JuxMem: An Adaptive Supportive Platform for Data Sharing on the Grid

Gabriel Antoniu, Luc Bougé, Mathieu Jan
IRISA/INRIA and ENS Cachan/Bretagne
Campus de Beaulieu, 35042 Rennes, France
Contact: Gabriel.Antoniu@irisa.fr

Abstract

We address the challenge of managing large amounts of numerical data within computing grids consisting of a federation of clusters. We claim that storing, accessing, updating and sharing such data should be considered by applications as an external service. *We propose a* hierarchical architecture *for this service, based on a* peer-to-peer *approach. This architecture is illustrated through a software platform called* JuxMem *(for Juxtaposed Memory), which provides transparent access to mutable data, while enhancing data persistence in a dynamic environment. Managing the* volatility of storage resources *is specially emphasized. As a proof of concept, we describe a prototype implementation on top of the JXTA peer-to-peer framework, and we report on a preliminary experimental evaluation.*

1. Introduction

A major contribution of the grid computing environments developed so far is to have decoupled *computation* from *deployment*. Deployment is then considered as an *external service* provided by the underlying infrastructure, outside the application. This service is in charge of locating and interacting with the physical resources, in order to efficiently schedule and map the computation. In contrast, as of today, no such sophisticated service exists regarding *data management* on the grid. Paradoxically enough, complex infrastructures are available for transparent computation scheduling on distributed sites, whereas the user is still left to explicitly store and transfer the data needed by the computation between these sites. At best, advanced FTP-like functionalities are proposed by existing environments. Within the context of a growing number of applications using large amounts of data, this *explicit data management* arises as a major limitation against the efficient use of modern computational grids.

Like deployment, we claim that an adequate approach to this problem consists in decoupling *data management* from *computation*, through an *external service* tailored to the requirements of scientific computation. In this work, we focus on the case of a grid consisting of a federation of distributed clusters. Such a *data sharing service* should meet the following two properties.

Persistence. The data sets used by the grid computing applications may be very large. Their transfer from one site to another may be costly (in terms of both bandwidth and latency), so such data movements should be carefully optimized. Therefore, a data management service should allow data to be stored on the grid infrastructure independently of the applications, in order to allow their reuse in an efficient way. Such a service should also provide data localization information, in order to co-operate with the computation scheduling service, and thereby enhance the global efficiency.

Transparency. Such a data management service should provide transparent access to data. It should handle data localization and transfer without any help from the programmer. Yet, it should make good use of additional information and hints provided by the programmer, if any. The service should also transparently use adequate replication strategies and consistency protocols to ensure data availability and consistency in a large-scale, dynamic architecture. In particular, it should support events such as computational and storage resources joining and leaving, or even unexpectedly failing.

At the same time, three main constraints need to be addressed:

Volatility and dynamicity. The clusters which make up the grid are not guaranteed to remain constantly available. Nodes may leave due to technical problems or because some resources become temporarily unavailable. This should obviously not result in disabling the data management service. Also, new nodes may dynamically join the physical infrastructure: the service should be able to dynamically take into account the additional resources they provide.

Scalability. The algorithms proposed for parallel computing have often been studied on small-scale configurations. Our target architecture is typically made of thousands of computing nodes, say tens of hundred-node clusters. It is well-known that designing low-level, explicit MPI programs is most difficult at such a scale. In contrast, high-level, peer-to-peer approaches have proved to remain effective at much larger scales.

Mutable data. In our target applications, data are generally shared and can be modified by multiple partners. A large number of strategies have been proposed for handling data replication and data consistency, in the context of Distributed Shared Memory (DSM) systems. Again, these strategies and protocols have been designed with the assumption of a small-scale, static, homogeneous architecture, typically of clusters of few tens of nodes. A data sharing service for the grid should consider consistency protocols adapted to a dynamic, large-scale, heterogeneous architecture.

The type of service we propose is similar in some respects to several types of existing data management systems. However, these systems address only partially the goals and the three constraints mentioned above.

Non-transparent, large-scale data management. Currently, the most widely-used approach to data management for distributed grid computation relies on *explicit data transfers* between clients and computing servers. As an example, the Globus [7] platform provides data access mechanisms (Globus Access to Secondary Storage [3]) based on the GridFTP protocol [1]. Though this protocol provides authentication, parallel transfers, checkpoint/restart mechanisms, etc., it is still a FTP-like protocol which requires explicit data localization and transfer. Globus also integrates data catalogs, where multiple copies of the same data can be recorded. The management of these catalogs is manual: it is the user's responsibility to record these copies and make sure they are consistent: no consistency guarantee is provided by Globus.

