JuxMem: Weaving together the P2P and DSM paradigms to enable a Grid Data-sharing Service

Gabriel Antoniu, Luc Bougé, Mathieu Jan

▶ **To cite this version:**

Gabriel Antoniu, Luc Bougé, Mathieu Jan. JuxMem: Weaving together the P2P and DSM paradigms to enable a Grid Data-sharing Service. [Research Report] RR-5082, INRIA. 2004. inria-00071501

HAL Id: inria-00071501

https://inria.hal.science/inria-00071501

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

JUXMEM: *Weaving together the P2P and DSM paradigms to enable a Grid Data-sharing Service*

Gabriel Antoniu, Luc Bougé, Mathieu Jan

N°5082

Janvier 2004

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--5082--FR+ENG

INRIA RENNES

JuxMem: Weaving together the P2P and DSM paradigms to enable a Grid Data-sharing Service

Gabriel Antoniu, Luc Bougé, Mathieu Jan

Thème 1 — Réseaux et systèmes
Projet Paris

Rapport de recherche n˚5082 — Janvier 2004 — 22 pages

Abstract: Although *P2P systems* and *DSM systems* have been designed in rather different contexts, both can be usefully woven together in the specific context of Grid computing. In this paper, we address the design of such a hybrid Grid Data-sharing Service, with intermediate hypotheses and properties. This service decouples data management from grid computation, by providing *location transparency* as well as *data persistence* in a *fully dynamic environment*. We illustrate the proposed concept and demonstrate its feasibility by reporting on a software prototype called the JuxMem platform.

Key-words: Data-sharing, grid, peer-to-peer, hierarchical architecture, JXTA

(Résumé : tsvp)

This rapport is an extended version of the paper published in the Proceedings of the Workshop on Adaptive Grid Middleware (AGRIDM 2003), and has been selected for a publication in *Kluwer Journal of Supercomputing*.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

JUXMEM: combiner les paradigmes P2P et DSM pour construire un service de partage de données pour la grille

Résumé : Bien que les *systèmes pair-à-pair* et les *systèmes à MVP* ont été conçus dans des contextes différents, les deux peuvent être habillement combinés dans le contexte spécifique du calcul sur grille. Dans ce papier, nous décrivons une proposition d'architecture pour un tel service hybride de partage de données pour la grille, avec des hypothèses et des propriétés intermédiaires. Ce service découple la gestion des données du calcul sur grille, en fournissant une *localisation transparente* ainsi qu'un stocakge *persistant* des donnéees dans un *environnement hautement dynamique*. Nous illustrons le concept proposé et démontrons sa faisabilité à travers un prototype appelé JUXMEM.

Mots-clé : partage de données, grille, pair-à-pair, architecture hiérarchique, JXTA

1. Why combine DSM systems and P2P systems?

1.1. Peer-to-peer systems: high scalability on highly-dynamic configurations

The Peer-to-Peer (*aka* P2P) model [23, 21] has been made popular by systems allowing music files to be shared among millions of intermittently connected users (Napster [22], Gnutella [24], KaZaA [18], etc.). Such systems have demonstrated the adequacy of the peer-to-peer approach for data sharing on *highly-dynamic, large-scale configurations*. The underlying model of these systems is simple and complementary to the client-server model: the relations between machines are symmetrical, each node can act as a client in one transaction, and as a server in another. It has been demonstrated that such a model scales very well, without any need for a centralized storage server for the shared files: each client node is a server as well, and provides files to the other peer nodes. According to a recent survey, up to 4,500,000 users can interact simultaneously within the KaZaA network, sharing 900,000,000 files, amounting to 9 Peta-Bytes of data. This *high scalability* has drawn the attention of the distributed systems community, since it shows a way to make an important step forward in this field. Traditional distributed systems based on the client-server model have often shown limited scalability, generally due to centralized servers. By removing these bottlenecks, the peer-to-peer model not only enhances the system's scalability, but also improves its fault tolerance and availability despite the *high node volatility*. The system's activity is no longer dependent on the availability of a single server.

These important properties explain why the peer-to-peer model has attracted the interest of the scientific distributed systems community. Within this context, the research efforts have mainly focused on devising efficient peer-to-peer localization and routing schemes [26, 33, 35, 37, 29, 17, 8], based on the use of distributed hash tables (DHT). These schemes have been illustrated by systems like Chord [35], Tapestry [37] and Pastry [33], which serve as basic layers for higher-level data management systems, such as CFS [7], OceanStore and PAST [34], respectively.

On the other hand, we can note that these systems focus on sharing *immutable* files: the shared data are *read-only* and can thus be replicated at ease, without any limit on the number of copies. If the data were mutable, then a mechanism would obviously be necessary to maintain the consistency of the data copies. But, guaranteeing consistency would set a non-desired limit to scalability: the more the data copies, the higher the consistency overhead. Therefore, most peer-to-peer systems make a compromise: they favor scalability by sacrificing data mutability.

Recently, some mechanisms for sharing mutable data in a peer-to-peer environment have been proposed by systems like OceanStore, Ivy and P-Grid [9]. In OceanStore, consistency is managed at the level of each single data. Only a small set of primary replicas, called the *inner ring*, agrees, serializes and applies updates. Updates are then multicast down a dissemination tree to all other cached copies of the data, called *secondary replicas*. However, OceanStore uses a versioning mechanism which has not been demonstrated to be efficient at large scale, and published measurements on the performance of updates only assume a single writer per data block. Although OceanStore provides hooks for managing the consistency of data, applications still have to use low-level mechanisms for each specific consistency model [32]. Finally, servers making up inner rings are assumed to be highly available.

The Ivy system is built on top of the Chord [35] localization layer. It also addresses the management of mutable data. Yet, its scalability is also limited by a number of factors. The root block of the whole file system needs to be redefined each time a new node is accepted in the system. The vector clocks used to globally order the file system modifications include one entry for each node in the system. This is incompatible with a variable number of nodes and therefore unsuitable for a configuration with a large number of volatile nodes. In addition, each node creates a private snapshot of the whole file system, which limits again the scalability of the system. Finally, the applications have to explicitly synchronize the possible conflicting writes, so that the number of potentially concurrent writers per data should remain limited.

