GDS: An Architecture Proposal for a grid Data-Sharing Service

Gabriel Antoniu, Luc Bougé, Mathieu Jan, Sébastien Monnet, Marin Bertier, Eddy Caron, Frédéric Desprez, Pierre Sens

▶ To cite this version:

Gabriel Antoniu, Luc Bougé, Mathieu Jan, Sébastien Monnet, Marin Bertier, et al.. GDS: An Architecture Proposal for a grid Data-Sharing Service. Workshop on Future Generation Grids, Nov 2004, Dagstuhl, Germany. pp.133-152, 10.1007/978-0-387-29445-2_8 . inria-00000983

HAL Id: inria-00000983

https://inria.hal.science/inria-00000983v1

Submitted on 9 Jan 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.

GDS: AN ARCHITECTURE PROPOSAL FOR A GRID DATA-SHARING SERVICE

Gabriel Antoniu, Luc Bougé, Mathieu Jan and Sébastien Monnet
IRISA/INRIA - University of Rennes 1 - ENS Cachan-Bretagne
Campus de Beaulieu, F-35042 Rennes cedex, France
Gabriel.Antoniu@irisa.fr

Marin Bertier
LRI/INRIA Grand Large - University Paris Sud
Université de Paris Sud, F-91405 Orsay cedex, France
Marin.Bertier@lri.fr

Eddy Caron and Frédéric Desprez
LIP/INRIA - ENS Lyon
46 Allée d'Italie, F-69364 Lyon Cedex 07, France
Eddy.Caron@ens-lyon.fr

Pierre Sens
INRIA/LIP6-MSI
8, rue du Capitaine Scott, F-75015 Paris, France
Pierre.Sens@lip6.fr

Abstract Grid computing has recently emerged as a response to the growing demand for resources (processing power, storage, etc.) exhibited by scientific applications. We address the challenge of sharing large amounts of data on such infrastructures, typically consisting of a federation of node clusters. We claim that storing, accessing, updating and sharing such data should be considered by applications as an *external service*. We propose an architecture for such a service, whose goal is to provide *transparent access* to *mutable* data, while enhancing data persistence and consistency despite node disconnections or failures. Our approach leverages on weaving together previous results in the areas of distributed shared memory systems, peer-to-peer systems, and fault-tolerant systems.

Keywords: Data sharing, grid computing, transparent access, mutable data, peer-to-peer systems, fault tolerance, consistency protocols

1. Introduction

Data management in grid environments. Data management in grid environments is currently a topic of major interest to the grid computing community. However, as of today, no approach has been widely established for *transparent data sharing* on grid infrastructures. Currently, the most widely-used approach to data management for distributed grid computation relies on *explicit data transfers* between clients and computing servers: the client has to specify where the input data is located and to which server it has to be transferred. Then, at the end of the computation, the results are eventually transferred back to the client. As an example, the Globus [16] platform provides data access mechanisms based on the GridFTP protocol [1]. Though this protocol provides authentication, parallel transfers, checkpoint/restart mechanisms, etc., it still requires *explicit* data localization.

It has been shown that providing data with some degree of persistence may considerably improve the performance of series of successive computations. Therefore, Globus has proposed to provide so-called *data catalogs* [1] on top of GridFTP, which allow multiple copies of the same data to be manually recorded on various sites. However, the consistency of these replicas remains the burden of the user.

In another direction, a large-scale data storage system is provided by IBP [5], as a set of so-called *buffers* distributed over Internet. The user can "rent" these storage areas and use them as temporary buffers for optimizing data transfers across a wide-area network. Transfer management still remains the burden of the user, and no consistency mechanism is provided for managing multiple copies of the same data. Finally, Stork [18] is another recent example of system providing mechanisms to *explicitly* locate, move and replicate data according to the needs of a sequence of computations. It provides the user with an integrated interface to schedule data movement actions just like computational jobs. Again, data location and transfer have to be explicitly handled by the user.

