Chapter 2

Scalable and Self-Optimizing Data Grids

ALEXANDER REINEFELD, FLORIAN SCHINTKE, THORSTEN SCHÜTT

Zuse Institute Berlin
Takustraße 7
14195 Berlin - Germany
{*reinefeld,schintke,schuett*}*@zib.de*

2.1 Motivation and Overview

Science and business relies to a large extent on the ability to quickly access remote computing resources without much administrative or technical overhead. It seems that the focus now shifts from a compute-centric to a data-centric view, that is, from 'Meta-Computing' to 'Data Grids'. Not only data mining applications require access to large amounts of geographically dispersed data, but also complex applications in classical science domains like particle physics, life sciences, or climate research, to name just a few. Scientific simulations are nowadays run in the Grid, that is, on networked computing and data resources.

Moreover, the greatly improved connectivity between geographically dispersed sites made global collaborations possible. Scientists are no longer in isolated local groups, but they collaborate in so-called *virtual organizations*—independent from time zones and locations. Grid computing [10], [24], [25], [44] came up to support such collaborations. The amount of data and computing resources that is handled by Grid software grows at an exponential pace, and we predict that the limits of current Grid systems will be reached in the very near future. Already today, doubts have been raised [16], [27], [48] that current Grid systems are indeed able, as claimed by their respective developers, to efficiently manage the required millions of files and jobs [11].

For this reason, we devised a scalable, self-optimizing architecture for next generation Grids. Our approach is based on the peer-to-peer paradigm in which all participating parties are equally important, resp. unimportant. Peers have only a limited knowledge on the global system status. All decisions and actions are taken locally, which improves scalability and fault-tolerance. While this is a very positive characteristic, the lack of global information

Draft Version of Annual Review of Scalable Computing. Vol. 6. Chapter 2.
Editor Yuen Chung Kwong. World Scientific. 2004.

unfortunately complicates the coordination and orchestration of the components, thereby making it difficult for the system to coordinately strive for common global goals.

Our architecture consists of a number of peer-to-peer (P2P) systems that are loosely coupled. In fact, it can be regarded as a superset of P2P systems. Each single P2P system appears as a component that provides just one specific function. Components affect each other by independently monitoring the system, recognizing changes, and by issuing appropriate actions. As we show later, this approach is generic and open for future extensions. It leads to scalable, self-optimizing and resilient Grid systems without single points of failure.

In the following section, we introduce the basic principles of our framework. Thereafter, we discuss a Data Grid for handling millions of distributed data objects. We focus on the following components:

- the *availability manager* that computes the optimal number of replicas required to guarantee a specified data availability (Sec. 2.4),
- the *synchronization manager* that is responsible for keeping the replicas in a consistent state (Sec. 2.5),
- and several *optimization components* that keep the system running without operator intervention (Sec. 2.6).

Finally, Section 2.7 describes the interactions between the components and Section 2.9 gives a brief summary.

2.2 P2P Systems as Components for Scalable Grids

Many common Grid environments like Globus [23], [26] or Unicore [7] are rather difficult to setup and maintain, because their functioning relies on the proper installation and configuration of many modules that must correctly play together. P2P systems, in contrast, are less complex. They contain a variety of small modules, each of them solving a single, isolated aspect.

Figure 2.1 illustrates a *component* (respectively a P2P system) consisting of *peers*, an *overlay network* and a *component interface*. All peers are the same. They act on the same hierarchy level (hence the name 'peers') and they are connected by a virtual overlay network. The P2P system is accessed via the component interface, that is, a service access point, which is available on every peer. Inside a P2P system the peers communicate via *peer interfaces*. The behavior of each peer is defined by the *peer algorithm* executed in each peer.

Major Grid systems like the latest version (3.0) of Globus [26] or Unicore use the recently established OGSI standard [53] for access and service invocation. P2P components can be made compatible by adapting their interfaces to the OGSI standard. This allows existing Grid systems to be gradually improved by exchanging existing services by newly developed scalable P2P services with compatible interfaces.

Complex or composed functions, as shown in Figure 2.2, are performed by multiple cooperating peer-to-peer systems, each of them being responsible for one specific aspect

Draft Version of Annual Review of Scalable Computing. Vol. 6. Chapter 2.
Editor Yuen Chung Kwong. World Scientific. 2004.

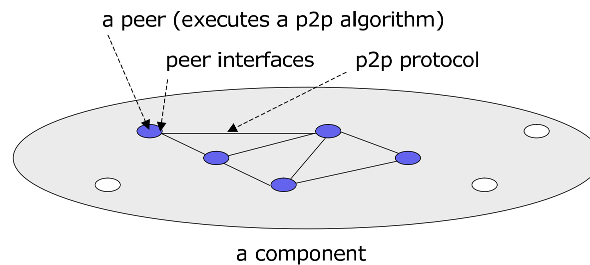


Figure 2.1. A peer-to-peer system performs its tasks by executing identical instances of the peer algorithm on each peer. Peers communicate with a P2P protocol via peer interfaces.

or task. Each of these P2P systems appears as an encapsulated component in our architecture [46].

In Figure 2.2 at the top layer, the availability component controls that there are always enough catalog- and replica-nodes in the system to guarantee a user-specified service availability. The middle layer provides a catalog service, that maps user-specified logical file names to physical file locators. This layer is typically implemented with distributed hashtables as in CAN [43], Chord [51], DKS(N,k,f) [5], or lh*rs [39]. The bottom layer provides synchronization services for keeping the catalogs and replicas in a consistent state. Note that the replicas (not shown here) would constitute a separate layer, similar to the catalog service component. All components use each others functions to provide a higher-level data access service to the user or application.

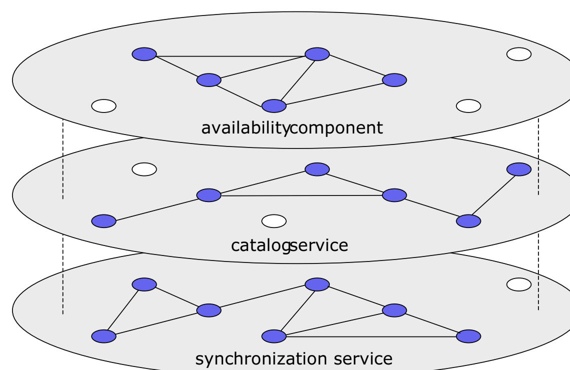


Figure 2.2. Three interacting P2P components. Nodes (light and dark) represent servers. The components' peers (black nodes) are linked via their respective overlay network (physical network not shown). Servers depicted by light nodes do not run this component.

2.3 A Scalable Data Grid with P2P Components

International scientific collaborations, like Cern's search for the Higgs Bosom [13] require concurrent access to millions distributed files from thousands of clients [11]. Such complex distributed environments can not be set up and organized as traditional systems where services are statically bound to specific hosts, because any such central site would inevitably become a performance bottleneck and a single point of failure. Rather, future Grids must operate in a self-optimizing way [32], [35], using replicated catalogs and services. For the proper functioning, mechanisms for the autonomous generation and placement of services at different sites are needed. From a coarse point of view, at least the following services must be provided:

1. *Data Replication* to improve data availability and to reduce wide area network transfers by distributing copies.
2. *Data Placement* to determine accurate locations for fast data access.
3. *Data Synchronization* to keep replicas in a consistent state.
4. *Data Caching and Staging* to benefit from spatial and temporal access locality and to speedup job execution by prefetching data.
5. *Data Movement* with efficient and secure protocols.