P-Grid introduces a flooding-based algorithm to update data, but assumes no conflicting writes. Only partial results have been published so far for this system. We can clearly conclude that handling consistency is a serious problem for peer-to-peer systems: the preliminary solutions tentatively proposed so far share the major drawback of limiting the system scalability, which is the main property which makes peer-to-peer systems so attractive.

1.2. Distributed Shared Memory: consistency and transparency

The problem of sharing mutable data in distributed environments has been intensively studied during the past fifteen years within the context of Distributed Shared Memory (*aka* DSM) systems [20, 27, 2]. These systems provide transparent data sharing, via a unique address space accessible to physically distributed machines. As in the case of peer-to-peer systems, reading data on multiple nodes may result in data replication. But the DSM nodes can also modify the data, and this results in triggering some consistency action (e.g., invalidation or update), according to some consistency protocol which implements a well-defined consistency model. A large variety of DSM consistency models, and for each model, of protocols protocols [27, 13, 6, 4, 14, 38] have been defined [28]. They provide various compromises between the strength of the consistency guarantees and the efficiency of the consistency actions. These efforts have been carried out within the context of research on high-performance parallel computing, most often with the goal of providing maximum transparency at a minimum cost.

A central feature of DSM systems is *transparency*. First, these systems provide *transparent access* to data: all nodes can read and write any shared data in a uniform way, should the data be local or remote. The DSM system internally checks for data locality and takes the appropriate action in order to satisfy the access. Second, DSM systems also provide *transparent localization*: if the program accesses remote data, then it is the responsibility of the DSM system to localize, transfer or replicate it, according to the specific consistency protocol.

However, existing DSM systems have generally demonstrated satisfactory efficiency (i.e., near-linear speedups) only on small-scale configurations: in practice, up to a few tens of nodes [28]. This is often due to the intrinsic lack of scalability of the algorithms used to handle data consistency. These algorithms have often been designed by assuming a small number of copies per shared data. For instance, Multiple-Reader-Single-Writer algorithms [20] clearly cannot perform well at large scale, since any write operation on some data results in an expensive invalidation of *all* existing data

Table 1: A grid data sharing service as a compromise between DSM and P2P systems.

	DSM	**Grid Data Service**	**P2P**
Scale	10^1–10^2	10^3–10^4	10^5–10^6
Topology	Flat	Hierarchical	Flat
Resource control and trust degree	High	Medium	Null
Dynamicity	Null	Medium	High
Resource homogeneity	Homogeneous (clusters)	Rather heterogeneous (federation of clusters)	Heterogeneous (Internet)
Data type	Mutable	Mutable	Immutable
Application complexity	Complex	Complex	Simple
Typical applications	Scientific computation	Scientific computation and data storage	File sharing and storage

copies. In the same way, Home-Based, Multiple-Writer algorithms [38] also rely on having the home node *centralize* and merge data modifications from *all* writers. On the other hand, an overwhelming majority of protocols assume a static configuration where nodes do not disconnect nor fail: the unique writer of a given data is not supposed to go down, nor is the home node in a home-based DSM. Only a few DSM systems have integrated some mechanisms for fault tolerance [36, 19]. However, nodes failures are supposed to be infrequent and are considered as an *exceptional behavior*. This is to be contrasted with the basic hypotheses of peer-to-peer systems, in which nodes are assumed to join and leave the network at any time, as a *regular behavior*. Getting DSM *highly scalable* and adaptive to *dynamic configurations* is therefore a real challenge.

1.3. Hybrid approach: a data sharing service for scientific computing

Although *P2P systems* and *DSM systems* have been designed in rather different contexts, we think that both can serve as major sources of inspiration for the design of a *hybrid* data sharing system. If DSM systems can usually handle configurations of *tens or hundreds* of nodes, corresponding to *cluster computing*, peer-to-peer systems generally target configurations of *millions* of nodes, corresponding to the scale of the *Internet*. The hybrid data sharing system we propose targets configurations of *thousands to tens of thousands* of nodes, which corresponds precisely to the scale of *grid computing* [11].

Therefore, we think the adequate approach for the design of such a system is not to build a *peer-to-peer DSM*, nor a *shared-memory peer-to-peer system*, but rather a *data sharing service for grid computing*. Such a service has to address the problem of managing *mutable data* on *dynamic, large-scale configurations*. The approach we propose benefits both from DSM systems (transparent

access to data, transparent data localization, consistency protocols) and from P2P systems (scalability, support for resource volatility and dynamicity).

These two classes of systems have been designed and studied in very different contexts. In DSM systems, the nodes are generally under the control of a single administration, and the resources are trusted. In contrast, P2P systems aggregate resources located at the edge of the Internet, with no trust guarantee, and loose control. Moreover these numerous resources are essentially heterogeneous in terms of processors, operating systems and network links, as opposed to DSM systems, where nodes are generally homogeneous. Finally, DSM systems are typically used to support complex numerical simulation applications, where data are accessed in parallel by multiple nodes. In contrast, P2P systems generally serve as a support for storing and sharing immutable files. These antagonist features are summarized in the left and right columns of Table 1.

A data sharing service targets physical architectures with *intermediate* features between those of DSM and P2P systems. It addresses scales of the order of thousands or tens of thousands of nodes, organized as a federation of clusters, say tens or hundreds of hundred-node clusters. At a global level, the resources are thus rather heterogeneous, while they can probably be considered as homogeneous within the individual clusters. The control degree and the trust degree are also intermediate, since the clusters may belong to different administrations, which set up agreements on the sharing protocol. Finally, the service targets numerical applications like heavy simulations, run by coupling individual codes. These simulations process large amounts of data, with significant requirements in terms of data storage and sharing. These intermediate features are illustrated in the middle column of Table 1.

The main contribution of such a service is to decouple data management from grid computation, by providing *location transparency* as well as *data persistence* in a *dynamic environment*. This approach is to be contrasted the widely-used approach to data management for grids based on *explicit data transfers* between clients and computing servers (e.g., GridFTP [1] or IBP [3]). As explained in the following scenarios, the service we propose can prove helpful for heavy numerical simulations based on code coupling, with significant requirements in terms of data storage and sharing.

