Fault Tolerance and Scalability Issues in Distributed File Systems

Bibliographic Report

André Lage Freitas
andre.lagefreitas@etudiant.univ-rennes1.fr

Supervisors: **Gabriel Antoniu, Luc Bougé**
{Gabriel.Antoniu,Luc.Bouge}@irisa.fr

IFSIC, IRISA, Paris Team
Janvier 2008

Keywords: distributed file systems, fault tolerance, scalability, NFS, GFarm, grid data sharing service.

Abstract

This report studies *distributed file systems* exposing their issues. It focuses on fault tolerance and scalability issues to better understand how they deal with failures and scalability. It is exposed that *distributed file systems* scalability mechanisms still do not perform satisfactorily in large-scale environments. In this respect, it is possible to combine other approaches that address scalability to supply this deficit.

Bibliographic report, stage M2RI IFSIC, University of Rennes 1.

Contents

1 Introduction

In computer systems data need to be stored and accessed. In non distributed systems, local *file systems* commonly take care of this task by exploiting storage devices (hard disks, optical medias, etc.). On the other hand, distributed systems can take advantage of their environment to provide an infrastructure to handle data in a distributed fashion. In this context, *data access models* are very useful, as they simplify the system design [7].

We can outline two *data access models* commonly used in distributed systems. In the *message passing* model, processes exchange data by sending messages. In contrast, the *shared memory* model provides a unique address space for data sharing among processes. In this model, it is not the programmer's charge to manage the shared data, he accesses them by using libraries interfaces. Moreover, the *shared memory* model is more appropriate in scenarios where applications use a significant amount of data.

Distributed Shared Memory systems implement the *shared memory* data access model in a distributed environment. As a result, different machines in a network can transparently share the mentioned unique address space. *Distributed Shared Memory* systems proved to be efficient when dealing mutable data in terms of data consistency. Nevertheless, classical consistency protocols employed in these systems have limitations in a sense of scalability and fault tolerance. Their algorithms are not scalable besides they rely on a scenario in which faults are rare [4, 6].

Also targeting transparency in shared memory context, *distributed file systems* implement *file system* services in a distributed way. They use the file abstraction to represent the shared data and also to provide transparency. In contrast to *Distributed Shared Memory* systems, *distributed file systems* provide fault tolerance mechanisms. Furthermore, these systems address scalability issues, however current *distributed file systems* implementations still do not address scalability.

In contrast, *peer-to-peer* systems are specifically designed to provide data sharing in a large-scale scenario. This approach reaches its goal since it manages immutable data. When handling immutable data, it is not necessary to implement, consistency protocols, because the data are usually read-only. A few *peer-to-peer* approaches are proposed for mutable data management. However, they are not as scalable as the ones which handle immutable data, or they only support a configuration with a small number of data modifications or with few writers [6].

Based on the technologies explained above, hybrid approaches have emerged. Actually, the real interest is to take advantage of the qualities of each system to conceive other more efficient likewise hybrid ones. One example of this idea is the concept of *grid data sharing service* [5] that aims to provide transparent data management in large-scale. For this, *grid data sharing service* takes advantage of *peer-to-peer* system's scalability, it also uses consistency protocols from *Distributed Shared Memory* and fault tolerance mechanisms from classical results in the area of fault tolerance distributed systems.

Finally, *distributed file systems* leverages the persistence provided by storage devices to offer data management. Even though, they miss data handling in large-scale environment (e.g., NFS [9], GFarm [10]). On the other hand, *grid data sharing service* (e.g., GDS [5]) supplies the missing scalability in *distributed file systems*. This report studies fault tolerance and scalability in *distributed file systems* to better understand how *distributed file systems* and *grid data sharing services* can be combined together. As a result, it is possible to design a distributed data shared system which addresses consistency, fault tolerance, scalability and persistence

storage.

This report is structured as follows. Section 2 exposes *distributed file system* main issues while Sections 3 and 4 are dedicated to fault tolerance and scalability, respectively. In Section 5 are discussed the case studies exposing their fault tolerance and scalability mechanisms. Section 6 concludes with a discussion.