Large-scale data storage. The IBP Project [2] provides a large-scale data storage system, consisting of a set of buffers distributed over Internet. The user can "rent" these storage areas and use them as temporary buffers for efficient data transfers across a wide-area network. IBP has been used by the Netsolve [18] computing environment to implement a service of persistent data. Transfer management is still at the user's charge. Besides, IBP does not handle dynamic join/departure of storage nodes and provides no consistency guarantee for multiple copies of the same data.

Transparent, small-scale data sharing. Distributed Shared Memory (DSM) systems provide transparent data sharing, via a unique address space accessible to physically distributed machines. Within this context, a variety of consistency models and protocols have been defined, in order to allow an efficient management of replicated data. These systems do offer transparent access to data: all nodes can read and write data in a uniform way, using a unique identifier or a virtual address. It is the responsibility of the DSM system to localize, transfer, replicate data, and guarantee their consistency according to some semantics. Nevertheless, existing DSM systems have generally shown satisfactory efficiency only on small-scale configurations, typically, a few tens of nodes [11].

Peer-to-peer sharing of immutable data. Recently, peer-to-peer (P2P) has proven to be an efficient approach for large-scale data sharing. The peer-to-peer model is complementary to the client-server model: the relations between machines are symmetrical, each node can be client in a transaction and server in another. This paradigm has been made popular by Napster [17], Gnutella [10], and now KaZaA [16]. We can note that these systems focus on sharing immutable files: the shared data are read-only and can be replicated at ease.

Peer-to-peer sharing of mutable data. Recently, some mechanisms for sharing mutable data in a peer-to-peer environment have been proposed by systems like OceanStore [8], Ivy [9] and P-Grid [6]. In OceanStore, for each 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 proven to be efficient at large scales. Second, despite it provides hooks for managing the consistency of data, applications still have to use low-level mechanisms for each consistency model [12]. Third, published measurements on the performance of updates only assume a single writer per data block. Finally, servers making up inner rings are assumed to be highly available. The Ivy system has one main limitation: applications have to repair conflicting writes, thus the number of writers per data is very limited. Both Oceanstore and Ivy target general-purpose, persistent file storage, not data management for high-performance, computing grids where for example distributed matrices have to be moved using parallel transfers. P-Grid proposes a flooding-based algorithm for updating data, but assumes no conflicting writes. Besides, no experimental results have been published so far for this system.

2. Designing a data sharing service for the grid

2.1. Motivating scenarios

Let us consider a distributed federation of 3 clusters: A_1, A_2 and A_3, which co-operate together as shown on Figure 1. Each cluster is typically interconnected through a high-performance local-area network, whereas they are all coupled 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 finally 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. Alternatively, cluster $A1$ may simulate the weather forecast in a given country, while A_2 et A_3 simulate it for two neighboring countries.

Such simulations produce large amount of numerical data, and data-related actions are deeply intricated with computation. The data management systems described in the previous section do not provide any simple technique to support such designs. Consider for instance transferring data from A_1 to A_2: a widely-used technique consists in *explicitly* writing the data on a disk within cluster A_1, then use a file transfer tool to deposit them on a disk within cluster A_2. The application is directly involved in this series of actions. In contrast, we propose to decouple the application from the data management, by making data storage and localization transparent with respect to the application. Cluster A_1 should only store the data within the federation-wide data management service, from which cluster A_2 could request them as needed. Data localization and transfer are then completely external to the applications.

Let us now suppose that our 3 applications no longer cooperate according to a pipeline scheme, but rather according to a *multiple-writers* scheme. For instance, each application simulates a single phenomenon part of the global weather forecast: say, wind, rain and clouds. In this case, each cluster needs data from the other ones in order to make progress. A data sharing service could allow the concurrent applications not only to read, but also to *write* to the globally shared data, while transparently handling data consistency. This is similar to DSM systems, but at a much larger scale, and in a fully dynamic context. Also, assume that some nodes fail in cluster A_2. Some of the data necessary for A_3 could thus become unavailable. The data sharing service should also provide mechanisms to tolerate such faults, for instance, based on redundancy.

