns status information on the current length of the queue for monitoring purposes. The data management component has information on files and their replicas.

A data-aware scheduler receives jobs from the user, where information on the expected amount of data accesses and CPU time consumption is specified in the corresponding job description. To determine an optimal job location, the data aware scheduler accesses both, the job management and the data management. From the former it gets the load information and from the latter the current replica locations. With this information it decides where to execute the job.

In general, different components monitor and optimize a system independently. They are either loosely coupled or even independent from each other, affecting their peer's work mainly by their actions (and not through interfaces).

Many components are necessary for a complete grid system: job control, replica catalog, replica storage, synchronizer, workload balancer, data-aware scheduler, availability manager, access performance optimizer, user management, network and bandwidth management, accounting manager, security infrastructure, and many more.

4. Example: Grid Data Management with the P2P Approach

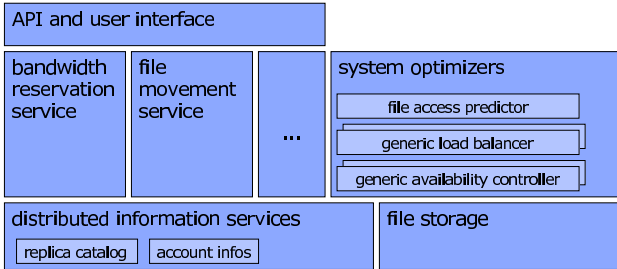


Figure 4. Components for a distributed data management system.

Fig. 4 shows the components of our data management system. At the lower level, the *distributed information service* allows persistent storage across the servers. It uses distributed hash tables similar to CAN or Chord [14, 18] for efficiently querying the system. Arbitrary meta information can be stored in this information service. The *replica catalog*, *account infos* and other meta data are also stored here.

```
interface ReplicaCatalog{
    // add new location of file
    void addMapping(string filename,
                   string location);

    // delete location of file
    void delMapping(string filename,
                   string location);

    // list locations of file
    list<string> listLocations(
        string filename);
};
```

Figure 5. Interface of a replica catalog component.

The replica catalog's component interface shown in Fig. 5 provides functions for inserting, deleting and listing of catalog entries. This is a transparent interface for the

user that hides the P2P system inside this component. No difference between a traditional grid component and a P2P component according to our approach is visible for the user. The *file storage* component at the right hand side of Fig. 4 manages the actual storage of the replicas in the system.

On top of these functional components higher level functional components like *bandwidth reservation* or a *file movement service* are setup similar to the approach already described in Sect. 3. Both use the distributed information service to retrieve information for planning their actions. Users either submit requests to the higher level components or retrieve information directly from the basic functional units.

Besides these functional units, some self-management components monitor the the system activities and try to improve its performance as discussed in the next Section.

4.1. Self-Management

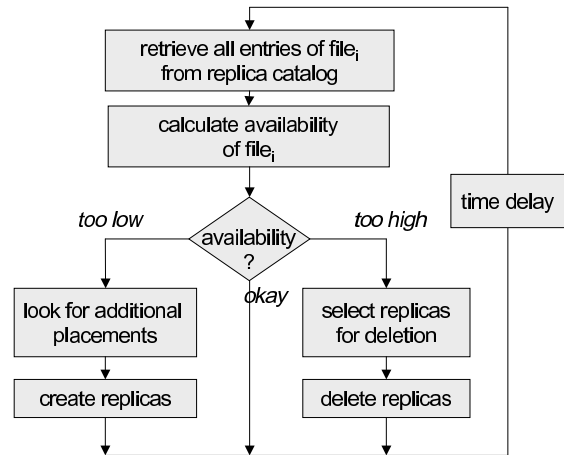


Figure 6. Outline of a typical self-management component. The flow diagram shows a peer algorithm for balancing data availability against space consumption.

In the field of data management several aspects fall into the category of self-maintenance, or self-management services. Replicas of files and redundant instances of services, for example, lead to an improved availability and more reliable systems, but need active management including creation and deletion of replicas or services. Load balancing with respect to disk capacity or CPU load is another issue that optimizes the system on the fly. When a distributed replica catalog grows to an unacceptable size in a certain node, a self-management component, that monitors the size of the catalogs, balances the size by moving entries to other servers. Analogously a monitor component observes the

number of requests to a server and balances the access load by moving catalog entries to other servers.