Our approach: transparent access to data. A growing number of applications make use of larger and larger amounts of distributed data. We claim that *explicit management of data locations* by the programmer arises as a major limitation with respect to the efficient use of modern, large-scale computational grids. Such a low-level approach makes grid programming extremely hard to manage. In contrast, the concept of a *data-sharing service* for grid computing [2] opens an alternative approach to the problem of grid data management. Its ultimate goal is to provide the user with *transparent access to data*. It has been illustrated by the experimental JUXMEM [2] software platform: the user only accesses data via *global handles*. The service takes care of data local-

ization and transfer without any help from the external user. The service also transparently applies adequate replication strategies and consistency protocols to ensure data persistence and consistency in spite of node failures. These mechanisms target a large-scale, dynamic grid architecture, where nodes may unexpectedly fail and recover.

Required properties. The target applications under consideration are scientific simulations, typically involving multiple weakly-coupled codes running on different sites, and cooperating via periodic data exchanges. Transparent access to remote data through an external data-sharing service arises as a major feature in this context. Such a service should provide the following properties.

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 provide persistent data storage, so as to save data transfers. It should rely on strategies able to: 1) reuse previously produced data, by avoiding repeated data transfers between the different components of the grid; 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.

Fault tolerance. The data-sharing service must match the dynamic character of the grid infrastructure. In particular, the service has to support events such as storage resources joining and leaving, or unexpectedly failing. Replication techniques and failure detection mechanisms are thus necessary. Based on such mechanisms, sophisticated fault-tolerant distributed data-management algorithms can be designed, in order to enhance data availability despite disconnections and failures.

Data consistency. In the general case, shared data manipulated by grid applications are *mutable*: they can be read, but also *updated* by the various nodes. When accessed on multiple sites, data are often replicated to enhance access locality. To ensure the consistency of the different replicas, the service relies on *consistency models*, implemented by *consistency protocols*. However, previous work on this topic (e.g., in the context of Distributed Shared Memory systems, DSM) generally assumes a small-scaled, stable physical architecture, without failures. It is clear that such assumptions are not relevant with respect to our context. Therefore, building data-sharing service for the grid requires a new approach to the design of consistency protocols.

In this paper, we address these issues by proposing an architecture for a data-sharing service providing grid applications with *transparent access to data*. We

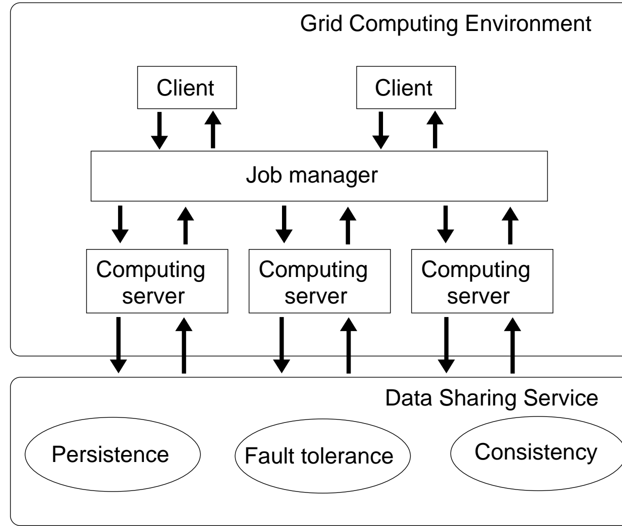

Figure 1. Overview of a data-sharing service.

consider the general case of a distributed environment in which *Clients* submit jobs to a *Job Manager*, an entity in charge of selecting the *Computing Servers* where job execution shall take place. When the same data are shared by jobs scheduled on different servers, the *Data-Sharing Service* can be used to store and retrieve them in a transparent way. This general organization scheme is illustrated on Figure 1.

The paper is organized as follows. Section 2 first describes a few motivating scenarios that illustrate the three required properties mentioned above. Section 3 presents an overview of a particular grid computing environment called DIET, whose architecture implements the generic organization scheme illustrated on Figure 1. More specifically, we discuss the needs of such an environment with respect to data management. In Section 4, the JUXMEM software data management platform is introduced and we show how it can be used as a basis to fulfill these needs. Several aspects related to fault tolerance and consistency are discussed in detail. Section 5 provides an overview of the global architecture. Finally, Section 6 concludes and discusses future directions.