2.2. Design principles

We consider two major sources of inspiration for the design of a data sharing service for scientific grid computing:

DSM systems, which propose consistency models and protocols for efficient transparent management of *mutable data, on static, small-scaled configurations (tens of nodes)*;

P2P systems, which have proven adequate for the management of *immutable data* on *highly dynamic, large-scale configurations (millions of nodes).*

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 first and third columns of Table 1.

Our data sharing service targets physical architectures with features intermediate between DSM and P2P systems. We address scales of the order of thousands of nodes, organized as a federation of clusters, say tens 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, we target numerical applications like heavy simulations, made 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 second column of Table 1.

The contribution of this paper is namely to propose an architecture for such a data sharing service, which addresses the problem of managing *mutable data* on *dynamic, large-scale configurations*. Our approach aims at taking benefit of both DSM systems (transparent access to data, consistency protocols) and P2P systems (scalability, support for resource volatility and dynamicity).

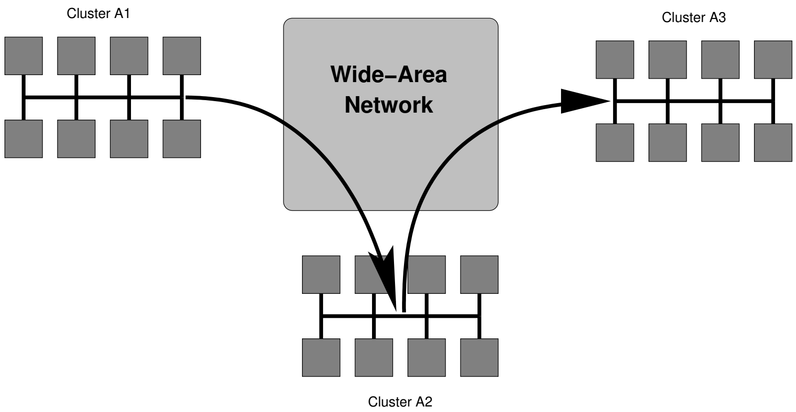

Figure 1. Numerical simulation for weather forecast using a pipeline communication scheme with 3 clusters.

	DSM	**Grid data service**	**P2P**
Scale	10^1–10^2	10^3	10^5–10^6
Resource control and trust degree	High	Medium	Null
Dynamicity	Null	Medium	High
Resource homogeneity	Homogeneous (clusters)	Rather heterogeneous (clusters 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

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

2.3. The JXTA implementation framework

Our proposal is partly inspired by the P2P approach. It can 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 [14]. 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.

JXTA is an open-source framework, which specifies a set of language- and platform-independent XML-based protocols [15]. 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.

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. They can also store advertisements of resources they discover in the network. Rendezvous peers have the extra ability of forwarding the requests they receive to other rendezvous peers. They can also offer a storage area for advertisements that have been published by edge peers. Finally, they are internally managed by JXTA using a *distributed hash table* (DHT) and are making up the frame of JXTA. They can thus be dynamically located in an efficient way. Joining, leaving, and even unexpected failing of 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.

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. 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.

The data sharing service that we propose is designed using the JXTA building blocks described above.

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

The architecture of the data sharing service mirrors an 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*), whose goal is to be the foundation for a data sharing service for grid computing environments, like DIET [4].

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 noted that a node can be at the same time cluster manager, client and provider, but for the sake of clarity, each node plays only one role in the example illustrated on the figure.

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 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 also 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, for each block of data inserted into the system, a `data` group is automatically instantiated.

API of 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.

`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 parameters for the view of the data block desired by the client, like for instance what we call the degree of consistency: some clients may have weaker consistency requirements than the one ensured by the default protocol used to manage the data block.

`put(id, value)` allows to modify the value of the data block identified by `id`. The new value is then `value`.