2 Distributed File Systems

Inherent to the operating system, the *file system* leverages the file abstraction to allow data access and storage in a transparently fashion. Operating system calls (syscalls) implement *file operations* (*create, delete, read, write*) and let users indirectly interact with storage devices. *Distributed file systems* implement *file system* services in a distributed environment allowing users to use them as in a local *file system* [8]. Thereby, *distributed file systems* can be implemented in kernel space or not.

This Section is dedicated to *distributed file system* main issues however fault tolerance and scalability are not discussed here due their great influence on the system performance and its working. In presence of failures, the *distributed file system* may delay requests responses (or even be stopped). This scenario can be achieved also if clients demand are huge or if the number of clients increases as well. Targeting the performance improvement and the system working, mechanisms for fault tolerance and scalability issues should be provided. Sections 3 and 4 show how such mechanisms can be provided and discuss them.

2.1 Transparency

A local *file system* is the interface between users and storage devices. When a user executes a *file operation*, it is translated to a lower level operation which can be understood by the storage device. *Distributed file systems* have to not only provide this transparency, but also to hide from users other details related to the distributed environment they are confined. The types of transparency inherent to a *distributed file system* are explained as follows.

Network transparency. *Distributed file systems* should store the data and allow their access by providing a common interface independently of the data location, through common *file operations*.

Access transparency. To provide users mobility, *access transparency* should be assured. It implies that users can choose the machine which they will log in and they will be able to access the same data despite the machine they chose.

Other types of transparency refer to the file name should be assured. *Location transparency* requires that the file name should not contain any information about the place where it is stored. *Location independence* assures that storage devices are not able to require the change of file name, i.e., the file name is not device storage dependent.

2.2 Semantics of sharing

When more than one user access a file, it is crucial for the *distributed file system* to specify how it will behave. These semantics define how *read* and *write* operations (file accesses) are

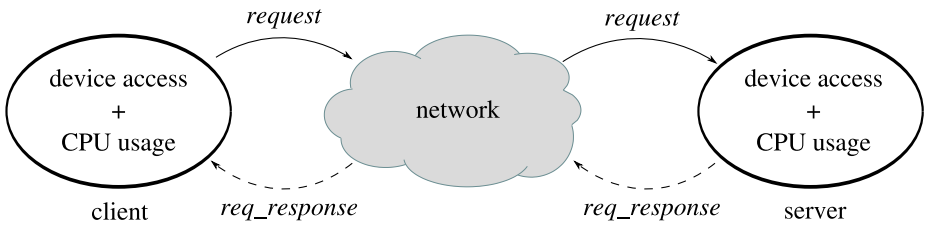

Figure 1: *Distributed file system* requests

related to each other during a *file session* - series of file accesses between opening and closing of a file. We can use different *semantics of sharing* to deal with shared accesses depending on the goal. In [8], four semantics of sharing are cited: *UNIX semantics*, *session semantics*, *immutable shared file semantics* and *transaction-like semantics*. Since *UNIX semantics* and *session semantics* are the most employed, we will focus on them.

UNIX semantics. In the *UNIX semantics*, each file has only a single image shared by all users. *Write* operations on it will instantaneously affect clients that have the same file opened. Moreover, *read* operations always correspond to the last previous *write* operation.

Session semantics. Different from *UNIX semantics*, *session semantics* does not provide a single file image. When performing a remote access on the latter, a local copy of the file is done on each client who opens it. File accesses are done locally in such a way that their modifications will only be updated when the file is opened.

2.3 Performance

Despite the issues about which a *distributed file system* should address, an acceptable performance also has to be achieved comparable with a local *file system* performance. To improve the *distributed file system* performance it is necessary to minimize the system throughput, since performance is understood as "the amount of time needed to satisfy a service requests" [8]. *Distributed file system* requests (file operations) are done using a network infrastructure to exchange messages between clients and servers (see figure 1) due to the distributed nature of the system. These messages increase the request overhead when comparing to a local *file system* request.