Figure 2.3 shows how these services can be orchestrated in a Data Grid. The optimization components, depicted at the top, trigger actions that are sent to a scheduler which takes care that the actions do not interfere with currently executing jobs or other optimization tasks. The scheduler checks permissions with the access component, makes reservations for the required resources, and starts jobs according to the schedule. Data transfers read data from mass storage and register new replica locations in the metadata component. A forecast unit retrieves logging data from the metadata catalog and detects usage patterns or network saturation. These forecasts are used by the optimization components to prepare the system for anticipated requirements in a timely fashion.

In the following, we describe a small subset of the components in more detail: the availability component, the synchronization component and some self-management components. For a more global view on the interaction of these components, the reader is referred to Fig. 2.13 in Section 2.7, where we put the mentioned components into the context.

2.4 Availability Component

The *Availability Component (AC)* computes the optimal number of replicas needed to meet a specified file availability in the Grid. In the following, we focus on the availability of data only. Note however, that the AC component can be applied to *any* replicated service—not just data[1].

[1] In the extreme case, the AC component could be recursively applied to itself, thereby ensuring that enough instances of itself are running to guarantee a continuous service.

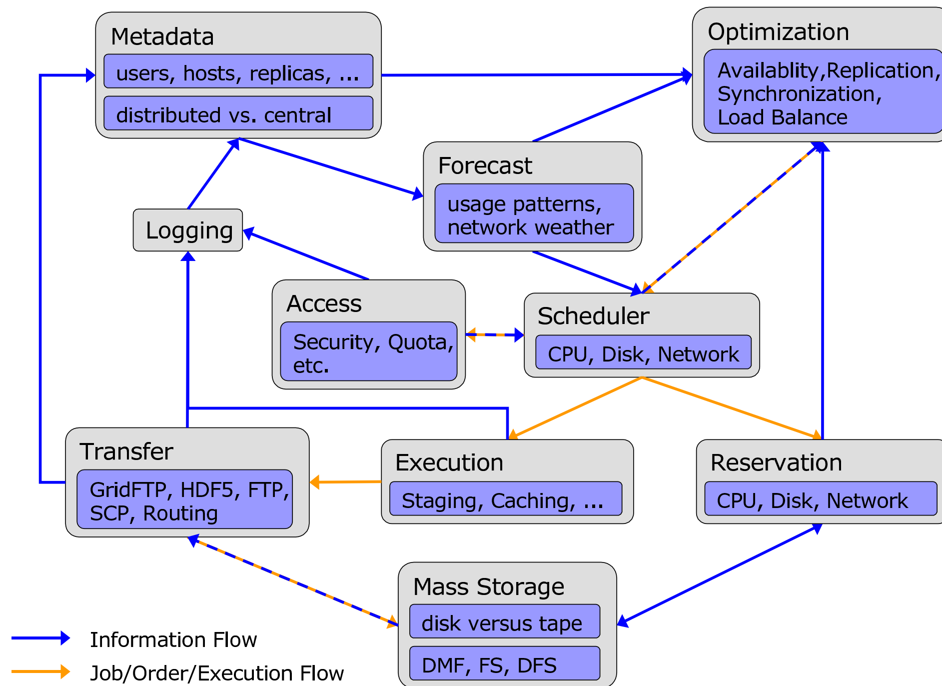


Figure 2.3. Components in a Data Grid and their interdependencies.

Let us assume that the nodes in our Grid can hold catalogs, files, or both. Catalogs and files may be replicated. It is the task of the AC to determine how many replicas are needed to guarantee a specified data availability in face of unreliable or unaccessible nodes.

In order to access a file, a *name resolution* must be performed to retrieve its physical location. More precisely, the given *logical file name (LFN)* must be mapped to a corresponding *physical file name (PFN)* that specifies the storage location. The mapping is done by querying one or more *replica catalogs*. Usually, there exist multiple replicas to a file, that is, a catalog lists more than one PFN for the same LFN. Additionally, we assume that the catalogs do not necessary contain the same information. For example, one catalog may have other PFNs to the same LFN than another one.

The file access is done in two stages: We first access the visible catalogs to get an aggregated list of PFNs. Then we try to access the PFNs, one after the other, until we succeed. In this procedure, three things may happen:

1. None of the catalogs can be accessed.
2. A catalog can be accessed but it has no PFN for the requested LFN.
3. A PFN has been obtained, but its corresponding node is not available.

We distinguish between a *local* and a *global* view. The *local view* corresponds to the perspective of a node inside the system. In large P2P systems, the local view may be very limited, disclosing only a small fraction of the participating nodes. The *global view* gives a bird's eye view on all components. This is a conceptual view in the sense that no single component may have a complete overview on a large P2P system. Typically, system designers or administrators are interested in the global view, because it allows to calculate the required resources and to configure, monitor and control the system according to the user requirements.

Several architectural choices for the catalog affect the scheme used in the availability component: single versus multiple catalogs, unreliable versus reliable catalogs, different catalog capacities, consistency models, and different probabilities of finding an entry in the catalog.

2.4.1 Basic Equations and Model Parameters

We first model the access to a single file. Accesses to multiple files can be modeled by applying our scheme to each single access and combining the results.

We make use of two basic equations, the $ok(p,n)$ function and the binomial equation $binom(n,k,p)$. The $ok(p,n)$ function (Eq. 2.4.1) describes the probability that at least 1 out of n redundant systems is available when each single systems is available with probability p on average.

$$\forall n \in \mathbb{N}, p \in \mathbb{R}, 0 \leq p \leq 1 :$$
$$ok(p,n) := 1 - (1-p)^n \tag{2.4.1}$$

Our second basic equation, $binom(n,k,p)$, determines the probability that k out of n independent items are selected, if each item is chosen with probability p on average (see Eq. 2.4.2). This equation can be used to compute the expected number of accessible replicas when using a given number of catalogs.

$$\forall n,k \in \mathbb{N}, p \in \mathbb{R},\ 0 \leq p \leq 1 :$$
$$binom(n,k,p) := \binom{n}{k} p^k (1-p)^{n-k} \tag{2.4.2}$$

We will use equations 2.4.1 and 2.4.2 as building blocks for deriving more complex equations. Our model takes the following architectural parameters into account:

p_{rep}: *The average probability for a replica node to be available.*

For simplicity, we do not distinguish different availabilities of different nodes. Hence, p_{rep} shall be the average availability or uptime probability over all replica nodes.

p_{cat}: *The average probability for a catalog node to be available.*

In client-server architectures, p_{cat} is usually greater than p_{rep} because catalogs are often kept on reliable servers with RAS features (reliability, availability, serviceability). In P2P systems, where catalogs are kept on the same nodes as replicas, p_{cat} equals p_{rep}.