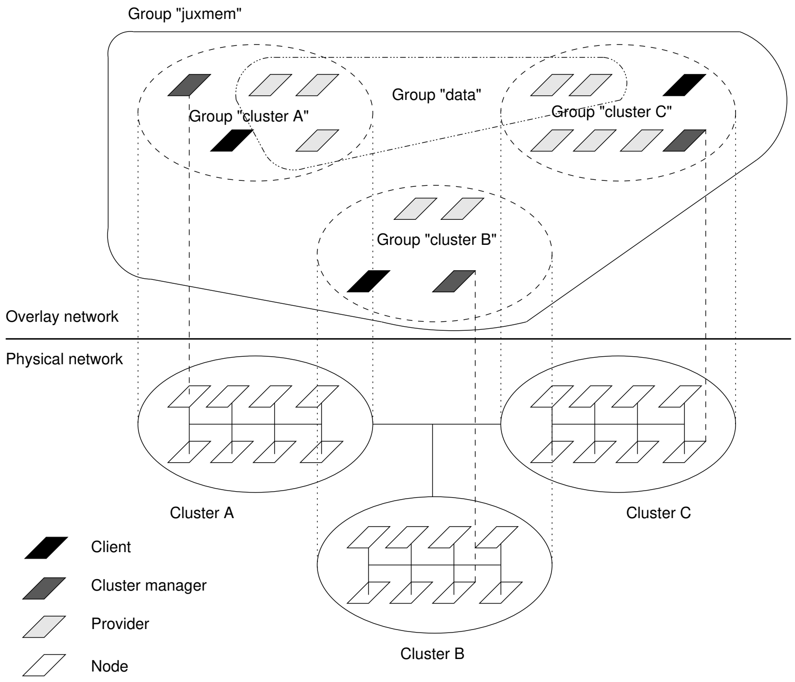

Figure 2. Hierarchy of the entities in the network overlay defined by JUXMEM**.**

`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 which access a shared data block need to synchronize using this lock.

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

`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, the `attributes` parameter also allows to specify the amount of memory that the node provides to JUXMEM.

3.2. Managing memory resources

Publishing and placement 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. He 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 providers of the associated `cluster` group. These cluster advertisements are published inside the `juxmem` group, so that they can then be used by all the clients in order to allocate memory.

Cluster managers are thus in charge of making the link between the `cluster` group and the `juxmem` group. They make up a network organized using a DHT at the level of the `juxmem` group level, in order to build the frame of the data sharing service. This frame is represented by the ring on Figure 3. Each cluster manager G1 to G6 is responsible for a cluster, respectively A1 to A6, each of which is made up of five 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. This cluster manager is not the peer that stores the advertisement, it only knows the cluster manager which published it in the JUXMEM network. This placement 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 mechanisms described above

Searching for advertisements is therefore short, 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. Thus, JUXMEM will search for the minimum memory size, given by the policy used, that is superior to the one requested by clients. For example, if a client wants to allocate a memory area of 1280 bytes, JUXMEM will internally and automatically search for a memory area of 2048 bytes, if it uses a power of 2 law for the space discretization. Providers also internally use the same law when offering memory areas, but provide the maximum memory size, given by the policy used, that is inferior to the one they wish to offer.

One of the constraints we fixed is to support the volatility of nodes which make up the clusters. Therefore, the advertisements published at a time $t1$ can be invalid at the time $t2 > t1$, since providers can disappear from JUXMEM at any time. The mechanism used to manage this volatility of peers is based on republishing the cluster advertisements whenever a changing of the amount of memory provided is detected. Besides, advertisements have a limited but parameterizable lifetime, so it is necessary to periodically republish them.

Processing an allocation request. Clients make allocation requests by specifying the size of the memory area they want to allocate. The different steps for such a request, numbered on the Figure 3, are the following:

1. The client C of the `cluster` group $A1$ wants to allocate a memory area of 8 MB with a redundancy degree of two. Consequently, it submits its request to the cluster manager $G1$ to which it is connected.

2. The cluster manager $G1$ then determines that the peer responsible of advertisements having a memory size of 8 MB in their list is the cluster manager $G3$, using the hash mechanism described previously. Therefore, the cluster manager peer $G1$ forwards the request to $G3$.

3. The cluster manager $G3$ then determines that cluster managers $G2$ and $G4$ match the criterion of the client, and asks them to forward their `cluster` advertisement to the client C.

4. The client C then chooses the cluster manager $G2$ as the peer having the "best" advertisement: for instance the corresponding cluster offers a higher degree of redundancy than the cluster handled by the cluster manager $G4$. Thus, it submits its allocation request to $G2$.