2. Designing a data sharing service for the grid

2.1. A data sharing service for the grid: sample scenarios

Persistence. Since grid applications can handle large masses of data, data transfer among sites can be costly, in terms of both latency and bandwidth. In order to limit these data exchanges, the data sharing service has to rely on strategies able to: 1) reuse previously produced data; 2) trigger "smart" pre-fetching actions to anticipate future accesses; and 3) provide useful information on data location to the task scheduler, in order to optimize the global execution cost.

Let us consider the following scenario, which illustrates the first point mentioned above. A client submits a computation $C = f_1(A, B)$ to the grid infrastructure. The execution is scheduled on server S_1. To run this computation, the client needs to transfer A and B (say, possibly very large matrices)

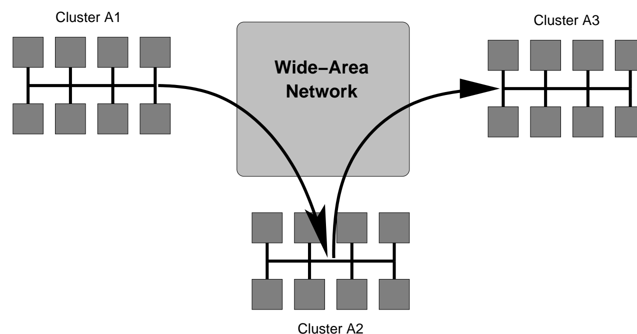

Figure 1: Numerical simulation for weather forecast using a pipeline communication scheme with 3 clusters.

to S_1. At the end of the computation, C is transferred back from S_1 to the client. Now, imagine that the client has to execute a second computation $D = f_2(B, C)$ on the same server. Then, the client would have to transfer B and C again to S_1. To avoid such unnecessary transfers, the data sharing service should provide *persistence*, thereby allowing to reuse the matrices already present on the storage infrastructure. The client should then be able to specify its persistence requirements for each data stored by the service.

Transparency. Another desirable feature for a data sharing service is *transparency with respect to data localization*: the service user should not explicitly handle data transfers between storage servers, but rather leave it to the service.

Let us consider a scenario in which a distributed federation of 3 clusters, A_1, A_2 and A_3, co-operate together as shown on Figure 1. Typically, each single cluster is built over a high-performance, local-area network, whereas the clusters are interconnected together through a regular, wide-area network. Consider for instance a weather forecast simulation. Cluster A_1 may compute the forecast for a given day, then A_2 for the next day, and eventually A_3 for the day after. Thus, A_3 uses data produced by A_2, which in turn uses data produced by A_1, as in a pipeline. To communicate data from A_1 to A_2, the usual approach [1] consists in writing data on a hard disk of A_1, then use some FTP-like tool to transfer them on a disk of A_2. This send-receive method requires an *explicit* participation of the application. Besides, it obviously does not scale: if multiple servers get involved in the co-operation, the management of communication and synchronization grows quickly very complex. In contrast, applications should be able to directly read/write data from/to a data sharing service, which is in turn in charge of *transparently* localizing and transferring data.

Automatic redistribution. Numerical grid applications usually manipulate *structured* data: matrices, meshes, etc., which can be distributed on multiple nodes. Descriptive informations about how data are structured and distributed and about the access patterns used by the applications can equally

help the service improve its performance, thanks to appropriate pre-fetching schemes. For example, when an element of some matrix distributed on a given cluster is accessed by a node in a remote cluster, this could trigger the matrix transfer to this latter cluster, including on-the-fly redistribution if necessary.

Let us consider again the pipeline scheme on the 3-cluster federation described in the previous scenario. Let us now assume that application A_1 uses a block data distribution, A_2 uses a cyclic distribution and A_3 uses a block-cyclic distribution. Basic communication strategies available in existing grid environments, based on explicit transfers, would clearly make the application code *explicitly* implement very complex communication patterns. Here again, a data sharing service can make an extra step forward toward transparency by providing facilities for automatic data redistribution. The application code is then greatly simplified.

2.2. The JXTA implementation framework

Our proposal is partly inspired by the P2P approach, and it can thus usefully benefit from a platform providing basic mechanisms for peer-to-peer interaction. To our knowledge, the most advanced implementation platform in this area is JXTA [15]. The name JXTA stands for *juxtaposed*, in order to suggest the juxtaposition rather than the opposition of the P2P and client-server models. JXTA is a project originally initiated by Sun Microsystems. (We consider Version 2.0 of the JXTA protocols specification.)

JXTA is an open-source framework, which specifies a set of language- and platform-independent XML-based protocols [16]. JXTA provides a rich set of building blocks for the management of peer-to-peer systems: resource discovery, peer group management, peer-to-peer communication, etc. The data sharing service that we propose is designed using the following JXTA building blocks.

Peers. The basic entity in JXTA is the *peer*. Peers are organized in networks. They are uniquely identified by IDs. An ID is a logical address independent of the location of the peer in the physical network. JXTA introduces several types of peers. The most relevant as far as we are concerned are the *edge peers* and *rendezvous peers*. Edge peers are able to communicate with other peers in the JXTA virtual network. Rendezvous peers have the extra ability of forwarding the requests they receive to other rendezvous peers. Joining, leaving, and even unexpected failing of edge peers or rendezvous peers are supported by the JXTA protocols.

Peer groups. Peers can be members of one or several *peer groups*. A peer group is made up of several peers that share a common set of interests, e.g., peers that have the same access rights to some resources. The main motivation for creating peer groups is to build services collectively delivered by peer groups, instead of individual peers. Indeed, such services can then tolerate the loss of peers within the group, as its internal management is not visible to the clients.

Pipes. Communication between peers or peer groups within the JXTA virtual network is made by using *pipes*. Pipes are unidirectional, unreliable and asynchronous logical channels. JXTA offers two types of pipes: point-to-point pipes, and propagate pipes. Propagate pipes can be

used to build a multicast layer at the virtual level. Peers can dynamically connect on existing pipes in order to send or receive data.