Draft Version of Annual Review of Scalable Computing. Vol. 6. Chapter 2.
Editor Yuen Chung Kwong. World Scientific. 2004.

p_{entry}: *The average fraction of all existing PFNs (r_g) listed in a catalog.*

Replica catalogs may be out-of-date, e.g., because they have been offline for some time. p_{entry} tells how well a catalog is informed. It also characterizes the consistency model used to synchronize the catalogs: $p_{entry} = 1$ denotes a strong consistency model, whereas a lower p_{entry} represents a weaker catalog consistency. Obviously, p_{entry} is always greater than 0. If it were zero, no catalog would list the file and no replica could ever be accessed.

r_g: *The total number of PFNs in the system for a given LFN.*

r_g is independent of the accessibility of the replicas. We use r_g to determine the amount of replicas needed to guarantee a certain file availability. Moreover, with r_g we calculate the total number of PFNs stored in all catalogs as $r_g \cdot p_{entry} \cdot c$.

r_ℓ: *The number of locally visible PFNs.*

When multiple catalogs are locally visible, r_ℓ describes only the number of *unique* entries. For systems with just one reliable catalog $r_\ell = p_{entry} \cdot r_g$ on average.

c: *The number of catalogs responsible for an LFN.*

Some of the catalogs may be unavailable due to network partitioning or system downtimes. A low p_{cat} or p_{entry} can be (partly) compensated by adding more catalogs to the system.

2.4.2 Analytical Model

System with one fully available catalog.

We first analyze the simple case of a system with one reliable catalog. This situation occurs when the catalog runs on a server with fail-over facility and redundant network connections. Still, the catalog may be out-of-date because new replicas may not have been registered or because previous entries may have been overwritten due to lack of storage space.

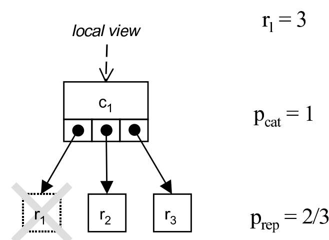


Figure 2.4. One fully available catalog in the local view

In the local view, illustrated in Fig. 2.4, the requester has reliable access to the catalog, $p_{cat} = 1$. He retrieves $r_\ell = 3$ PFNs. But when he actually tries to access the first PFN r_1,

he finds the corresponding server down. Only the other two PFNs are currently accessible, hence $p_{rep} = 2/3$.

In general, in the local view a requester with a reliable catalog accesses a file with $ok(p_{rep}, r_\ell)$ probability. The success rate can be improved by increasing either the availability of the replica servers p_{rep} or the number of PFN entries r_ℓ in the catalog.

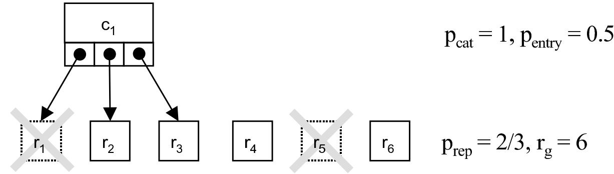


Figure 2.5. One fully available catalog in the global view.

In the global view, shown in Fig. 2.5, our example scenario has $r_g = 6$ PFNs. However, only $p_{rep} = 2/3$ of them can be accessed, because the servers holding r_1 and r_5 are temporarily down.

In general, only the PFNs listed in the catalog influence the overall availability, provided they are online. The probability that a specific catalog holds exactly i PFNs is:

$$\binom{r_g}{i} p_{entry}{}^i (1 - p_{entry})^{r_g - i}$$

Summing all cases, weighted with the availability $ok(p_{rep}, i)$, we get the file availability with one catalog in the global view:

$$\sum_{i=1}^{r_g} binom(r_g, i, p_{entry}) \cdot ok(p_{rep}, i)$$

System with c fully available catalogs

We now discuss the file access in a system with c reliable catalogs instead of just one. Here, the equation for the local view does not change, because the local requester follows the same principle: He first looks into all visible catalogs, determines the union of the results and then tries to access the replicas. In the snapshot illustrated in Fig. 2.6 two of the six entries in the catalogs refer to the same replica, hence $r_\ell = 5$.

In the global view, we may use the same equation as above, but must allow for c catalogs instead of just one. With c catalogs, each of them holding entries with probability p_{entry}, the overall probability for a replica to be listed in at least one catalog is $ok(p_{entry}, c)$. By replacing p_{entry} in the last scenario by $ok(p_{entry}, c)$ we derive the following file availability with c reliable catalogs in the global view:

$$\sum_{i=1}^{r_g} binom\big(r_g, i, ok(p_{entry}, c)\big) \cdot ok(p_{rep}, i)$$

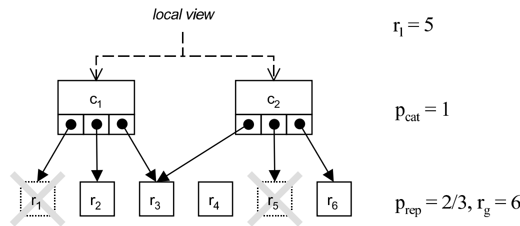


Figure 2.6. Two reliable catalogs in the local view.

In very large data Grids the namespace is often split up among several servers to keep the catalogs at a manageable size. This catalog splitting does not affect our model because by definition only the c catalogs responsible for the given filename are considered.

System with one unreliable catalog

When the catalog is installed on a regular rather than a fail-safe server, we have to take into account the probability p_{cat} that the catalog is not available. This is common for large distributed environments. In both, the local and the global view, we have to multiply the above equations by p_{cat}, because we model two events in a sequence: first a lookup to a catalog on an unreliable node, then a lookup to a PFN on an unreliable node. Hence, we get for the global view with 1 catalog of availability p_{cat}:

$$p_{cat} \cdot \sum_{i=1}^{r_g} binom(r_g, i, p_{entry}) \cdot ok(p_{rep}, i)$$

As can be seen, the overall availability of a file is restricted by the minimum of p_{cat} and the second factor. It is impossible to increase the overall availability above p_{cat} because the catalog is a single point of failure.

System with c unreliable catalogs

With c unreliable catalogs chances of getting access to a replica are higher than when using just one unreliable catalog. In the local view, we access c catalogs, determine the union over all retrieved PFNs and try to access them one after the other until we succeed. Rather than p_{cat} as before, chances are now $ok(p_{cat}, c)$ to get access to an active catalog server. Hence the equation for the local view is

$$ok(p_{cat}, c) \cdot ok(p_{rep}, r_\ell).$$

c	p_{cat}	local view	global view
1	1	$ok(p_{rep}, r_\ell)$	$\sum_{i=1}^{r_g} binom(r_g, i, p_{entry}) \cdot ok(p_{rep}, i)$
c	1	$ok(p_{rep}, r_\ell)$	$\sum_{i=1}^{r_g} binom\big(r_g, i, ok(p_{entry}, c)\big) \cdot ok(p_{rep}, i)$
1	p_{cat}	$p_{cat} \cdot ok(p_{rep}, r_\ell)$	$p_{cat} \cdot \sum_{i=1}^{r_g} binom(r_g, i, p_{entry}) \cdot ok(p_{rep}, i)$
c	p_{cat}	$ok(p_{cat}, c) \cdot ok(p_{rep}, r_\ell)$	$\sum_{j=1}^{c} \Big[binom(c, j, p_{cat}) \cdot \Big(\sum_{i=1}^{r_g} binom\big(r_g, i, ok(p_{entry}, j)\big) \cdot ok(p_{rep}, i)\Big)\Big]$

Table 2.1. File availability in the local and the global view.