Request overhead. The figure 1 illustrates requests performed in a *distributed file system* and exposes their overhead. In local *file systems*, performance is measured accounting the time to access storage devices plus a little time of CPU usage. In *distributed file systems*, requests are transmitted over the network, so communication time is added to processing time. Moreover, to provide fault tolerant and scalable services, robust protocols must be implemented. These protocols also increase the request overhead since they will use network bandwidth and significant CPU cycles to work.

Implementation design. It is possible to implement *distributed file systems* using two different approaches: in *user* or *kernel* spaces. This choice also has an important on performance. If the first one is chose, the system will have more flexibility and a simpler design. Hence, it is easier to implement it in *user space*. On the other hand, *distributed file systems*

implemented in *kernel space* will probably be more efficient than other implemented in *user space*. This is achieved because *kernel space* approach takes advantage of a straight way to communicate to the operating system kernel, decreasing the system overhead. In spite of this benefit, maintaining such systems relied on a coupled design is more difficult due its complexity.

2.4 Caching

In *distributed file system* context, files are frequently accessed, thus the system will have to deal with a great number of requests. Furthermore, the scenario can be worst if the *distributed file system* is relied on a *remote service*. In this method of access, each request becomes a message to the server and its result is also sent as a message. Instead of always using remote accesses, it is encouraged to employ *caching* on clients to improve performance.

When *caching* is employed, a part of the file (or even the whole file) is copied. However, *caching* does not mean file replication - file replication will be explained in Setion 3.2. This part of the file (or block) is commonly recorded in main memory, besides other storage devices can also be used. *Caching* is very useful to provide fault tolerance and scalability. The details about how *caching* can be used in these contexts are exposed in sections 3.3 and 4.3 respectively.

2.4.1 *Caching* tuning

Once relied on the fact that file *caching* is impracticable because of their size, the block size (quantity of data) has to be defined. It is necessary to deal with the trade-off between the cache size and the block size. A small cache is suitable for small block sizes while larger caches are suitable for big size block. In other words, the size of the block have to be proportional to the cache space. Furthermore, dealing with large block sizes reduce the network overhead. Referring to *caching* location, if *caching* is performed in main memory, it will take advantage of its fast access. Otherwise, hard disk *caching* is persistence, in case of failures data will not be lost likewise it is possible to provide recovering mechanisms. Despite persistence, the usage of main memory is encouraged since *caching* aims performance.

Defining when the block will be updated, i.e, recorded into the file it belongs to, can be done in different ways. For example, we can update the file every moment the block is modified, update it later (maybe periodically), update the file when it is closed, etc. The approach used have to consider the network usage to avoid performance decaying besides to consider also how often blocks are modified (e.g., read operations may be commonly). Updating a cached block may imply in a consistency problem if the file was already written by another client. In this way, the server or the client have to monitor the caches to realize when the modification is done and then propagate the update. In both situations the period chose to check inconsistency have crucial impact to performance if it is set up a low one.

3 Fault Tolerance in Distributed File Systems

In brief, we can say that in a computational system data are processed to produce information. Once produced, usually these information are stored in a media by the *file system* for further accesses Their necessity of being accessible implies to provide safe mechanisms for

storing and accessing data/information which claims fault tolerance issues. Besides, in *distributed file systems*, not only local failures (e.g., due storage devices) should be dealt with, but also other failures inherent to the distributed environment. Following, we address some issues that are strict related with fault tolerance in *distributed file systems*.

3.1 *Stateful* and *stateless* services

To better understand how fault tolerance can be employed in a *distributed file system*, it is useful to understand how the services are provided. *Distributed file systems* services can be implemented by using two different service designs: *stateful service* or *stateless service*. These paradigms have contradictory concepts, however both supply the *file operations*.

Stateful service. In this type of service, information about *file operations* are kept in the server during all the *file session*. A communication channel is established between the client and the server when a the client explicitly solicits the file opening. A number (identifier) is used to define the communication channel then this identifier will be used to perform *file operations*. To attend its clients, the server copies data from the storage devices to memory and let them there till the file closing.

Stateless service. On the other hand, the *stateless service* does not establish a communication channel. Moreover, there is no necessity for explicit file opening and closing: before executing a file operation the server will automatically open and close the file. Each request sent to the server must define the desired file likewise, if a *read* or *write* operation is requested, it must contain the position in the file referring to the respective operation.