Advertisements. Every resource in the JXTA network (peer, peer group, pipe, service, etc.) is described and published using *advertisements*. Advertisements are structured XML documents which are published within the network of rendezvous peers, using a *distributed hash table* (DHT) scheme. To request a service, a client has first to *discover* a matching advertisement using specific localization protocols.

JXTA protocols. JXTA proposes six generic protocols. Out of these, two are particularly useful for building higher-level peer-to-peer services: the *Peer Discovery Protocol*, which allows for advertisement publishing and discovery; and the *Pipe Binding Protocol*, which dynamically establishes links between peers communicating on a given pipe.

3. JUXMEM: a supportive platform for data sharing on the grid

The software architecture of the data sharing service, mirrors a hardware architecture consisting of a federation of distributed clusters. The architecture is therefore *hierarchical*, and is illustrated through the proposition of a software platform called JUXMEM (for *Juxtaposed Memory*). Its ultimate goal is to provide a data sharing service for grid computing environments, like DIET [5].

3.1. Hierarchical architecture

Figure 2 shows the hierarchy of the entities defined in the architecture of JUXMEM. This architecture is made up of a network of peer groups (`cluster` groups A, B and C), which generally correspond to clusters at the physical level. All the groups are inside a wider group which includes all the peers which run the service (the `juxmem` group). Each `cluster` group consists of a set of nodes which provide memory for data storage. We will call these nodes *providers*. In each `cluster` group, a node is in charge of managing the memory made available by the providers of the group. This node is called *cluster manager*. Finally, a node which simply uses the service to allocate and/or access data blocks is called *client*. It should be stressed that a node may at the same time act as a cluster manager, a client, and a provider. However, each node only plays a single role in the example illustrated on the figure for the sake of clarity.

Each block of data stored in the system is associated to a group of peers called `data` group. This group consists of a set of providers that host copies of that very data block. Note that a `data` group can be made up of providers from different `cluster` groups. Indeed, a data can be spread over on several clusters (here A and C). For this reason, the `data` and `cluster` groups are at the same level of the group hierarchy. Note that the `cluster` groups could also correspond to subsets of the same physical cluster.

Another important feature is that the architecture of JUXMEM is dynamic, since `cluster` and `data` groups can be created at run time. For instance, a `data` group is automatically instantiated for each block of data inserted into the system.

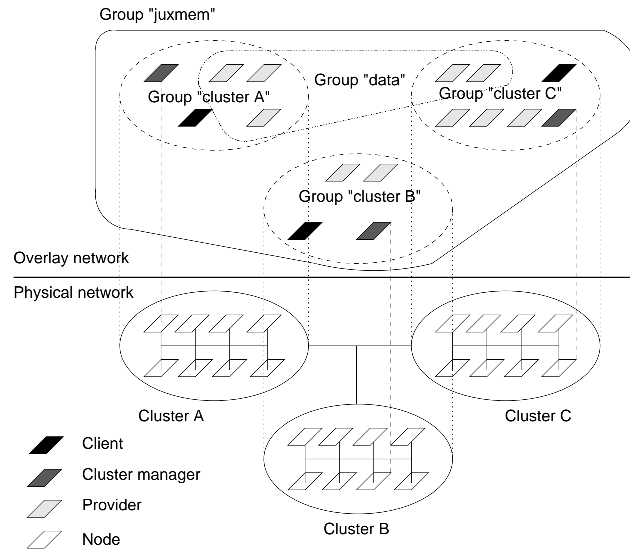

Figure 2: Hierarchy of the entities in the network overlay defined by JUXMEM.

Using the data sharing service. The Application Programming Interface (API) provided by JUXMEM illustrates the functionalities of a data sharing service providing data persistence as well as transparency with respect to data localization.

`id=alloc(size, attributes)` allows to create a memory area of the specified `size` on a cluster. The `attributes` parameter allows to specify the level of redundancy and the default protocol used to manage the consistency of the copies of the corresponding data block. This function returns an ID which can be seen at the application level as a data block ID.

`map(id, attributes)` allows to retrieve the advertisement of a data communication channel which has to be used to manipulate the data block identified by `id`. The `attributes` argument allows to specify requirements regarding the data block. For instance, the client may wish to specify the level of consistency: some clients may require weaker, but cheaper consistency requirements than the default one.

`put(id, value)` allows to set the data block identified by `id` to some `value`.

`get(id)` allows to get the current value of the data block identified by `id`.

`lock(id)` allows to lock the data block identified by `id`. A lock is implicitly associated to each data block. Clients accessing a shared data block need to synchronize using this lock.

`unlock(id)` allows to unlock the data block identified by `id`.

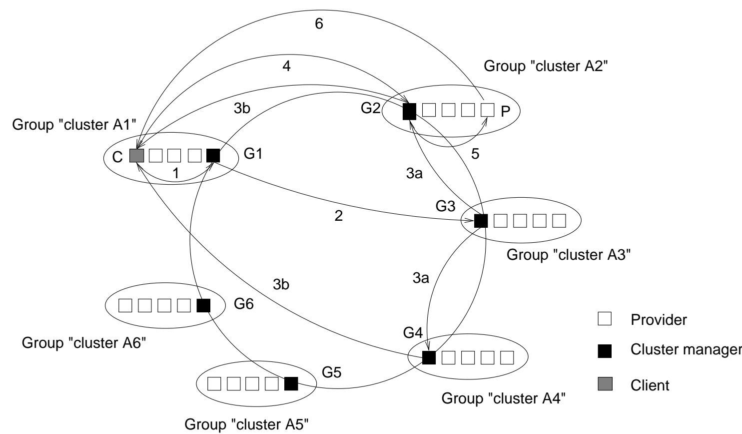

Figure 3: Steps of an allocation request made by a client.

`reconfigure(attributes)` allows to dynamically reconfigure a node. The `attributes` parameter allows to indicate if the node is going to act as a cluster manager and/or as a provider. If the node is going to act as a provider, then the `attributes` parameter also allows to specify the amount of memory that the node provides to JUXMEM.