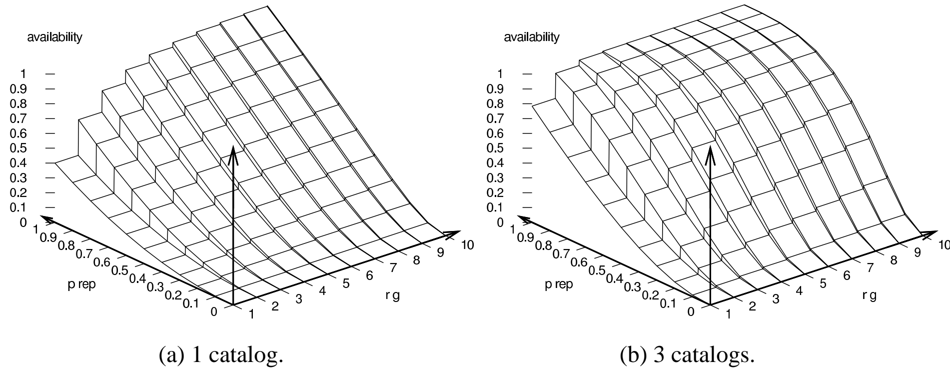
Note that the availability grows with each additional catalog that is accessed in the local view. Still, the upper limit is given by the maximum of $ok(p_{cat}, c)$ and $ok(p_{rep}, r_\ell)$ for the reasons described above.

Depending on the number of accessible catalogs and their degree of overlap, the listed replicas are more or less complete. To model this in the global view, we distinguish the different cases and sum all availabilities. The resulting file availability with c catalogs and availability p_{cat} in the global view is:

$$\sum_{j=1}^{c} \Big[binom(c, j, p_{cat}) \cdot \Big(\sum_{i=1}^{r_g} binom\big(r_g, i, ok(p_{entry}, j)\big) \cdot ok(p_{rep}, i)\Big)\Big]$$

Summary of Equations

Before we discuss the application of our model in a practical Grid environment, Table 2.1 gives a brief summary on the derived equations. All equations are shown for the local and the global view with different numbers of catalogs c and different catalog reliabilities p_{cat}.



(a) 1 catalog.

(b) 3 catalogs.

Figure 2.7. Availability in a P2P system ($p_{cat} = p_{rep}$) with $p_{entry} = 0.4$.

2.4.3 Applying the Model

The analytical model can be used to determine an optimal combination of the

- number of catalogs c,
- catalog size and coherence protocol in terms of entry probability p_{entry},
- catalog node availability p_{cat},
- replica node availability p_{rep},
- number of replicas in the system r_g,
- number of locally visible replicas r_l.

Finding an optimal parameter combination for a given situation is an optimization problem that can be solved with nonlinear integer programming [19] or heuristics [40]. In the following, we discuss some typical scenarios to get a qualitative understanding on the number of catalogs and replicas required for a certain overall availability.

Figure 2.7 shows a scenario where the catalog servers are as unreliable as the replica servers: $p_{cat} = p_{rep}$. We have chosen a probability $p_{entry} = 40\%$ of finding a PFN in the table. Obviously, the maximum system availability is strictly limited by the reliability of the catalog servers. This can be seen in the curve with $r_g = 10$ in Fig. 2.7.a, which is almost straight. Note that this curve is positioned slightly below an ideal straight line (not plotted), especially for small p_{rep}. This is because p_{cat}, having the same value as p_{rep}, increases the existing tendency.

With three unreliable catalogs in Fig. 2.7.b an almost arbitrary large availability can be reached, independent of p_{rep}. In other words, a low replica availability can be compensated by adding more replicas and more catalogs. Whether it is more effective to create an additional catalog or an additional replica, can also be evaluated with the model.

In practice, p_{rep} and p_{cat} are usually given by the hardware while r_g, c, and p_{entry} can be chosen within certain bounds. We therefore plotted the availability in a system with a catalog availability of 90 % and a replica availability of 50 %. Under these circumstances, adding further catalogs does not help much, as shown in Figure 2.8, because the bottleneck is not in the catalog availability but in the reliability of the replica servers and in the low entry rate of the catalogs. If we would have chosen a P2P system with $p_{rep} = p_{cat}$ instead, the curves in Figure 2.8 would have been symmetric.

Vendors often advertise their computers to have an availability of 'four nines' (99.99 %). Our model shows that such a high availability can be obtained even with unreliable components of only 90 % availability, when taking four to five catalogs and replicas. At a first sight, four to five catalogs resp. replicas may seem to be a high price, but note that the 99.99 % availability corresponds to a yearly downtime of less than an hour—produced with unreliable components each of them being down 36 days per annum!

2.5 Synchronization Component

After modeling the availability of replicated files, we now take care that all replicas are consistent, i.e., have the same contents. This is done with the *Synchronization Component (SC)* that builds on our *nsync* tool [47]. *nsync* can be manually invoked by the user or it can be integrated in the periodic optimization loop within a synchronization component. It supports a weak consistency model where replicas may be modified independently between synchronization points.

nsync computes nearly optimal synchronization plans with hierarchical gossip algorithms that take the network topology into account. Our primary design goals were maxi-

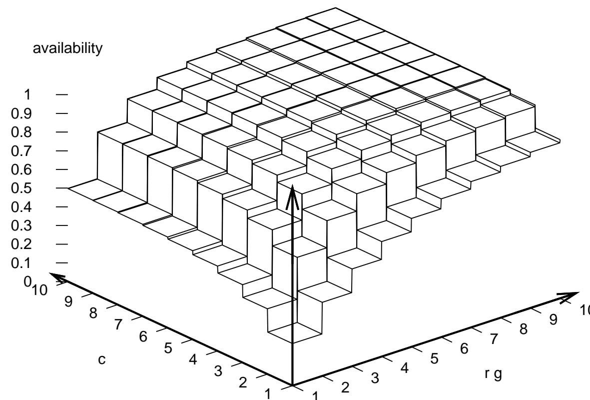


Figure 2.8. Availability with $p_{cat} = 0.9, p_{rep} = 0.5$ and $p_{entry} = 0.5$.

Draft Version of Annual Review of Scalable Computing. Vol. 6. Chapter 2.
Editor Yuen Chung Kwong. World Scientific. 2004.

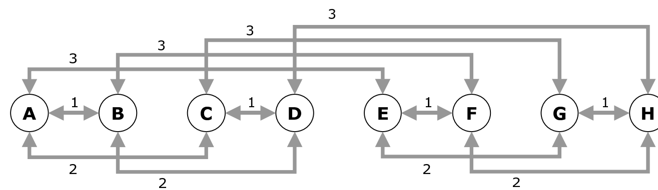


Figure 2.9. Three rounds of point to point synchronizations can synchronize eight peers.

mum performance and maximum scalability. They have been achieved by

- exploiting parallelism in the planning and the synchronization phase,
- omitting transfer of unnecessary metadata,
- synchronizing at a block level rather than file level,
- using sophisticated compression methods whenever appropriate.

By performing only point-to-point synchronizations between (peer) nodes, *nsync* provides the scalability and fault-tolerance features known from P2P systems. With its relaxed consistency semantic, *nsync* neither needs a master copy nor a quorum for updating distributed replicas. Conflicting modifications are reported and must be resolved by the user. Each replica is kept as an autonomous entity and can be handled with common tools and applications.