2. Application scenarios

Our approach can be best motivated by a grid application managing large data sets and needing data persistence. One such project is called Grid-TLSE [11] and is supported by the French ACI GRID Research Program. It aims at designing a Web portal exposing the best-level expertise about sparse

matrix manipulation. Through this portal, the user may gather actual statistics from runs of various sophisticated sparse matrix algorithms on his/her specific data. The Web portal provides an easy access to a variety of sparse solvers, and it assists the comparative analysis of their behavior. The input data are either problems submitted by the user, or representative examples picked up from the matrix collection available on the site. These solvers are executed on a grid platform. Since many sparse matrices of interest are very large, avoiding useless data movement is of uttermost importance. A sophisticated data management strategy is thus needed.

The process for solving a sparse symmetric positive definite linear system, $Ax = b$, can be divided into four stages as follows: ordering, symbolic factorization, numerical factorization and triangular system solution. We focus on ordering in this scenario. The aim of ordering is to find a suitable permutation P of matrix A. Because the choice of permutation P will directly determine the number of fill-in elements, the ordering has a significant impact on the memory and computational requirements for the latter stages.

Let us consider a typical scenario to illustrate the need for threefold requirement for data persistence, fault tolerance and consistency. It is concerned with the determination of the *ordering sensitivity* of a class of solvers such as MUMPS, SuperLU or UMFPACK, that is, how performance is impacted by the matrix traversal order. It consists of three phases. Phase 1 exercises all possible internal orderings in turn. Phase 2 computes a suitable metric reflecting the performance parameters under study for each run: effective FLOPS, effective memory usage, overall computation time, etc. Phase 3 collects the metric for all combinations of solvers/orderings and reports the final ranking to the user.

If Phase 1 requires exercising n different kinds of orders with m different kinds of solvers, then $m \times n$ executions are to be performed. Without persistence, the matrix has to be sent $m \times n$ times. If the server provided persistent storage, the data would be sent only once. If the various pairs solvers/orderings are handled by different servers in Phase 2 and 3, then consistency and data movements between servers should be provided by the data management service. Finally, as the number of solvers/orderings is potentially large, many nodes are used. This increases the probability for faults to occur, which makes the use of sophisticated fault-tolerance algorithms mandatory.

Another class of applications that can benefit from the features provided by a data-sharing service is *code-coupling applications*. Such applications are structured as a set of (generally distributed) autonomous codes which at times need to exchange data. This scheme is illustrated by the EPSN [10] project, also supported by the French ACI GRID Research Program. This project focuses on steering distributed numerical simulations based on visualization. It relies on a software environment that combines the facilities of virtual reality

with the capabilities of existing high performance simulations. The goal is to make the typical work-flow (modeling, computing, analyzing) more efficient, thanks to on-line visualization and interactive steering of the intermediate results. Possible errors can thus be detected and the the researcher can correct them on-the-fly, by tuning the simulation parameters. The application consists in a visualization code coupled with one or more simulation codes. Each simulation code may be parallel and may manipulate data according to some specific distribution. As in the case of the Grid-TLSE application described above, a data-sharing service providing *persistent, transparent* access to distributed data can simplify the data movement schemes between the coupled codes. Moreover, in the case of code coupling, the basic operations consist in extracting and modifying the simulation data. As the data status alternates from consistent to inconsistent during the simulation, it is important for the visualization code to be able to obtain a consistent view of the data. This can be ensured thanks to the *consistency protocols* provided by the data service.

3. Overview of a grid computing environment: the DIET platform

The GridRPC approach [22] is a good candidate to build Problem Solving Environments (PSE) on the computational grid. It defines an API and a model to perform remote computation on servers. In such a paradigm, a client can submit problem to an agent that selects the best server among a large set of candidates, given information about the performance of the platform gathered by an information service. The goal is to find a suitable (if not the best!) trade-off between the computational power of the selected server and the cost of moving input data forth and back to this very server. The choice is made from static and dynamic information about software and hardware resources, as well as the location of input data, which may be stored anywhere within the system because of previous computations. Requests can be then processed by sequential or parallel servers.