Self-management components work according to a common general approach—the feedback loop. Fig. 6 depicts the loop for controlling the file availability. Note that only a small set of the whole system is monitored in one instance and decisions are taken according to local status. Overlapping sets ensure an optimization across set boundaries.

In [16] we developed a component model for the availability management of replicas. Users specify a required file availability for files or groups of files. Running on each node the 'availability manager' calculates for each file on that host, according to the model, the number of necessary copies. Then it observes the neighborhood and counts the replicas of each file. If there does not exist enough replicas, it creates new copies. After some idle time, this process starts again. For performance reasons, not all nodes have their own local optimizer. In this case, the existing optimizers are responsible for sets of nodes.

```
interface Loadbalance{
  // get current load
  Load getLoad();

  // get maximal managable load
  Load getMaxAcceptableLoad();

  // push load to other host
  void pushLoadTo(string host, Load load);

  // pull load from other node
  void pullLoadFrom(string host, Load load);
};
```

Figure 7. Interface of a generic load balancer component.

Schemes for load balancing can be applied in several domains. With a generic interface as shown in Fig. 7, a 'load balancer' self-management component can be used to balance replicas on servers, replica entries in catalogs, or jobs on nodes. This concept can be applied to other aspects of self-management as well. The resulting re-use of existing code simplifies the system design and code maintenance. In the extreme case, components could even manage themselves (or other instances of themselves) if they provide the necessary interfaces.

5. Conclusion

Our approach of using P2P components is a viable method for implementing large grid environments that are truly scalable by employing interacting peers to perform the

necessary tasks. Moreover, the approach allows to combine simple components to perform complex services. Generic monitoring and optimization components can be used to control the health status of the system and take appropriate actions. Note that this principle can be recursively applied to autonomously control the health care components.

The concepts described above are used in our current prototype of a distributed data management system. The core components for replica management [16] and topology-aware synchronization [17] exist and they will be successively extended by additional components for job management, network management, software management etc.

References

- [1] A. Chervenak et al. Giggie: A framework for constructing scalable replica location services. *Proceeding of the IEEE Supercomputing*, November 2002.
- [2] K. Aberer, M. Puceva, M. Hauswirth, and R. Schmid. Improving data access in P2P systems. *IEEE Internet Computing*, 6(1):58–67, Jan./Feb. 2002.
- [3] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in P2P systems. *Communications of the ACM*, 46(2):43–48, 2003.
- [4] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. *Proceedings of CASCON'98*, 1998.
- [5] Clip2 Distributed Search Services. *The Gnutella Protocol Specification v0.4*.
- [6] D. Davis and M. Parashar. Latency performance of SOAP implementations. *Proceedings of the IEEE CCGrid*, pages 407–412, May 2002.
- [7] I. Foster, C. Kesselmann, J. M. Nick, and S. Tuecke. Grid services for distributed system integration. *IEEE Computer*, 35(6):37–46, June 2002.
- [8] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. *Proceedings of the 14th Conference on Scientific and Statistical Database Management*, 2002.
- [9] Globus Project. <http://www.globus.org>.
- [10] L. Gong. Project JXTA: A technical overview. Technical report, Sun Microsystems, 2001.
- [11] R. Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 1960.
- [12] J. Kephart and D. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, January 2003.
- [13] D. Milojjic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-peer computing. Technical report, HP Laboratories, 2002.
- [14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001.
- [15] A. Samar and H. Stockinger. Grid data management pilot (GDMP): A tool for wide area replication in high-energy physics. *Proceedings of IASTED International Conference on Applied Informatics (AI 2001)*, 2001.

- [16] F. Schintke and A. Reinefeld. On the cost of reliability in large data grids. Technical Report ZR-02-52, Zuse Institute Berlin (ZIB), Dec. 2002.
- [17] T. Schütt, F. Schintke, and A. Reinefeld. Efficient synchronization of replicated data in distributed systems. In *Proceedings of the International Conference on Computational Science*. Springer, 2003.
- [18] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of the 2001 Conf. on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.