The *nsync* Algorithm

nsync synchronizes whole directories or subsets thereof. It first checks for files that have been modified since the last run and then propagates the updated data. In the worst case, i.e. when all nodes have updated files, an all-to-all communication is needed.

In a naive approach, each peer would send its updates to all partners, resulting in a total of $(n-1)n/2$ concurrent file exchanges. Each single exchange would trigger a separate local disk access, provoking many disk accesses and therefore resulting in a slow data transfer rate[2]. To avoid this, *nsync* uses only point-to-point synchronizations. Each peer participates in at most one point-to-point synchronsization at a time. Therefore, at most $n/2$ synchronizations are run concurrently. Note that in each synchronization a peer propagates not only the modifications made to its own data, but also the modifications it received from other peers in earlier rounds of the same synchronization. In total, each peer needs a maximum of $\log_2(n)$ point-to-point synchronizations in a complete graph, see the discussion below.

The synchronization process is given by a list of *rounds* of parallel point-to-point synchronizations. No barrier operation is executed between the rounds and therefore rounds

[2]Note that with the increased network bandwidth the disc access time often limits the synchronization time—rather than the link bandwidth.

may overlap. Figure 2.9 depicts the synchronization of eight peers A to H, split up in three rounds, each of them requiring four parallel point-to-point synchronizations.

Synchronization Phases in *nsync*

In more detail, the synchronization process is done in seven phases:

1. **Configuration:** The configuration is read and data structures are initialized. The configuration file (in XML notation) lists the repositories with their corresponding hosts. The network topology is treated as a hierarchical composition of basic topologies like switched networks, hubs, rings, chains, meshes, tori. Regular expressions are used to specify a subset of files for synchronization.

2. **Startup:** Instances of *nsync* are started on all participating peers via ssh. Each instance allocates an Internet network socket and informs the sync-manager about its port number. All further communication between the instances and the sync-manager is done via this socket and is not tunneled via ssh for performance reasons.

3. **Checking for Changes:** Each peer determines all changes since the last sync and sends an estimation on the expected data transfer size to the sync-manager.

4. **Planning:** The sync-manager calculates a synchronization plan and broadcasts it to all peers. In this calculation, it takes the topology and the communication bandwidths of the network links into account and computes an optimized communication sequence with a distributed gossip scheme (described below).

5. **Dry Run:** The peers simulate the synchronization plan without sending file contents to check for possible conflicts. False conflicts are sorted out by checking the conflicting replicas with MD5 checksums. If they were changed in the same way, that is, if they have the same MD5 checksum, *nsync* will not report them as conflicting. Only true conflicts are reported to the user.

6. **Synchronization:** The planned synchronization rounds are concurrently executed on all nodes. Each node communicates with at most one other node in a round to update the files on both sides.

   For synchronization, *nsync* uses different block transfer methods, depending on the available bandwidth between the nodes. For links with a low bandwidth *rsync* [52] and/or *bzip2* [12] are used to reduce the data volume. On faster links, when the compression process would take longer than the data transmission, rsync and bzip2 are switched off.

7. **Finalization:** The sync-manager waits for all peers to finish. All peers update their local metadata in a two phase commit protocol.

Topology-Aware Synchonization

Several gossip algorithms [8], [29], [33] have been proposed for synchronizing data with point-to-point communication. The *constant model* takes only the startup cost of a connection into account. In the *linear model* the communication cost c is proportional to the size l of the data volume: $c = \beta + l\tau$, where β is the startup cost and τ is the transfer time of a unit-length message. τ is assumed to be constant for all links. Since *nsync* is designed for synchronizing large amounts of data, we use the linear model.

Determining a cost-optimal gossip plan for arbitrary graphs is NP-hard [36]. We therefore simplify our algorithm by not supporting arbitrary graphs but only hierarchies of regular graph classes. Additionally, we treat network hubs like switches, thereby neglecting possible congestion in the hub. With these two restrictions, our plan generation algorithm has a runtime complexity of $O(n^2)$, but in practice often runs in $O(n \log_2 n)$. For some classes of graphs [29], [33], it generates optimal plans.

Switched networks are modeled by complete homogeneous graphs, which can be solved with the optimal algorithm in [33]. It needs $\lceil \log_2 n \rceil$ rounds (resp. $\lceil \log_2 n \rceil + 1$ rounds) in graphs with an even (resp. odd) number of peers. Depending on the amount of updated repositories, the synchronization time varies between $O(\log_2 n)$ for a one-to-all broadcast and $O(n)$ for an all-to-all broadcast. Similar algorithms for other graphs like rings and busses are also known [33], but they are only optimal in the constant model where the link bandwidth is ignored.

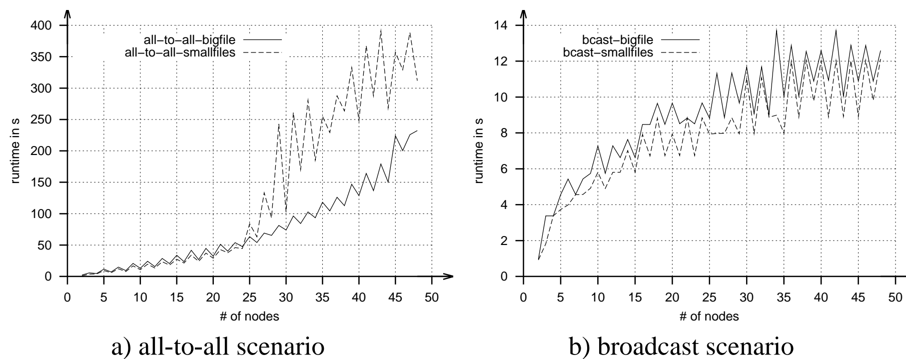
Applying the scheme for complete graphs to practical scenarios can lead to inefficient synchronization. Therefore we developed two heuristics that take the network characteristics into account. We treat topology graphs as a hierarchy of subgraphs. Our heuristic tries to use low-bandwidth links less frequently and it tries to start the local distribution before the communication across wide area networks so that both can overlap in time. These heuristics can be recursively applied for hierarchical topologies, which further improves the performance [47].

Results

We demonstrated the performance of *nsync* on a large Intel cluster with the following four scenarios:

- *all-to-all-bigfile:* One 10 MB file on each peer must be propagated to all others.
- *all-to-all-smallfiles:* Same as before, but the data is split into 1024 files.
- *bcast-bigfile:* One 10 MB file on *one* peer must be broadcasted to all others.
- *bcast-smallfiles:* Same as before, but the data is split into 1024 files.

Figure 2.10.a shows the results for the all-to-all communications and Figure 2.10.b for the broadcast scenarios. Each data point shows the best result from two independent runs. The benchmark was executed on a 96-node dual Intel Pentium-III cluster with 850 MHz, 512 MB memory, and 4 GB disk space, and a Fast Ethernet LAN switched by a CISCO 5509 with 3.6 Gbps internal bandwidth. Hence, the switch can be saturated with eighteen



a) all-to-all scenario b) broadcast scenario

Figure 2.10. Performance of *nsync* on clusters with up to 48 nodes.

100 Mb links in full duplex mode, and we therefore run our benchmarks only on up to 48 nodes (= 24 point-to-point communications).