The GridRPC API is the grid form of the classical Unix RPC (*Remote Procedure Call*) approach. It has been designed by a team of researchers within the Global Grid Forum (GGF). It defines a standard client API to send requests to a Network Enabled Server (NES) system [23], therefore promoting portability and interoperability between the various NES systems. Requests are sent through synchronous or asynchronous calls. Asynchronous calls allow a non-blocking execution, thereby providing another level of parallelism between servers. A function handle represents a binding between a problem name and an instance of such function available on a given server. Of course several servers can provide the same function (or service) and load-balancing can be done at the agent level before the binding. A session ID is associated to

each non-blocking request and allows to retrieve information about the status of the request. Wait functions are also provided for a client to wait for specific request to complete. This API is instantiated by several middleware such as DIET [7], Ninf [20], NetSolve [4], and XtremWeb [15].

The paradigm used in the GridRPC model is thus two-level, mixed parallelism, with different (potentially parallel) requests executed on different servers. However, the server-level parallelism remains hidden to the client.

3.1 Overall architecture of DIET

In this section, we focus on our GridRPC-based middleware: the DIET platform. The various parts of the DIET architecture are displayed on Figure 2.

The Client is an application which uses DIET to solve problems. Different types of clients should be able to use DIET, as problems can be submitted from a web page, a specific PSE such as Scilab, or directly from a compiled program.

The Master Agent (MA) receives computation requests from clients. A request is a generic description of the problem to be solved. The MA collects the computational capabilities of the available servers, and selects the *best* one according to the given request. Eventually, the reference of the selected server is returned to the client, which can then directly submit its request to this server.

The Local Agent (LA) transmits requests and information between a given MA and the locally available servers. Note that, depending on the underlying network architecture, a hierarchy of LAs may be deployed between a MA and the servers it manages, so that each LA is the root a of subtree made of its son LAs and leaf servers. Each LA stores the list of pending requests, together with the number of servers that can handle a given request in its subtree. Finally, each LA includes information about the data stored within the nodes of its subtree.

The Server Daemon (SeD) encapsulates a computational server. The SeD stores the list of requests that its associated computational server can handle. It makes it available to its parent LA, and provides the potential clients with an interface for submitting their requests. A SeD also stores the list of data (that is in our case, matrices) available on its associated server, together with some meta-information about them: data distribution, access path, etc. Finally, a SeD periodically probes its associated server for its *status*: instantaneous load, free memory, available resources, etc. Based on this status, a SeD can provide its parent LA with accurate performance prediction for a given request. This uses FAST [13], a dynamic performance forecasting tool.

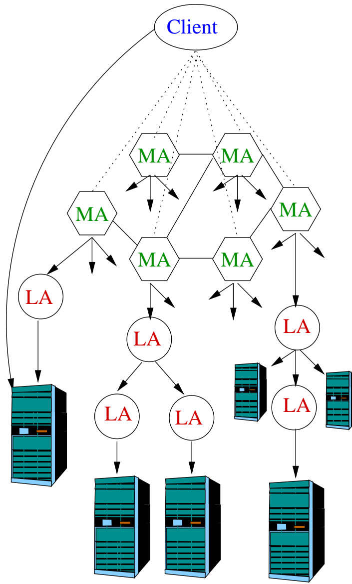

Figure 2. The hierarchical organization of DIET.

When a client wishes to submit a computational request using DIET, it must first obtain a reference to the server that is best suited for handling its request. Either the client can obtain the name of some MA through a dedicated name server, or it can find one by browsing a specific Web page which stores the various MA locations. The client request consists of a generic description of the problem to be solved. The MA first checks the request for correctness, e.g., all the necessary parameters are provided. Then, it broadcasts the request to the neighboring nodes, LAs or MAs, which in turn forward the request to the connected SeDs. Each server sends back its status to its parent LA. Based on these information, each LA selects the best server and forwards its name and status to its parent LA or MA. The root MA aggregates all the best servers found by its LAs or by other MAs. It ranks them by status and availability, and eventually forwards the result to the client. The client goes through the resulting server list, and successively attempts to contact each server in turn. As soon as a server can be reached, the client moves the input data of the request to it. The server executes the request on behalf of the client, and returns the results.