The usage of the main memory can improve performance whereas memory access is faster than disk one. The memory can be used for caching in the *stateless service* but its usage is not obligatory as in *stateful service*. As a result to this, *stateful service* presents an advantage when compared to the *stateless* one. On the contrary, the advantage of using the main memory becomes a disadvantage with respect to the fault tolerance context. If the server crashes, information stored in memory will probably be lost or at least harder to be recovered.

The consequences of servers and clients failures will depend of the used service. If a *stateless* server crashes, the previous *file sessions* will not be disturbed. On the other hand, if the server crashes in a *stateful service*, it should be able to recover the *file session* state, likewise it has inconveniences as mentioned earlier. Focusing on the client failures, in a *stateful service* the server should be able to realize when it happens to free the allocated memory. On the contrary, *stateless service* servers do not need to handle client faults. However, *stateless service* clients can experience a situation in which they can not distinguish a slow server from a recovering one.

Finally, we can also compare both approaches referring to service overhead. Due to the communication channel established on the *stateful service* its overhead is significant lower than the *stateless service*. The reason is that in a *stateful service* it is not necessary to send details about the *file operation* each time a request is sent. Additionally, such a service can be understood as a centralized and coupled service. In contrast, a *stateless service* is decentralized and decoupled that delegates service tasks to clients. The great decentralized participa-

tion of clients in *stateless service* allows to better provide fault tolerance mechanisms once it is possible to avoid single points of failures.

3.2 Replication

In distributed systems, replication techniques can be employed with different goals. Targeting consistency, for example, replication is useful for performance issues, e.g., accessing data at the same time or accessing the data copy whose network communication has a low latency. Replication can also be used for availability and fault tolerance [4], moreover, replication means file replication in *distributed file system* context. Furthermore, replication mechanisms should to keep replicas consistent even if they are used to provide fault tolerance. It implies a fault tolerance trade-off between consistency and performance [8]. The techniques used in replication should choose one of these characteristics to prioritize.

Consistency vs. performance. When a client performs a *file operation* the file changes have to be updated to all its replicas. Consistency protocols manage them, providing atomicity to replica updates. In other words, these protocols ensure that all replicas of a file will correspond to the last change done in it. To achieve this, consistency protocols must avoid that clients open outdated replicas. This task is not trivial for *distributed file systems* that deal with mutable data (files). By that very fact, such protocols increase the system overhead which can degrade system performance. Consequently, the harder the consistency protocol is, the lower the performance that the system will acquire [8].

File replication location. Another interesting aspect of file replication is the place where it will be replicated. Basically, it is possible to explore two approaches. The first one relies on replicating them on the same machine - be replicas on the same storage device or on different ones. This approach can profit a RAID scheme if relied on different medias (hard disks). In opposition, the second approach concerns replicas in different machines likewise using the network infrastructure to manage them. It is important to remember that despite the approach chosen, clients should not care about file replication, i.e., it should be transparent for them. However, details about file replication (e.g., number of replicas) can be exposed to clients if the *distributed file system* allows them to tune it by themselves.

Yet, there is the possibility to provide a hybrid approach that relies on file replication on different machines and also taking advantage of different medias (in some servers or in all of them). This scenario may increase the system throughput however, if well tuned, it can offer an environment to build robust fault tolerance services without degrading the system performance. Moreover, fault tolerance consistency protocols could improve their performance when relying on this hybrid approach. Fault tolerant mechanisms would profit from different machines to avoid single points of failure and the consistency protocol could be better performed relying on faster accesses to local storage devices.

3.3 Caching

If caching relies on a approach in which the server monitors the cache to initiate the update, it would be incompatible to implement a *stateless service*. In this case, the server should store information about the service that is running on the clients and this is contradictory to the *stateless service* concept. Moreover, delegating to the server the task of cache monitoring will make it a single point of failure. Another aspect between *caching* and fault tolerance is the place where the cache will be performed. Using persistence medias (e.g., hard disks)

to perform *caching* allows to implement recovering mechanisms. Even though, using hard disks to cache is less performance than main memory.