The zigzag pattern in all graphs is caused by the gossip scheme, which needs more rounds in graphs with an odd number of peers. In the *all-to-all-bigfile* scenario, the runtime grows linearly with the number of peers. At 45 peers a total of 450 MB of user data is handled in each peer, which does not fit into the peers filesystem cache, and therefore a sharper rise can be noticed. In the *all-to-all-smallfiles* scenario, the Linux file handling and caching mechanisms come to a limit at 25 peers, where each peer has to handle 25,600 files.

In the two broadcast scenarios the curves rise logarithmic with the number of peers, because a broadcast tree is built up.

2.6 Self-Management Components

Having several file replicas and redundant service instances contributes to an improved overall availability. As seen in the previous sections they require active runtime management for creating and deleting replicas or service instances. Ideally, a system should be capable of maintaining and even optimizing its services without human intervention. In this section, we present methods for *self-maintenance* and *self-optimization* that build on the P2P paradigm.

Feedback Loop

All self-management components work according to a common pattern, the *feedback loop*. Fig. 2.11 illustrates a feedback loop for controlling file availability. It periodically checks the file accessibility in its local horizon and triggers actions when the availability is below or above a given threshold. Any single instance of such a self-management component is only capable of monitoring and controlling a small subset of the whole system. Hence it can only take local decisions. However, the combined effort of instances with overlapping horizons optimizes the global system across the set boundaries.

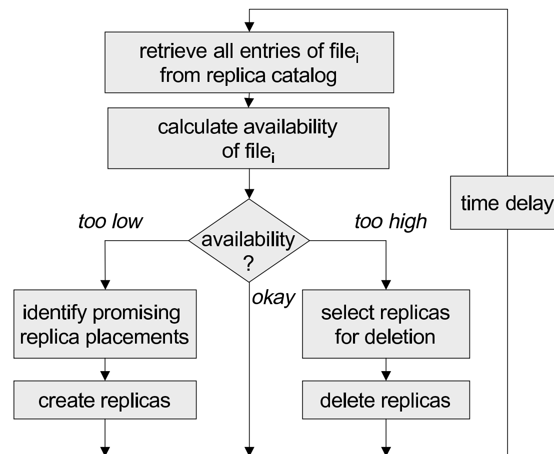


Figure 2.11. Outline of a typical self-management component with a feedback loop. The diagram shows a peer algorithm for balancing data availability against space consumption.

This does not only apply to file availability but also to the load balancing of CPUs, disk space, tape archives, networks, or other resources. Self-management components should be designed in a generic style so that they are capable of optimizing a wide variety of resources. The resulting code re-use simplifies the system design and code maintenance. In the extreme case, components can even manage themselves, that is, other instances of themselves, if they are equipped with the necessary interfaces.

Generic Load-Balancing Component

The concept of load balancing is so fundamental and generic that it can be applied to many different domains. Figure 2.12 illustrates some different kinds of 'loads'. In the most

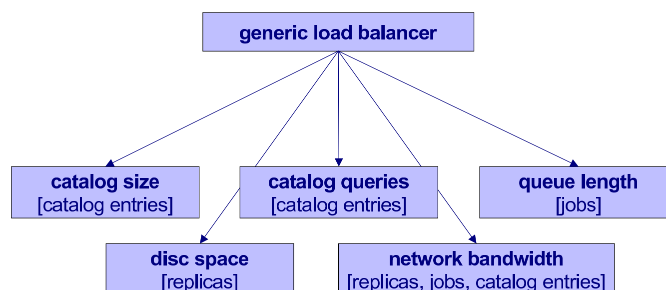


Figure 2.12. A generic load balancer balances different components.

general implementation, the components report their load to the load balancer and receive instructions on how much load to shift. This is done via generic interfaces:

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

  // get maximal managable load
  Load getMaxManagableLoad();