3.2. Managing memory resources

Publishing and distribution of resource advertisements. Memory resources are managed using *advertisements*. Each provider publishes the amount of memory it offers within the `cluster` group to which it belongs, by the means of a *provider advertisement*. The cluster manager of the group stores all such advertisements available in his group. It is also responsible for publishing the amount of memory available in the cluster by using a *cluster advertisement*. This advertisement lists the amounts of memory offered by the providers of the associated `cluster` group. These cluster advertisements are published within the `juxmem` group, so that they can then be used by all the clients in order to allocate memory.

Cluster managers are in charge of co-ordinating the local `cluster` group with the overall `juxmem` group. They make up a network organized using a DHT at the level of the `juxmem` group, to build the frame of the data sharing service. This frame is represented by the ring on Figure 3. Each cluster manager G_1 to G_6 is responsible for a specific cluster, respectively A_1 to A_6, each of which is made up of 5 nodes. At the level of the `juxmem` group, the DHT works as follows. Each cluster advertisement contains a list which enumerates the amounts of memory available in the cluster. Each individual amount is separately used to generate an ID, by means of a hash function. This ID is then used to determine the cluster manager responsible for all advertisements having this amount of available memory in their list. Observe that this cluster manager is not the peer that actually stores

the advertisement: it only knows of the cluster manager which published it in the JUXMEM network. This indirect management of cluster advertisements allows clients to easily retrieve advertisements in order to allocate memory: any request for a given amount of memory is directed to the cluster manager responsible for that amount of memory, using the hash mechanism described above.

Searching for advertisements is therefore fast, and responses are exact and exhaustive, e.g., all the advertisements that include the requested memory size will be returned. But since using a DHT on memory sizes means to generate a different hash for each memory size, JUXMEM uses a parameterizable policy for the discretization of the space of memory sizes. JUXMEM will search for the least memory size allowed by the policy, that is greater to the one requested by clients. For instance, consider a client wanting to allocate a 1280-Byte memory area. In the current prototype version, JUXMEM uses powers of 2 for space discretization. It will thus internally and automatically search for a memory area of 2048 Bytes. Providers internally use the same law when offering memory areas, and provide the maximum memory size allowed by the policy that fits within their capacity.

Our central objective is to support the volatility of nodes which make up the clusters. The advertisements published at some time t_1 may become invalid at a later time $t_2 > t_1$, since providers may disappear from JUXMEM at any time. The mechanism used to manage such a volatility is based on republishing the cluster advertisements whenever a modification of the amount of memory provided is detected. Also, advertisements have a limited but parameterizable lifetime, so it is anyway necessary to periodically republish them.

Processing an allocation request. Clients issuing an allocation request specify the size of the required memory area. The successive steps for such a request are displayed on Figure 3.

1. Client C of Cluster Group $A1$ wants to allocate a memory area of 8 MB with a redundancy degree of 2. Consequently, it submits its request to Cluster Manager G_1, to which it is connected.
2. Cluster Manager G_1 then determines the peer responsible of advertisements having a memory size of 8 MB in their list, say, Cluster Manager G_3. This uses the hash mechanism described previously. Then, Cluster Manager G_1 forwards the request to G_3.
3. Cluster Manager G_3 then determines that Cluster Managers G_2 and G_4 both match the criterion of the client. It requests them to forward their cluster advertisement to Client C.
4. Client C then selects Cluster Manager G_2 as the peer having the "best" advertisement: for instance, the underlying cluster offers a higher degree of redundancy than the cluster handled by Cluster Manager G_4. Then, it submits its allocation request to G_2.
5. Cluster Manager G_2 receives the allocation request and handles it. If it can satisfy the request, then it requests one of its providers, say, P, to allocate a 8 MB memory area. If the request cannot be satisfied, then an error message is sent back to the client.
6. If Provider P can satisfy this request, then it creates a 8 MB memory area, and it sends back the advertisement of this memory area to Client C. P becomes the data manager of the

associated `data` group, which means that it is responsible for requesting other providers to join this group, to meet the redundancy degree specified by the client. If Provider P cannot satisfy the request, then an error message is sent back to Cluster Manager $G2$, which can then select another provider peer within the `cluster` group.

If no providers can be found in the last step of an allocation request, then an error message is sent back to the client. The client can then restart the allocation request from Step 4, e.g., with another `cluster` manager matching the requested memory size. Finally, if no cluster manager can allocate the requested memory area, then the client increases the requested memory size, and restarts the allocation request from the beginning. This can be done up to N times (for instance, $N = 3$) until the request is satisfied, or an error is reported at the application level.

3.3. Managing shared data

When a memory area is allocated by a client, a `data` group is created by the selected provider and a data communication channel advertisement is sent to the client. This advertisement allows the client to communicate with the `data` group. This advertisement is published at the `juxmem` group level, but only the ID of this advertisement is returned at the application level. Access to data by other clients is then possible by using this ID without having to worry about the data localization: the platform *transparently* locates the requested data block.

Storage of data blocks is *persistent*. Indeed, when clients disconnect from JUXMEM, data blocks still remain stored in the data sharing service on the providers. Consequently, clients can access data blocks previously stored by other clients: they simply need to look for the advertisement of the data communication channel associated with the data block (whose ID is assumed to be known). This is the role of the `map` primitive of JUXMEM, which takes as input the ID of the data block. Clients wishing to share data blocks with other peers simply need to explicitly exchange the ID of these data blocks.

Processing a `map` request. The successive steps for a `map` request are displayed on Figure 4.

1. Client C of Cluster Group A_1 wants to access a previously stored Data Block D, whose ID is I (which is a data communication channel advertisement). This data block is stored on some providers of Cluster Group A_2. Client C has first to to retrieve the advertisement. Thus, it submits its request to Cluster Manager G_1, to which it is connected.

2. By using the same hash mechanism as the one used in the distribution of `cluster` advertisements, Cluster Manager G_1 determines that the peer responsible for Advertisement I is Cluster Manager G_4. Therefore, G_1 forwards the request to G_4.

3. Cluster Manager G_4 then determines that Cluster Manager G_2 holds Advertisement I in its cache. Therefore, G_4 requests G_2 to forward the data communication channel advertisement to Client C.