4 Scalability in Distributed File Systems

It is interesting for the *distributed file system* to support a significant number of users and also to support the system growth (accomplished by intermittent users or not). Targeting these issues, *distributed file systems* should be able to offer techniques to work in a large-scale environment without degrading performance as well as service availability. These techniques provide *scalability* to the system. Scalability is not only interesting to the *distributed file system*, but also necessary if one wants to expands its usage to a larger environment rather than LANs (e.g., MANs, WANs or even Internet). The incapacity of satisfying client requests can affect the *distributed file system* progression or even stop its services (as explained in Section 2.3). Intending to avoid this scenario, the system can handle scalability issues by implementing mechanisms based on different approaches which ones will be explained above.

4.1 Network

Mechanisms can rely on a limited service demand by defining a constant - which is independent of the system size. This approach is attractive because the growth capacity of the system does not depend of the number of its resources. Likewise, adding resources to the system is not a solution for scalability since the system will still have its growth capacity limited by resources [8]. In network context, broadcast communications are unacceptable in scalable systems designs because they require the participation of all nodes. Thus, we can extend the idea of limiting the service demand to the network link usage to avoid a high latency.

We can not suppose that *distributed file systems* will always operate in high speed networks. In this case, the system will have a strong constraint and it is not desired. When designing a distributed system, it is desirable to provide a solution that will work in most cases. Thereby, network bandwidth presents an obstacle to scalability. However, if the system relies on a relaxed semantics (e.g., *session semantics*, Section 2.2) network could not be a critical issue. Strong semantics (as *UNIX* one) are inversely proportional to scalability because it is hard to well tune a *distributed file systems* which implements it.

4.2 System design

As discussed in section 3, centralizing the system makes it more vulnerable to failures. When scalability is targeted, central points are not also compatible as they will probably become bottlenecks, avoiding the system growth. To circumvent this issue, a decentralized design can be proposed in such a way that system load could be balanced between nodes. Thereby, a *distributed file systems* based on *stateless service* is naturally more suitable to scalability rather than other that implements *stateful service*.

Scalability improvement still depends on how processes will be implemented. Choosing an unique process to attend users is not a interesting choice due to an obvious reason: the *distributed file systems* would have its performance degraded when waiting for I/O requests.

Additionally, implementing one process for each client request is still not interesting. Context switch between processes are expensive and it will be frequently when dealing with lots of user requests. This situation encourages threads usage. Threads context switches are less expensive comparing with processes ones, besides they share address space and operating system resources (e.g., I/O devices). Thereby, toward a scalable *distributed file system* threads have to be seriously considered in its design.

4.3 Caching

Remote accesses will be decreased if *caching* is used and well tuned in *distributed file system* as cited in section 2.4. As a result of this, servers and clients will not probably congest the network; in addition they will not become potential bottlenecks. These characteristics enhance scalability. Otherwise, if the system has a complex design, a good *caching* tuning may be difficult to achieve that allows the appearance of congestion points, thus, degrading scalability. Also, when caching a larger block size corresponding to the file, scalability issues can be improved. However, this idea is closer to the *session semantics* and maybe it could not be desired.

5 Fault Tolerance and Scalability: Cases Studies

5.1 NFS

Network File System (NFS) is a *distributed file system* protocol proposed initially by Sun Microsystems and since 1998 it has been defined by the IETF (Internet Engineering Task Force [2]). NFS protocol is current in its fourth version (NFSv4) and its main goal is to provide file access in a heterogeneous network independent of the operating system and transport layer [9, 3]. The NFS protocol implementations are also known as NFS [8] however when we refer to NFS, we mean the NFS protocol instead of any implementation - more precisely, the NFSv4. Nowadays, NFS is the most used *distributed file system* in organizational networks, usually LAN ones. This is often due the NFS interoperability among different systems and networks.

To improve performance, aggressive caching on clients is the main technique used in NFS . The server controls when the cache will be updated but it also allows clients to control it during a bounded time. The client manages its caching by comparing with server information if the cached data was changed since the file was opened. When the file is closed, any modification is sent to the server. In [9], it is assured that this consistency technique is sufficient for most applications and users.