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

  // pull load from other peer
  void pullLoadFrom(Host host, Load load);
};
```

Each component that needs load-balancing service must support this interface. With this scheme, only one generic load-balancer needs to be implemented. Multiple domain-specific strategies allow to cope with dynamic changes in the environment.

Avoiding Thrashing Effects

Because all control and actions are done locally, thrashing effects might occur. As an example, a *space saver* and an *availability maximizer* might fight against each other which leads to thrashing effects.

To avoid thrashing effects, components must allow small deviations from the optimal state before they initiate actions. If thresholds can be dynamically adjusted, a monitoring component that detects thrashing patterns may be able to raise the threshold limits of the thrashing partners until the system stabilizes. This way, a hierarchy of controlling components is introduced which control each other for the benefit of the overall system services.

Placement of Self-Management Components

Due to the lack of global status information, the placement of self-management components is not obvious in P2P systems. In the following, we discuss the placement of management components with our file availability system in mind.

The first and simplest approach would be to install optimization service instances on each peer. They would determine the required amount of replicas for each file of local interest and would check how many of them are available in the neighborhood. If there are not enough copies, additional ones must be created. Clearly, this simple approach is not suitable for practical application, because it causes unnecessary overhead.

In the second approach, optimizer instances run on some peers only. The optimizers monitor the file availability within their horizon. In this hierarchical approach it is difficult to ensure that all peers get monitored by at least one optimizer instance. Moreover, single points of failure must be avoided by introducing areas of controlled overlap between the optimizer instances.

In the third approach, that our system follows, each peer that needs optimization services is responsible for registering for their optimizer instance(s). If there is only an insufficient number of optimizers available or if the existing instances are overloaded, additional ones are created via a factory mechanism. With this bottom-up approach, each peer gets monitored and optimized without incurring too much overhead.

2.7 Putting it All Together

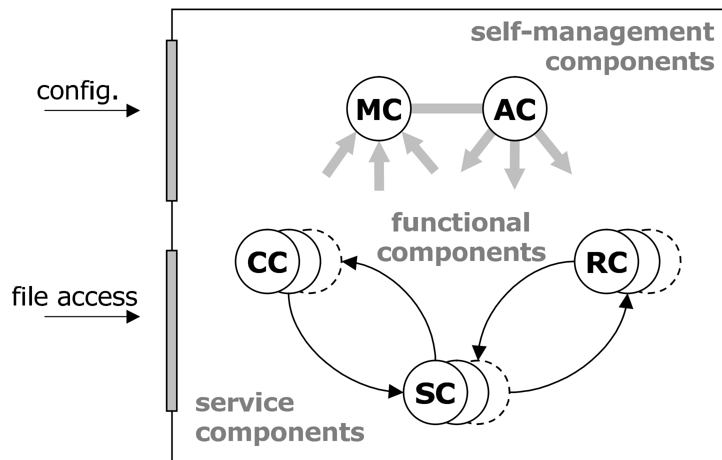


Figure 2.13. Key components of a distributed data management system as described in this paper. The availability component (AC) computes how many instances of the catalog component (CC) and replica component (RC) are needed to guarantee a certain overall file availability. To do so, AC gets its input from the monitoring component (MC). The synchronization component (SC) synchronizes the replicas and catalogs.

The described components constitute the core of a self-organizing Data Grid based on P2P techniques. Figure 2.13 illustrates their interplay. As can be seen, there exist several instances of the catalog (CC) and replica (RC) components. The synchronizer component (SC) ensures that all replicas and catalogs are in a consistent state. The self-management components are depicted in the upper part of the figure. In our simple example, only two components work tightly together: the monitoring component (MC) with the availability component (AC). The former checks periodically the number of active instances of the various components, and the latter triggers the necessary actions. Note that there are only two interfaces to the outside world, one for the administrator to define the targeted availability and one for the user (or application) to access a file. All other components do not need to be visible.

Clearly, this is just a simple sketch of the most basic services required in a Data Grid. A complete system comprises many more components. However, their functioning and their

interplay is very similar to the framework described here. In our design we focused on versatility. The monitoring (MC) component, for example, is not only used to monitor CC and RC as shown in Fig. 2.13, but also monitors SC and AC—and even its own instances. Likewise, AC computes the required availability of all components, including itself.

2.8 Related Work

Grid Scalability. Historically, Grids were built on centralized components, and switched only recently to hierarchically distributed architectures. Following this trend further on will lead to P2P-based Grid services and will thereby result in improved scalability and reliability.

With the increasing deployment of Grid systems in practice it became clear that the central Metadata Directory Service (MDS) [21] of Globus [23] is a performance bottleneck. Being based on LDAP, the write performance of MDS is poor [48]. Consequently, a decentralized approach using index servers and registration protocols were implemented in the follow-up version MDS-2, which was then called Globus Monitoring and Discovery Service [15], but still the performance is not as good as expected [56].

The convergence of Grid and P2P systems is also predicted by Foster *et al.* [22]. With the OGSA architecture in mind, they claim that developers of more powerful P2P systems "are going to become increasingly interested in standard infrastructure and tools for service description, discovery, and access, as well in standardized service definitions and implementations".

To further improve the scalability and reliability of services, Grids should utilize P2P techniques [41]. The distributed hash tables of CAN [43], Chord [51] and others [1], [9] show how efficient, scalable catalogs can be built on top of unreliable peer-to-peer nodes. Fox *et al.* [28] propose to organize Grid services on top of multiple 'peer groups' that perform the work of different tasks. This is similar to our approach [46], but they do not show how to perform self-management and optimization of the middleware itself.

Availability. Most availability models proposed in the literature focus on replication and consistency in distributed database systems. A primary topic is the network traffic overhead caused by the consistency schemes [34], [37], [38], [55]. Quorum consensus [31] in the case of network separations and other system malfunctioning has also been extensively studied. These aspects are important for database installations, where transaction processing is a major concern. Our focus, however, is on very large, possibly international Grid environments with a high degree of site autonomy and therefore with a weaker data consistency. Hence, our model is intended for scientific domains where the reading of existing data and the writing of new data happens several orders of magnitude more often than modifications on existing data. One such scenario is the European DataGrid project [20], where schemes for the distributed management of several petabytes [49] are developed. In DataGrid, some hundred million files located in hundreds of geographically distributed sites must be reliably maintained for a large physics community.

The scheme for creating new replicas in a feedback loop presented in [42] is less expressive than our model, because it only captures the local view of a system with a single

central replica catalog. The adaptive data replication algorithm described in [54] focuses on caching and supporting consistency for the replicas. Another dynamic replication scheme presented in [2] uses finite automata to predict future access patterns which are used to improve the data location with respect to network transfer cost. Economy-based approaches are proposed in [14] and [50].

Compared to our model [45], none of these approaches distinguishes the global from a local view, and none of them reflects the multi-staged access via distributed catalogs (with partly redundant entries) and replica servers.

Synchronization. The replica management systems discussed in the literature are mainly targeted at distributed file systems like AFS, where strict data consistency is of prime importance. In such settings a much tighter connection between the individual nodes is assumed than can be supported in geographically distributed Grid environments. Here, many different administrative domains with different local policies are combined into a single environment. Moreover, in Grids the files are kept autonomous and are less often updated from remote nodes, hence the weaker consistency model. More flexible consistency models [34], [55] do not match our requirements, because we aim at offline synchronization that allows the users to trigger transactions.

For more loosely coupled Grid environments the *Rumor* [30] system is most notably. It supports synchronization in environments with occasionally disconnected nodes. For this purpose, Rumor uses versioning vectors to update offline nodes at a later time.

The broadcasting of the differences between files can be done with *rsync+* [18] instead of *rsync* to save computational overhead. It is based on a master-copy model. If one out of *n* copies has been changed, the differences are locally calculated (using only one other client and thereby reducing transfer traffic) and only the patch is transmitted to all other peers.

Autonomic Computing. The upcoming field of autonomic computing [32] tries to make systems self-managing and self-adapting to dynamic environments. Applied to the domain of Grid computing, where heterogeneous systems at geographically distributed sites, differing local system policies, and installation status must work smoothly together, this can help to reduce maintenance, installation, and test efforts.