5. The cluster manager $G2$ receives the allocation request and handles it. If it can satisfy the request then it asks one of its providers, for example P, to allocate a 8 MB memory area. If the request cannot be satisfied, an error message is sent back to the client.

6. If the provider P can satisfy this request, it creates a 10 MB memory area, then sends back the advertisement of this memory area to the client C. P becomes the cluster manager of the associated `data` group, which means that it is responsible for replicating the data block stored in that memory area. If the provider P cannot satisfy the request, an error message is sent back to the cluster manager $G2$, which can try other provider peers of the `cluster` group.

If no providers can be found on the last step of an allocation request, an error message is sent back to the client. Then the client can 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 memory area, the client increases the requested memory size and restarts the allocation request from the beginning. This can be done N times (for example $N = 3$) 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 on the chosen provider and an advertisement is sent to the client. This advertisement allows the client to communicate with the `data` group. This advertisement is published at the `juxmem`'s 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: the platform *transparently* locates the corresponding data block.

Storage of data blocks is independent of clients. Indeed, when clients disconnect from JUXMEM, data blocks still remain stored in the data sharing service on the providers. Consequently, clients can have access to data blocks previously stored by other clients: they simply need to look for the advertisement of the `data` group associated with the data block (whose identifier is assumed to be known). The `map` primitive of the API of JUXMEM does this by taking in input the ID of the data block. In this way, the storage of data blocks is persistent.

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.

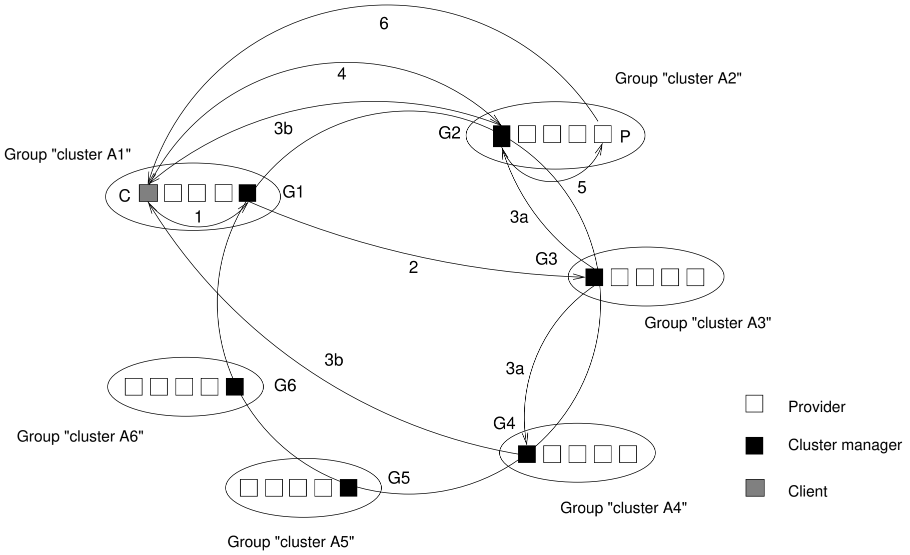

Figure 3. Steps of an allocation request made by a client.

The consistency of the different copies must then be handled. In this first version of JUXMEM, the use of a multicast at the level of the `juxmem` group solves this problem: the different 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. Note that clients which have previously read a data block are not notified of this update: clients do not store a copy of data block. Therefore, the result of a reading which is valid at a time $t1$, may not be valid at time $t2 > t1$. It is worth noting that this difference between client and 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. Handling volatile providers

In order to tolerate the volatility of peers, a static replication of data on a fixed and parameterizable number of providers is not enough. Indeed, the set of providers hosting a copy of the same data block can successively become unavailable. A dynamic monitoring of the number of copies for data is therefore needed. Consequently, each `data` group has a manager (noted *data manager*) 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 must search and ask a provider to host an extra copy of the data block. When the data manager decides to replicate it, it must first lock it (internally) in order to maintain consistency. The provider which will host this new copy is then responsible for unlocking it. A *timeout* mechanism followed by a *ping* test is used in order to detect if the provider became unavailable just before unlocking the data block. If it is the case, then the data manager unlocks itself the data block.