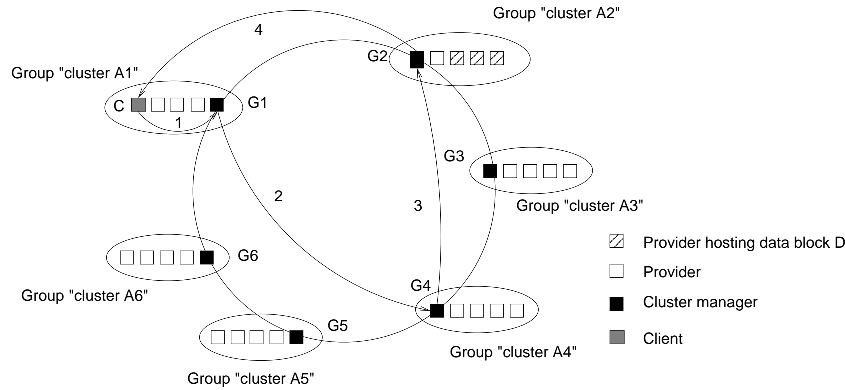

Figure 4: Steps of a `map` request made by a client.

4. The client now holds the data communication channel advertisement of Data Block D in its cache. It can then use it to send requests to read, write, lock or unlock this data block.

Managing consistency. Each data block is replicated on a fixed, parameterizable number of providers for a better availability. This redundancy degree is specified as an attribute at allocation time. The consistency of the different copies must then be handled. In this preliminary version of JUXMEM, this is addressed using the JXTA multicast primitive at the level of the `juxmem` group. The various copies of a same data block are simultaneously updated whenever a writing access is made. Alternative consistency models and protocols will be experimented in further versions. Observe that clients which have previously read the data block are not notified of this update, as they do not store a physical copy of the data block, but only an ID. It is up to the clients to take the adequate actions, possibly by locking the data block. It is worth stressing that this very difference between the clients and the providers allows to handle a high number of clients without having to deal with a high number of copies of data blocks. Synchronization between clients which concurrently access a data block is handled using the `lock`/`unlock` primitives.

3.4. Managing volatility

Handling volatile providers. Our objective is to tolerate the volatility of providers. A static replication of data on a fixed and parameterizable number of providers is not enough. Indeed, the providers hosting a copy of the same data block can successively become unavailable. A dynamic monitoring of the number of backup copies is therefore needed for each data. Each `data` group is therefore equipped with a *data manager*. It is a member of the group which is in charge of monitoring the level of redundancy of the data block. If this number goes below the one specified by clients, the data manager requests an alternative provider to host an extra copy of the data block.

Now, observe that this fresh backup copy has to be installed in an atomic way, to prevent concurrent modifications. The data manager must first lock it (internally) in order to maintain consistency. The provider which will host this new copy is responsible for eventually unlocking it. A *timeout* mechanism followed by a *ping* test is used to detect if the provider became unavailable before unlocking the data block. If it is the case, then the data manager unlocks the data block for its sake, and restarts the procedure.

Handling volatile managers. If a cluster manager goes down, then all resources provided by a whole cluster may become unavailable. The role of the *main cluster manager* is therefore automatically duplicated on another provider of the cluster, called the *secondary cluster manager*. These two managers periodically synchronize together using a mechanism based on the exchange of provider advertisements, in order to find out new advertisements published. They can thus both update the amount of memory available in the cluster. A mechanism based on periodical heartbeats allows to dynamically ensure this duplication of cluster managers. A similar mechanism is also used for the data managers (see Section 3.4). Note that the possible replacements of managers in the `cluster` group and the `data` group are not detected outside these groups. The availability of clusters and of data blocks is thus maximized, whereas the perturbation on the client side is minimized.

4. Implementation and preliminary evaluations

4.1. Implementation of JUXMEM within the JXTA framework

We have implemented a prototype of the software architecture described in the previous section. This JUXMEM prototype is based on the JXTA generic peer-to-peer framework (see Section 2.2). It uses the reference Java binding of JXTA, which is today the only binding compatible with the JXTA 2.0 specification. JUXMEM is entirely written in Java and includes about 50 classes (5000 code lines).

JXTA proves to be a very convenient support for JUXMEM. The managers of `data` and `cluster` groups are based on JXTA's *rendezvous peers*. Indeed, managers in charge of a cluster or a block of data have to make sure that their associated providers remain alive. They use a periodic ping test. This can only be done if providers have previously published their advertisements on the managers, which can then extract the address of each provider. Only JXTA's rendezvous peers can forward requests within the JXTA network; these peers thus play the role of *main managers*. For instance, the data managers have to forward the access requests issued by clients, to providers hosting a copy of the data block. In the same way, the cluster managers have to forward the allocation requests issued by clients, to the providers. Clients and providers which do not act as data managers for blocks of data are based on JXTA's *edge peers*. Indeed, they do not have to play any role in the dynamic monitoring of the number of backup copies associated with a block of data. Therefore, they do not have to store published provider advertisements. Moreover, the clients only need to discover and store the cluster advertisements which will allow them to allocate memory areas. The various groups defined in JUXMEM are implemented by JXTA's peer groups. The

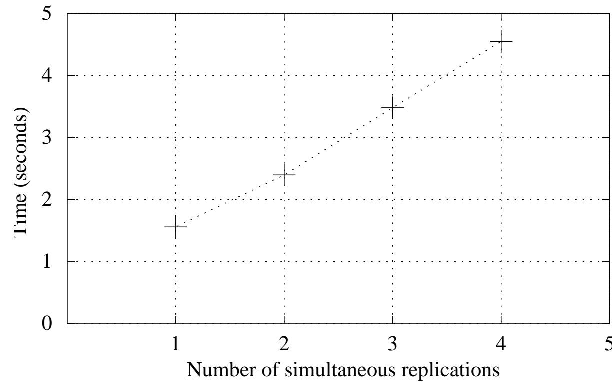

Figure 5: Time needed to simultaneously replicate α copies of a data block

`juxmem` group implements a JXTA peer group service, thereby providing the API of JUXMEM (see Section 3.1). Finally, the communication channels of JXTA also offer a convenient support to build multicast communications for simultaneously updating copies of the same block of data.