5.1.1 Fault tolerance

NFSv4 provides a mechanism that allows to ensemble *file operations* in a single client request which receives a unique response from the server. With respect to performance, this mechanism seems to be attractive because it reduces the communication overhead. However, the mechanism is not an atomic which implies in a weak fault tolerance approach. When the server realizes a failure between two successive operations coupled in a single request, it delegates to clients the failure treatment by sending to it the error output.

NFS protocol changed its design from a *stateless service* (NFSv3) to a *stateful service*. In [9], it is argued that it was necessary due to problems when implementing file locking, file sharing compatible with Windows operating systems semantics and aggressive client caching. On the other hand, it suffers the disadvantages of a *stateful service*. To address fault tolerance issues, NFS uses the clientid to unlock both server and client. The clientid is sent by the server to the client when a file is opened.

If the server crashes, after its boot the server will send to the client another clientid and the latter will leave the locking state. If the clients crashes, after its boot it will acquire another clientid then the server is able to free the old state from its memory. Moreover, the locking mechanism is based on the transport layer. In spite of some techniques implemented to improve reliability when NFS is combined with unreliable transport protocols, it still misses robust fault tolerance mechanisms.

NFS uses a filehandle to uniquely identify a file in a server. In older NFS versions, the servers used persistent filehandles to send to clients. The NFSv4 introduces a volatile filehandle concept to provide interoperability among file systems. Then, clients map path file name and filehandle in cache. When the volatile filehandle expires, i.e., the file was renamed, deleted or it is not accessible anymore, the client receives an error and it deletes the mapping from memory. The volatile filehandle approach is weak because information stored in memory can be lost any time. Also, *file operations* done by other clients in the same file will imply on filehandle expiration. It is realized that NFS designers decided to sacrifice correctness to provide interoperability.

5.1.2 Scalability

NFS does not address scalability mainly due to its *stateful service*. Even though, the aggressive caching scheme used seems to decrease the system overhead, it is not enough to become NFS scalable. However, NFS can be adapted for using on Internet. For this, it is necessary to use the TCP (*Transmission Control Layer*) as transport layer because of its congestion control mechanism. Moreover, NFS defines that server will listen on port 2049, then firewalls can be easily configured for allowing or denying NFS services among networks.

5.2 GFarm

Grid Datafarm architecture aims on data-intensive computing by taking advantage of dispersed resources in a large-scale environment. For this, Gird Datafarm relies on a grid computing infrastructure that uses a *distributed file system* (in this also know as *grid file system*). GFarm is a *distributed file system* prototype that attempts to implement the Grid Datafarm architecture specifications. One of them, for example, is the capacity to provide not only the common *distributed file system* services, but also allow to use the nodes to compute the data that they store [10].

GFarm provides a virtualization scheme: files from different nodes (possibly from different *file systems*) can be represented by a unique GFarm file. This abstraction allows to better exploit the parallel access from node files, besides this abstraction also allows to exploit a good node locality. The parallel access of node files improves GFarm performance. However, it uses a I/O daemon called *gfsd* implemented in user space, which decreases the system performance. Moreover, GFarm current implementations runs only on UNIX-like

systems: GNU/Linux, FreeBSD and NetBSD [1]; however, it is possible to access the GFarm file system by using standard protocols as *scp*, GridFTP and SMB [10].

5.2.1 Fault tolerance

GFarm offers fault tolerance mechanisms as file replication and safe metadata update. The second one ensures that information about the file (metadata) and information about the physical file will be updated preserving consistency. In this respect, the *gfsd* daemon is responsible to close the file in a *file session* besides the metadata is also updated by the daemon if the client connection fails. Then, the metadata will be consistent even in presence of an application failure [10]. In spite of offering fault tolerance mechanisms, GFarm relies on a centralized approach to store the metadata. It uses a metadata server which represents a single point of failure to the system.