3.5. Handling volatile managers

If a cluster manager goes down, this could lead to the unavailability of resources provided by a whole cluster. The role of cluster manager (noted *main cluster manager*) is therefore automatically duplicated on another provider of the cluster (called *secondary cluster manager*). Managers periodically synchronize using a mechanism based on the exchange of provider advertisements, in order to find out new advertisements published. They can thus both know in a nearly accurate manner the amount of memory available in the cluster. A mechanism based on periodical heartbeats allows to dynamically ensure this duplication of cluster managers. Such a mechanism is also used for the data managers (see Section 3.4). Note that, the possible changes of managers in the `cluster` and `data` groups, due to the unavailability of managers, are not seen 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

In order to build a prototype of the software architecture described in the previous section, we have used the JXTA generic peer-to-peer framework (see Section 2.3). Our JuxMem prototype uses the reference Java binding of JXTA (which is today the only binding compatible with the JXTA 2.0 specification). JuxMem is written in Java and includes about 50 classes (5000 code lines).

JXTA fully meets the needs of JuxMem. Thus, managers of `data` and `cluster` groups are based on JXTA's *rendezvous peers*. Indeed, managers have to know if providers are still alive by using a ping test in order to manage a cluster or a block of data. This can only be done if providers have previously published their advertisements on managers, which need to extract the address of each provider. Moreover, only JXTA's rendezvous peers can forward requests inside the JXTA network; these peers correspond to the role of *main managers*. For example, data managers have to forward access requests, made by clients, to providers hosting a copy of the data block. In the same way, cluster managers have to foward allocation requests, made by clients, to providers. Clients and providers which do not act as data managers for one or several blocks of data are based on JXTA's *edge peers*. Indeed, they do not have to play a role in the dynamic monitoring of the number of copies for a block of data in the system. Therefore, they do not have to store published provider advertisements. Moreover, clients only need to discover and store cluster advertisements which will allow them to allocate memory areas. The various groups defined in JuxMem are implemented by JXTA's peer groups. The `juxmem` group implements a JXTA peer group service providing the API of JuxMem (see Section 3.1). Finally, the communication channels of JXTA also offer the needed support for building multicast communications for simultaneously updating copies of the same block of data.

4.2. Preliminary evaluations

For our preliminary experiments, we used a cluster of 450 MHz Pentium II nodes with 256 MB RAM, interconnected by a 100 MB/s FastEthernet network.

We first measured the memory consumption overhead generated by the different JuxMem peers with respect to the underlying JXTA peers used to build JuxMem peers. This overhead is reasonable: it ranges between 5% and 7.4%.

We then measured the influence of the volatility degree of provider peers on the duration of a sequence `lock-put-unlock` executed in a loop by a client. This sequence in the loop is made on a 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 for a given block of data. This replications are transparently triggered when the service detects that a provider holding a data block goes down. If these replications take place while a client accesses the data block being replicated, these accesses slow down.

The test program first allocates a small memory area (1 byte) on a provider belonging to cluster and writes to it 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. 16 machines of the cluster previously described host a provider, one machine of the same cluster hosts a cluster manager and another machine of the same cluster hosts a client. The client executes a 100 iteration loop, and each iteration consists of a sequence `lock-put-unlock`.

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 measure only the overhead due to the volatility of providers, the data manager of the associated group is never killed.

Figure 4 shows the relative overhead measured, with respect to a stable system (i.e. where no provider goes down during the loop execution: $\delta = \infty$). When the data manager detects that providers holding a copy of the data block have gone down, it tries to replicate the block on other providers available, which are not already hosting a copy of the data block. During the replication, the system has to internally lock the data, since clients must not modify the data during its replication. As a result of this internal locking, the sequence `lock-put-unlock` is longer, since the client is blocked and has to wait for the lock to be set free.

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 (which is 3 for this test). For the whole duration of our test, the number of replications is given in the Table 4.2 as a function of the δ parameter.