4.2. Preliminary evaluations

We report on preliminary experiments run on a cluster of 450 MHz Pentium II nodes with 256 MB RAM, interconnected by a regular 100 MB/s FastEthernet local area network.

We first measured the memory consumption overhead generated by the various JUXMEM peers with respect to the underlying JXTA peers. This overhead remains small: it ranges between 5% and 8%.

We then measured the time it takes for JUXMEM to replicate a data block (1 byte) with a redundancy degree α ranging from 1 to 4. As explained in Step 6 of the allocation request (see Section 3.2), the data manager of the data group is responsible for concurrently requesting other providers to join the data group according to the specified redundancy degree. In order to get accurate times, the sequence executed to maintain this redundancy degree is averaged over 100 iterations. At the beginning of each iteration, only the data manager holds a copy of the block. Figure 5 shows the time in seconds needed to find α fresh providers, and to simultaneously replicate a copy of the data block on each of them. As expected, the curve is linear with the number of copies to replicate: it takes 1.6 s for $\alpha = 1$, 2.4 s for $\alpha = 2$, etc.

Finally, we measured the influence of the volatility degree of providers on the duration of a sequence `lock-put-unlock` repeatedly executed in a loop by a client on a given data block stored in JUXMEM. The goal of this measure is to evaluate the relative overhead generated by the replications which take place in order to maintain a given redundancy degree. These replications are transparently triggered when the service detects that a provider holding a data block goes down. The data is

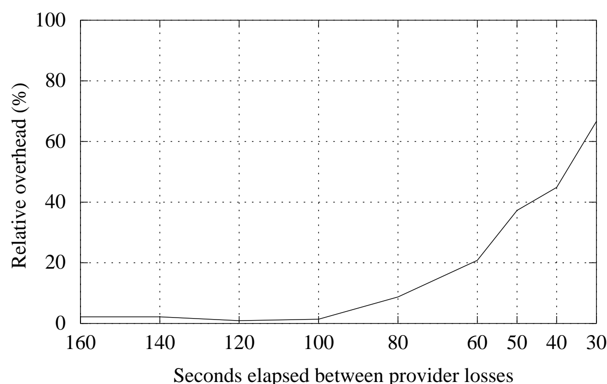

Figure 6: Relative overhead due to the volatility of providers for a sequence `lock-put-unlock`, with respect to a stable system.

then locked by the manager until a new backup copy has been installed on a fresh provider. If this takes place while the client accesses attempts to lock the data block, then the client is delayed until the replication procedure has completed.

The test program first allocates a small memory area (1 byte) on a provider, and writes it to a data block, with a redundancy degree of 3. The allocation takes place on a cluster initially consisting of 16 providers and one cluster manager. Each peer, the client, the providers, and the manager, are installed on separate nodes for the sake of the experiment. The client executes a 100-iteration loop, whose body is a sequence `lock-put-unlock` on the data.

During the execution of this loop, a random provider hosting a copy of the data is killed every δ seconds, where δ is a parameter of the experiment. In order to specifically measure the overhead due to the volatility of providers, the data manager of the associated group is never killed.

Figure 6 displays the relative overhead with respect to a stable system (i.e, where no provider goes down during the loop execution: $\delta = \infty$). The curve profile is explained by the number of times the system replicates the data on providers, in order to maintain the redundancy degree specified by the client (3 in this case). For the whole duration of our test, the number of replications is given in the Table 2 as a function of the δ parameter.

For highly volatile systems ($\delta < 80\,\mathrm{s}$), the number of replications triggered becomes higher than 2 and the relative overhead becomes significant. For $\delta = 30\,\mathrm{s}$, it reaches more than 65% (10 replications triggered). However, in a realistic situation, the node volatility on the architecture we consider is typically a lot weaker ($\delta \gg 80\,\mathrm{s}$). For such values, the reconfiguration overhead is less than 5%. We can reasonably conclude that the JuxMem platform includes a mechanism which allows to *dynamically* maintain a certain redundancy degree for data blocks, in order to improve data availability, *without significant overhead*, while authorizing node failures.

Table 2: Number of triggered replications when the volatility of provider peers evolves from 160 to 30 seconds.

Seconds	160	140	120	100	80	60	50	40	30
Number of triggered replications	1	1	1	1	2	2.5	5	5.5	10

5. Conclusion

We introduced the concept of a *Data Sharing service*, as a hybrid approach weaving together the DSM and P2P paradigms. We show that such a concept fits medium-scale Grid architectures with *intermediate* features between those of small-scale, homogeneous, static clusters and large-scale, heterogeneous, dynamic Internet. The contribution of this paper is namely to propose an architecture for such a Grid Data sharing Service, which addresses the problem of managing *mutable data* on *dynamic*, *large-scale configurations*. Our architecture is *hierarchical*: it addresses a target grid infrastructure consisting of a federation of clusters. Not only the architecture allows to reduce the number of messages to search for a piece of data, thanks to a hierarchical search scheme, but it also allows to take advantage of specific features of the underlying physical architecture. The management policy for each cluster can be specific to its configuration, for instance in terms of the network links to be used. For instance, some clusters could use high-bandwidth, low-latency networks for intra-cluster communication, if ever available.

The proposed architecture has been demonstrated through the JUXMEM platform. The JUXMEM user can allocate memory areas in the system, by specifying an area size and some attributes (e.g., the redundancy degree or the consistency protocol). The allocation primitive returns an ID which identifies the block of data. Then, data localization and transfer is fully transparent: the ID is sufficient to allow any peer to access and manipulate the corresponding data wherever it is: no IP address, nor port number needs to be specified at the application level.

Our architecture supports the volatility of all types of peers. This kind of volatility is also supported in peer-to-peer systems such as Gnutella or KaZaA, which enhance data availability thanks to redundancy. However, this is in their case a side effect of the user actions. In contrast, our system takes into account this volatility *actively*: this not only allows to maintain a certain degree of data redundancy (as in systems like Ivy or CFS [7]), but also to support the volatility of peers with "specific" responsibilities (e.g., cluster managers, or data managers).