3.2 Managing data in DIET

The first version of the GridRPC API does not include any support for data management, even though discussions on this aspect have been started. Data movement is left to the user, which is clearly a major limitation for efficiently programming grids, as discussed in Section 1. Introducing a *transparent access* to data and some *persistence modes* into this API would remove a significant burden from the programmer. It would also contribute to master communication overheads, as it saves unnecessary movements of computed data between servers and clients.

Transparent access can be achieved using a specific ID for each data. It is the responsibility of the data management infrastructure to localize the data based in this ID, and to perform the necessary data transfers. Thanks to this approach, the clients can avoid dealing with the physical location of the data.

Persistence modes allow the clients to specify that data blocks should be stored on the grid infrastructure, "close" to computational servers, rather than be transferred back to the client at each computation step. Also, the data generated by some computational request can be simply re-used by other servers in later requests through the data ID. Thanks to the transparent access scheme, the clients only have to provide the request server with the ID, not with the physical data.

In order to let GridRPC applications express constraints with respect to data transparency and persistence, discussions on extensions of the GridRPC API are in progress within the GridRPC working group of GGF. These extensions allow GridRPC computing environments to use external data management infrastructures, as illustrated on Figure 3. In this example, a client C successively uses two servers ($S1$ and $S2$) for two different computations. We assume that the second computation depends on the first one: the output data $D2$ produced on server $S1$ is used as input data for the computation scheduled on $S2$. We also assume that $D2$ is intermediate data that is not needed by the client. On the left side, we illustrate a typical scenario using the current GridRPC API, with no support for data management. The client C needs to explicitly transfer the output data $D2$ from server $S1$ to server $S2$ (steps 2 and 3). Then, the second computation on server $S2$ can take place and returns data $D3$ to client C (step 4). On the right side we show how these computations would be handled if the GridRPC infrastructure provided support for localization transparency and persistence. The server $S1$ stores the output data $D2$ in a data management infrastructure (step 2). Then, the client C only needs to transmit the ID of data $D2$ to $S2$ (step 3a). Consequently, the data transfer between $S1$ and $S2$ occurs in a transparent manner for the client (step 3b). We assume that

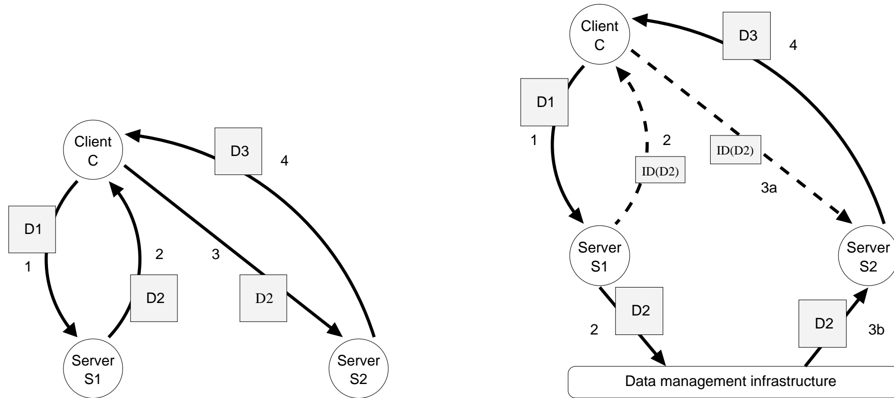

Figure 3. Steps of a client request without (left side) and with (right side) a data management infrastructure.

the servers are connected to the storage service using high-performance links, whereas this may not be true for the links between clients and servers. Using a data management infrastructure clearly avoids unnecessary and costly data transfers between the client and servers. Arrows 2 and 3 (left side) represent these unnecessary data transfers that can be optimized out when using a data management infrastructure.

As a preliminary step, a data management service called Data Tree Manager (DTM) has been specifically developed for the DIET platform [14]. This solution uses DIET's computing servers (SeD) for persistent data storage and needs no external storage resources. However, a simpler and more flexible approach is to fully let data management at the charge of an *external* data-sharing service. As explained in the previous section, the benefits of such a service consist in its mechanisms for transparent access, persistence, fault tolerance and consistency. This approach is at the core of the design of the JUXMEM software platform, as described in the following section.