GFarm implements *session semantics* and it ensures that two clients will not simultaneously write on the same file. In spite of the *session semantics*, GFarm also provides a file lock mechanism in *write* and *read* operations for the whole file or a block. When the lock is used, the client stops caching the data (file or block), thus it is ensured that other clients can access updated contents. The file lock mechanism is not interesting for fault tolerance because the lock also relies on one single point: the blocking node.

5.2.2 Scalability

In GFarm, the main idea is to move processing to nodes instead of moving data to nodes. The criteria used to move the processing to nodes is the *file-affinity*, i.e., the machine that has the data needed to the processing. Thereby, the system will significantly decrease the communication rate and it will probably have its load balanced among the nodes. This scenario is certainly suitable for scalability.

However, GFarm has to obtain information about the nodes[1] to schedule the processes. This task will increase the system throughput. Also, the metadata server is a potential bottleneck which degrades the system scalability. These two issues plus the complexity of the GFarm design are inconveniences for the system scalability.

6 Conclusion

In *distributed file systems* context, the communication overhead and the overload on single nodes degrade the system performance. Targeting scalability in such systems, performance improvement have to be ensured since a node congestion can become a bottleneck in the system. The *caching* usage is encouraged to improve performance as well as replication technique. With respect to fault tolerance, not only bottlenecks have to be avoided, but also it is necessary to exploit system designs relied on *stateless services*, replication and recovering techniques.

In spite of efforts employed by current *distributed file system* implementations, their mechanisms to provide scalability in large-scale environments are still not satisfactory. As mentioned in Section 1, *distributed file systems* can take advantage of other distributed approaches

[1] As CPU usage and machine status, both monitored by the *gfsd* daemon.

(e.g. *grid data sharing services*) which offer scalability and, then, to result on a hybrid approach. In spite of complementarity of *distributed file systems* and *grid data sharing service*, these two approaches have common challenges to deal. Each one has solved them referring to their inherent environments likewise they can exchange ideas to take advantage of their existing mechanisms. Mainly, fault tolerance and scalability issues are a great point of exploring mutual benefits between them. The analysis of how the common tasks are related to each other makes clear how *grid data sharing service* systems can rely on *distributed file systems* persistence storage.

References

[1] Gfarm file system. `http://datafarm.apgrid.org/`, January 2008.

[2] Ietf, the internet engineering task force. `http://www.ietf.org/`, January 2008.

[3] Nfsv4. `http://www.nfsv4.org/`, January 2008.

[4] G. Antoniu, J.-F. Deverge, and S. Monnet. How to bring together fault tolerance and data consistency to enable grid data sharing: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(13):1705–1723, 2006.

[5] Gabriel Antoniu, Marin Bertier, Luc Bougé, Eddy Caron, Frédéric Desprez, Mathieu Jan, Sébastien Monnet, and Pierre Sens. *Future Generation Grids*, volume XVIII, CoreGrid Series of *Proceedings of the Workshop on Future Generation Grids November 1-5, 2004, Dagstuhl, Germany*, chapter GDS: An Architecture Proposal for a Grid Data-Sharing Service. Springer Verlag, 2006.

[6] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. Peer-to-peer distributed shared memory? In *Proc. IEEE/ACM 12th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT 2003), Work in Progress Session*, pages 1–6, New Orleans, Louisiana, September 2003.

[7] Foster Ian and Kesselman Carl. *The Grid: Blueprint for a New Computing Infrastructure*, chapter 2, pages 15–51. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[8] Eliezer Levy and Abraham Silberschatz. Distributed file systems: concepts and examples. *ACM Comput. Surv.*, 22(4):321–374, 1990.

[9] Brian Pawlowski, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler, David Noveck, David Robinson, and Robert Thurlow. The NFS version 4 protocol. *Proceedings of the 2nd international system administration and networking conference (SANE2000)*, page 94, 2000.

[10] Osamu Tatebe, Satoshi Sekiguchi, Youhei Morita, Noriyuki Soda, and Satoshi Matsuoka. Gfarm v2: A grid file system that supports high-performance distributed and parallel data computing. In *Computing in High Energy and Nuclear Physics (CHEP04)*, Interlaken, Switzerland, September 2004.