We plan to use JUXMEM as a platform to experiment with various data consistency models and/or protocols supporting peer volatility. We also plan to enable the platform to use high-performance networks (such as Myrinet or SCI) when available for optimized data transfer. The long-term goal is to integrate this service into a large-scale computing environment, such as DIET [10], developed within the ReMaP [31] Research Group. This will allow an extensive evaluation of the service, with realistic codes, using various data access schemes.

These issues are currently subject to research within the GDS [12] (*Grid Data Service*, `http://www.irisa.fr/GDS/`) Project, which gathers together the PARIS [25], ReMaP and REGAL [30] Research Groups of INRIA.

Acknowledgments

The GDS Project is a joint action of the French ACI MD (*Action Concerté Incitative Masses de Donnés*) supported by the French Ministry of Research, INRIA and CNRS. The authors thank the Britanny Region and the French ACIMD GDS [12] project for their support.

References

[1] Bill Allcock, Joseph Bester, John Bresnahan, Ann Chervenak, Lee Liming, Samuel Meder, and Steven Tuecke. *GridFTP Protocol Specification*. GGF GridFTP Working Group Document, September 2002. `http://www.globus.org/research/papers/GridftpSpec02.doc`.

[2] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[3] Alessandro Bassi, Micah Beck, Graham Fagg, Terry Moore, James Plank, Martin Swany, and Rich Wolski. The Internet Backplane Protocol: A study in resource sharing. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2002)*, pages 194–201, Berlin, Germany, May 2002. IEEE.

[4] Brian N. Bershad, Mattew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the 38th IEEE International Computer Conference (COMPCON Spring'93)*, pages 528–537, Los Alamitos, CA, February 1993.

[5] Eddy Caron, Frédéric Desprez, Frédéric Lombard, Jean-Marc Nicod, Martin Quinson, and Frédéric Suter. A scalable approach to network enabled servers. In B. Monien and R. Feldmann, editors, *8th International Euro-Par Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 907–910, Paderborn, Germany, August 2002. Springer-Verlag.

[6] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 152–164, Pacific Grove, CA, October 1991.

[7] Frank Dabek, Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 202–215, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.

[8] Frank Dabek, Ben Zhao, Peter Druschel, and Ion Stoica. Towards a common API for structured peer-to-peer overlays. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*,

volume 2735 of *Lect. Notes in Comp. Science*, pages 98–107, Berkeley, CA, February 2003. Springer-Verlag.

[9] Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003)*, pages 76–87, Providence, Rhode Island, USA, May 2003.

[10] The DIET project: Distributed interactive engineering toolbox. `http://graal.ens-lyon.fr/~diet/`.

[11] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Supercomputer Applications*, 15(3):200–222, March 2001.

[12] The GDS project: Grid data service. `http://www.irisa.fr/GDS/`.

[13] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th International Symposium Computer Architecture (ISCA 1990)*, pages 15–26, Seattle, WA, June 1990.

[14] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proceedings of the 8th ACM Annual Symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 277–287, Padova, Italy, June 1996.

[15] The JXTA project. `http://www.jxta.org/`.

[16] JXTA v2.0 protocol specification. `http://spec.jxta.org/nonav/v1.0/docbook/JXTAProtocols.pdf`, March 2003.

[17] Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal hash table. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, volume 2735 of *Lect. Notes in Comp. Science*, pages 33–44, Berkeley, CA, February 2003. Springer-Verlag.

[18] KaZaA. `http://www.kazaa.com/`.

[19] Anne-Marie Kermarrec, Gilbert Cabillic, Alain Gefflaut, Christine Morin, and Isabelle Puaut. A recoverable distributed shared memory integrating coherence and recoverability. In *The 25th International Symposium on Fault-Tolerant Computing Systems (FTCS-25)*, pages 289–298, Pasadena, California, June 1995.

[20] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[21] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-peer computing. Technical Report HPL-2002-57, HP Labs, March 2002. `http://www.hpl.hp.com/techreports/2002/HPL-2002-57.pdf`.

[22] Napster protocol specification. `http://opennap.sourceforge.net/napster.txt`, March 2001.

[23] Andy Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly, 2001.

[24] Andy Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, chapter Gnutella, pages 94–122. O'Reilly, May 2001.

[25] The PARIS research group: Programming distributed parallel systems for large scale numerical. `http://www.inria.fr/recherche/equipes/paris.en.html`.

[26] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '97)*, pages 311–320, Newport, Rhode Island, June 1997.

[27] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: concepts and systems. *IEEE Paralel and Distributed Technology*, pages 63–79, 1996.

[28] Jelica Protić, Milo Tomasević, and Veljko Milutinović. *Distributed Shared Memory: Concepts and Systems*. IEEE, August 1997.

[29] Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. Routing algorithms for DHTs: Some open questions. In *1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, number 2429 in Lecture Notes in Computer Science, pages 45–52, Cambridge, MA, March 2002. Springer-Verlag.

[30] The REGAL research group: Resources management in large scale distributed systems. `http://www.inria.fr/recherche/equipes/regal.en.html`.

[31] The ReMaP research group: Regularity and massive parallel computing. `http://www.inria.fr/recherche/equipes/remap.en.html`.

[32] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the oceanstore prototype. In *2nd USENIX Conference on File and Storage Technologies (FAST '03)*, California, CA, USA, March 2003.

[33] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *18th IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2001)*, pages 329–350, Heidelberg, Germany, November 2001.

[34] Antony I. T. Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 188–201, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.

[35] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM 2001 conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM 2001)*, pages 149–160, San Diego, CA, August 2001.

[36] Florin Sultan, Thu Nguyen, and Liviu Iftode. Scalable fault-tolerant distributed shared memory. In *The IEEE/ACM SuperComputing conference (SC'00)*, pages 54–55, Dallas, Texas, November 2000.

[37] Ben Yanbin Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001. `http://citeseer.nj.nec.com/article/zhao01tapestry.html`.

[38] Yuanyuan Zhou, Liviu Iftode, and Kai Li. Performance evaluation of two home-based lazy release consistency protocols for shared memory virtual memory systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 75–88, Seattle, WA, October 1996.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifi que,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399