4. A data management environment: the JUXMEM platform

The goal of this section is to introduce the JUXMEM data-sharing software platform, designed to serve as a basis for a grid data-sharing service. We first present JUXMEM's architecture and then discuss its mechanisms for handling fault tolerance and data consistency.

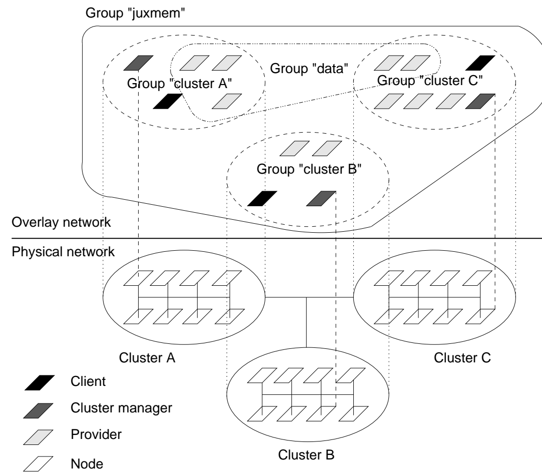

Figure 4. Hierarchy of the entities in the network overlay defined by JUXMEM.

4.1 Overall architecture of JUXMEM

The software architecture of JUXMEM (for *Juxtaposed Memory*), mirrors a hardware architecture consisting of a federation of distributed clusters and is therefore *hierarchical*. Figure 4 shows the hierarchy of the entities defined in JUXMEM's architecture. This architecture is made up of a network of peer groups (`cluster` groups A, B and C on the figure), which usually correspond to clusters at the physical level. However, a `cluster` groups could also correspond to a subset of the same physical cluster, or alternatively to nodes spread over several physical clusters.

All the groups belong to a wider group, which includes all the peers which run the service (the `juxmem` group). Each `cluster` group includes several kinds of nodes. Those which provide memory for data storage are called *providers*. In each `cluster` group, a node is used to make up the backbone of JUXMEM's network of peers. 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 replicated and associated to a group of peers called `data` group. 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 (e.g., A and C on the figure). For this reason, the `data` and `cluster` groups are at the same level of the group hierarchy. 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.

When allocating memory, the client has to specify on how many clusters the data should be replicated, and on how many nodes in each cluster. This results into the instantiation of a set of data replicas. The allocation operation returns a global data ID. This ID can be used by other nodes in order to identify existing data. To obtain read and/or write access to a data block, the clients only need to use this ID. It is JUXMEM's responsibility to localize the data, and then perform the necessary data transfers.

The design of JUXMEM is detailed in [2]. JUXMEM is currently being implemented using the generic JXTA [25] P2P library. In its 2.0 version, JXTA consists of a specification of six language- and platform-independent, XML-based protocols that provide basic services common to most P2P applications, such as peer group organization, resource discovery, and inter-peer communication. To the best of our knowledge, a lot of on-going efforts for integrating grid services with P2P techniques are based on JXTA.

4.2 Fault tolerance issues in JUXMEM

In grid environments, where thousands of nodes are involved, failures and disconnections are no longer exceptions. In contrast, they should be considered as plain, ordinary events. The data blocks handled by JUXMEM should remain available despite such events. This property of *data availability* is achieved in JUXMEM by replicating each piece of data across the `data` groups, as described above. The management of these groups in the presence of failures relies on group communication and group management protocols that have been extensively studied in the field of (most often theoretical!) fault-tolerant distributed systems. This section describes in detail the fault-tolerant building blocks we build on.

4.2.1 Assumptions.

Timing model. We rely on the model of partial synchrony proposed by Chandra and Toueg in [8]. This model stipulates that, for every execution, there exists global bounds on process speeds and on message transmission delays. However, these bounds are not known and they only hold after some unknown time. This assumption seems reasonable within grid context.

Failure model. We assume that only two kinds of failure can occur within grids: node failures and link failures. Node failures are assumed to follow the *fail-silent* model. The nodes act normally (receive and send messages according to their specification) until they fail, and then stop performing any further action for ever. We also consider link failures.

We assume *fair-lossy* communication channels. If Process p repeatedly sends Message m to Process q through a fair-lossy channel, and if Process q does not fail, then Process q eventually receives Message m from Process p. Informally, this means that network links may lose messages, but not all of them. Note that a single node or link failure may induce other failures, so that simultaneous failures of any kind have to be taken into account.