For highly volatile systems ($\delta < 80$ s), the number of replications triggered becomes higher than 2 and the relative overhead becomes significant. For $\delta = 30$ 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 ($\delta >> 80$ s). For such

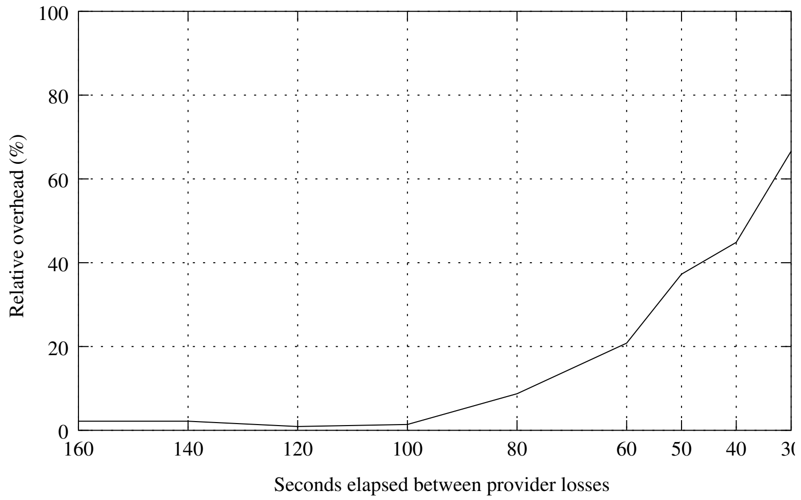

Figure 4. Relative overhead due to the volatility of providers for a sequence `lock-put-unlock`**, with respect to a stable system.**

values, the reconfiguration overhead is less than 5%. We can reasonably say 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.

5. Conclusion

This paper defines a *hierarchical* architecture for a data sharing service managing mutable data within a grid consisting of a federation of clusters. This architecture has been designed using a peer-to-peer approach, and demonstrated through the JUXMEM platform. 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 network links to be used. Thus, some clusters could use high-bandwidth, low-latency networks for intra-cluster communication, if available.

The JUXMEM user can allocate memory areas in the system, by specifying an area size and some attributes, such as a redundancy degree. The allocation primitive returns an ID which identifies the block of data. Then, data localization and transfer is fully transparent, since this ID is sufficient in order 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 a side effect of the user actions. In contrast, our system *actively* takes into account this volatility: this allows not only to maintain a certain degree of data redundancy (as in systems like Ivy or CFS [5]), but also to support the volatility of peers with "specific" responsibilities (e.g., cluster managers, or data managers).

The implementation of a JXTA-based prototype has shown the feasibility of such a system. However, note that the design of JUXMEM is not dependent on JXTA. Actually, other libraries could be used, such as JavaGroups [13]. We used the Java version of JXTA, since this is the most advanced binding of JXTA, the only one compatible with the JXTA 2.0 specification.

The modular architecture of JXTA allows to easily add and remove services and/or protocols, including communication protocols. This should eventually allow the platform to take advantage of high-performance networks (such as Myrinet or SCI) for data transfer. We plan to address this problem in the future. We also plan to use JUXMEM as an experimental platform for different data consistency strategies supporting peer volatility, in order to build a configurable, adaptive data sharing service for mutable data. The final goal is to integrate this service into large-scale computing environments, such as DIET [4], developed at ENS Lyon. This will allow an extensive evaluation of the service, with realistic codes, using various data access schemes.

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

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

References

[1] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, and S. Tuecke. *GridFTP Protocol Specification*. GGF GridFTP Working Group Document, Sept. 2002.

[2] A. Bassi, M. Beck, G. Fagg, T. Moore, J. Plank, M. Swany, and R. 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.

[3] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A data movement and access service for wide area computing systems. In *6th Workshop on I/O in Parallel and Distributed Systems (IOPADS '99)*, pages 77–88, Atlanta, GA, May 1999. ACM Press.

[4] E. Caron, F. Desprez, F. Lombard, J.-M. Nicod, M. Quinson, and F. 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, Aug. 2002. Springer-Verlag.

[5] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. 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, Oct. 2001.

[6] A. Datta, M. Hauswirth, and K. 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.

[7] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[8] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *9th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS 2000)*, number 2218 in Lecture Notes in Computer Science, pages 190–201, Cambridge, MA, Nov. 2000. Springer.

[9] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, Dec. 2002.

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

[11] J. Protić, M. Tomasević, and V. Milutinović. *Distributed Shared Memory: Concepts and Systems*. IEEE, Aug. 1997.

[12] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the oceanstore prototype. In *2nd USENIX Conference on File and Storage Technologies (FAST '03)*, Californie, CA, USA, Mar. 2003.

[13] JavaGroups. `http://www.javagroups.com/javagroupsnew/docs/index.html`.

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

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

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

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

[18] The NetSolve project. `http://icl.cs.utk.edu/netsolve/`.