Projects like AutoMate or OptimalGrid propose autonomic Grid applications [4], [17], that use the Grid environment more efficient and discuss the required infrastructure extensions. To support such applications, AutoMate uses an autonomic composition engine to calculate a composition plan based on dynamically defined objectives and constraints that describe how a given high-level task can be achieved by using available basic Grid services [3]. Their dynamic composition model is based on relational algebra and graph theory. Our approach, in contrast, does not attempt to make applications autonomic and self-optimizing, but rather the Grid middleware.

2.9 Conclusion

We described the concepts that are fundamental to our *ZIB-DMS* prototype of a Data Grid. The core components for metadata management, replica management, and topology-aware

synchronization are already in place and their stability has been proved in the GridLab project [6]. Further components are currently being added to realize the architecture shown in Fig. 2.3.

Acknowledgements

We are grateful to IBM for awarding a Faculty Award to the first author, which made possible to tackle this fundamental research project. Additional funding came from the European Commission through the EU projects GridLab and DataGrid.

2.10 Bibliography

[1] K. Aberer, M. Punceva, M. Hauswirth, and R. Schmid. Improving data access in P2P systems. *IEEE Internet Computing*, 6(1):58–67, Jan./Feb. 2002.

[2] S. Acharya and S.B. Zdonik. An efficient scheme for dynamic data replication. Technical Report CS-93-43, Department of Computer Science, Brown University, September 1993.

[3] M. Agarwal and M. Parashar. Enabling autonomic compositions in Grid environments. In *Proceedings of the 4th Intl. Workshop on Grid Computing*, pages 34–41. IEEE Computer Society, 2003.

[4] M. Agarwal et al. AutoMate: Enabling autonomic applications on the Grid. In *5th Annual Intl. Workshop on Active Middleware Services (AMS'03) 2003 Autonomic Computing Workshop*, pages 48–57. IEEE Computer Society, June 2003.

[5] L.O. Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS (N, k, f): A family of low communication, scalable and fault-tolerant infrastructures for P2P applications. In *Proceedings of the IEEE CCGRID2003*, pages 344–350. IEEE Computer Society, 2003.

[6] G. Allen et al. Enabling applications on the Grid - a GridLab overview. *Intl. Journal on High Performance Computing Applications*, 17(4):449–466, 2003.

[7] J. Almond and D. Snelling. Unicore: Secure and uniform access to distributed resources via the world wide web. White paper, October 1998.

[8] B. Baker and R. Shostak. Gossips and telephones. *Discrete Mathematics*, 2:191–193, 1972.

[9] 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.

[10] F. Berman, G. Fox, and T. Hey, editors. *Grid Computing*. Wiley, May 2003.

[11] S. Bethke, M. Calvetti, H.F. Hoffmann, D. Jacobs, M. Kasemann, and D. Linglin. Report of the steering group of the LHC computing review. Technical report, CERN European Organization for Nuclear Research, February 2001.

[12] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Systems Research Center, 1994.

[13] D. Butler. The Grid: Tomorrow's computing today. *Nature*, 422:799–800, April 2003.

[14] M. Carman, F. Zini, L. Serafini, and K. Stockinger. Towards an economy-based optimisation of file access and replication on a data Grid. In *Proceedings of the IEEE CCGrid2002*, pages 340–345, Berlin, Germany, May 2002. IEEE Computer Society.

[15] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *10th IEEE Symp. On High Performance Distributed Computing*, 2001.

[16] D. Davis and M. Parashar. Latency performance of SOAP implementations. In *Proceedings of the IEEE CCGrid2002*, pages 407–412, May 2002.

[17] G. Deen, T. Lehman, and J. Kaufman. The Almaden OptimalGrid project. In *5th Annual Intl. Workshop on Active Middleware Services (AMS'03) 2003 Autonomic Computing Workshop*, pages 14–21. IEEE Computer Society, June 2003.

[18] B. Dempsey and D. Weiss. On the performance and scalability of a data mirroring approach for I2-DSI. In *Network Storage Symposium*, 1999.

[19] L.W. Dowdy and D.V. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287–313, 1982.

[20] M. Draoli, G. Mascari, and R. Puccinelli. DataGrid - project presentation, 2001.

[21] S. Fitzgerald, I. Foster, C. Kesselman, G. v. Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proceedings of the 6th IEEE Symposium on High-Performance Distributed Computing*, pages 365–375, 5-8 August 1997.

[22] I. Foster and A. Iamnitchi. On death, taxes, and the convergence of peer-to-peer and Grid computing. In *2nd Intl. Workshop on Peer-to-Peer Systems (IPTPS'03)*, pages 118–128. Springer Verlag Heidelberg, October 2003.

[23] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl. Journal on Supercomputing Applications*, 11(2):115–128, 1997.

[24] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Intl. Journal on Supercomputer Applications*, 15(3):200–222, 2001.

[25] I. Foster and C. Kesselmann, editors. *GRID Blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers, 2nd edition, December 2003.

[26] I. Foster, C. Kesselmann, J.M. Nick, and S. Tuecke. Grid services for distributed system integration. *IEEE Computer*, 35(6):37–46, June 2002.

[27] I. Foster et al. Giggle: A framework for constructing scalable replica location services. *Proceedings of the Supercomputing Conference (SC2002)*, November 2002.

[28] G. Fox et al. Peer-to-peer Grids. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing*, chapter 18, pages 471–490. Wiley, May 2003.

[29] P. Fraigniaud and E. Lazard. Methods and problems of communication in usual networks. *Discrete Applied Mathematics*, 53:79–133, 1994.

[30] R. G. Guy, P. L. Reiher, D. Ratner, M. Gunter, W. Ma, and G. J. Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. In *Advances in Database Technologies, ER '98 Workshop*, volume 1552 of *LNCS*, pages 254–265. Springer, 1998.

[31] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, 1986.

[32] P. Horn. Autonomic Computing: IBM's perspective on the state of information technology. IBM Corp., October 2001.

[33] J. Hromkovic, C. Klasing, B. Monien, and R. Peine. Dissemination of information in interconnection networks. *Combinatorial Network Theory*, pages 125–212, 1995.

[34] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. How to select a replication protocol according to scalability, availability, and communication overhead. In *IEEE Intl. Conference on Reliable Distributed Systems (SRDS'01)*, pages 24–35, New Orleans, October 2001. IEEE CS Press.

[35] J. Kephart and D. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, January 2003.

[36] D.W. Krumme, G. Cybenko, and K.N. Venkataraman. Gossiping in minimal time. *SIAM Journal on Computing*, 21(1):111–139, 1992.

[37] A. Kumar and A. Segev. Cost and availability tradeoffs in replicated data concurrency control. *ACM Transactions on Database Systems*, 18(1):102–131, 1993.

[38] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, 1992.

[39] W. Litwin and T. Schwarz. lh*rs: a high-availability scalable distributed data structure using Reed Solomon codes. In *Proceedings of the 2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 237–248. ACM Press, 2000.

[40] S. Mahmoud and J. S. Riordon. Optimal allocation of resources in distributed information networks. *ACM Transactions on Database Systems*, 1(1):66–78, 1976.

[41] D. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-peer computing. Technical report, HP Laboratories, 2002.

[42] K. Ranganathan, A. Iamnitchi, and I. Foster. Improving data availability through dynamic model-driven replication in large peer-to-peer communities. In *Proceedings of the IEEE CCGrid2002*, pages 376–381, Berlin, Germany, May 2002. IEEE Computer Society.

[43] 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.

[44] A. Reinefeld and F. Schintke. Concepts and technologies for a worldwide Grid infrastructure. In *EuroPar 2002 Parallel Processing*, volume 2400 of *LNCS*, pages 62–71. Springer, 2002.

[45] F. Schintke and A. Reinefeld. Modeling replica availability in large data Grids. *Journal of Grid Computing*, 1(2), 2003.

[46] F. Schintke, T. Schütt, and A. Reinefeld. A framework for self-optimizing Grids using P2P components. In *Proceedings of the 1st Intl. Workshop on Autonomic Computing Systems*, pages 689–693. IEEE Computer Society, September 2003.

[47] T. Schütt, F. Schintke, and A. Reinefeld. Efficient synchronization of replicated data in distributed systems. In *Proceedings of the Intl. Conference on Computational Science*, volume 2657, pages 274–283. Springer, 2003.

[48] W. Smith, A. Waheed, D. Meyers, and J. Yan. An evaluation of alternative designs for a Grid information service. *Cluster Computing*, 4(1):29–37, 2001.

[49] H. Stockinger, A. Samar, B. Allcock, I. Foster, K. Holtman, and B. Tierney. File and object replication in data Grids. *Journal of Cluster Computing*, 5(3):305–314, July 2002.

[50] H. Stockinger, K. Stockinger, E. Schikuta, and I. Willers. Towards a cost model for distributed and replicated data stores. In *9th Euromicro Workshop on Parallel and Distributed Processing PDP 2001*, Mantova, Italy, February 2001. IEEE Computer Society.

[51] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 149–160. ACM Press, 2001.

[52] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, 1999.

[53] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt. Open Grid Service Infrastructure (OGSI) version 1.0. GGF Proposed Recommendation, July 2003.

[54] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, June 1997.

[55] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 29–42. ACM Press, 2001.

[56] Xuehai Zhang, Jeffrey L. Freschl, and Jennifer M. Schopf. A performance study of monitoring and information services for distributed systems. In *Proceedings of the 12th IEEE Intl. Symposium on High Performance Distributed Computing (HPDC'03)*, pages 270–281. IEEE Computer Society, 2003.

Draft Version of Annual Review of Scalable Computing. Vol. 6. Chapter 2.
Editor Yuen Chung Kwong. World Scientific. 2004.