4.2.2 Fault tolerance building blocks.

Group membership protocols. The *group membership* abstraction [9] provides the ability to manage a set of nodes in a distributed manner, and to provide to the external world with an abstraction of a single entity. In particular, it is possible to send a message to this virtual entity, which means that either the message is eventually delivered to all the non-faulty nodes of the group, or to none of them. The nodes belonging to the group have to maintain the current composition of the group in some local member list, called their *view*. As nodes may join or leave the group, and even crash, the composition of a group is continuously changing. The role of the *group membership* protocol is thus to ensure the consistency of the local views with the actual composition of the group. It is achieved by synchronizing the members' views of the group. Between two consecutive view synchronizations, the same set of messages from the external world should be delivered to all the non-faulty nodes within a group. In the case of JUXMEM, a *group membership* protocol is applied to each `data` group gathering nodes which store a copy of a same piece of data.

Atomic multicast. Since the nodes members of a `data` group may crash, we use a *pessimistic replication* mechanism to ensure that an up-to-date copy of the common replicated data remains available. When the data is accessed by the external world (here, the DIET SeDs), all the members of the corresponding `data` group are concurrently updated. This is achieved by delivering all access messages from the external world to all non-faulty group members in the same order using an *atomic multicast* mechanism. Therefore, all non-faulty group members have to agree upon an order for message delivery. This is achieved using a *consensus* mechanism.

Consensus protocols. A *consensus* protocol allows a set of (possibly fail-prone) nodes to agree on a common value. Each node proposes a value, and the protocol ensures that (1) eventually all non-faulty nodes decide on a value; (2) the decided value is the same for all nodes; and (3) the decided value has been initially proposed by some node. In our case, the

decision regards the order in which messages are delivered to the group members.

Failure detectors. The consensus problem in *fully* asynchronous systems can only be solved deterministically thanks to *unreliable failure detectors* [8]. The role of these detectors is to provide a list of nodes suspected to be faulty. This list is only approximately accurate, as a non-faulty node may be suspected, and a faulty node may remain unsuspected for a while. Fortunately, there exists consensus protocols which can cope with this approximation.

To summarize, *failure detectors* are needed in order to perform *consensus* in the presence of failures; this provides a way to implement *atomic multicast*, which is the basis for replication within JUXMEM's `data` groups. While classical algorithms can be used for the higher layers of this stack, special attention needs to be paid to the design of the low-level, failure detection layer. This layer needs to fit the hierarchical structure of the grid.

4.2.3 A hierarchical approach to failure detection. A failure detection service often relies on a heartbeat or ping flow between all the nodes involved in the architecture. This induces a significant traffic overhead, which may grow as fast as the square of the number of nodes. On the other hand, grid architectures gather thousands of nodes, and no steady quality of service may be expected from the numerous network links. The failure detectors have to take this tough context into account, in order to provide suspect lists as accurate as possible.

A possible approach is to leverage on the hierarchical organization of most grids, which are made of a loosely-coupled federation of tightly-coupled clusters. Therefore, we propose to take advantage of this natural hierarchy: a similar hierarchical organization of the detectors [6] enables to reduce the overall amount of exchanged messages. Failure detection is handled at two different levels. At cluster-level, each node sends heartbeats to all the other nodes of its own cluster. Each cluster selects a mandatory, which is in charge of handling failure detection at grid level. Note that this mandatory may fail: in this case, its failure is detected at cluster-level, and another mandatory is selected. Moreover, it is possible to adapt the detection quality of service with respect to the application needs and the network load. For instance, the trade-off between detection accuracy and reactivity may be different for JUXMEM `cluster` managers, and for JUXMEM `data` providers. A detailed description of the hierarchical detector used in this design can be found in [6].

4.3 Data consistency issues in JUXMEM

JUXMEM uses replication within `data` groups to keep data available despite failures. Also, as multiple nodes perform accesses to a same piece of data, replication can be used to enhance locality, and thus performance. Consequently, JUXMEM has to manage the consistency of the different copies of a same piece of data. Consistency protocols have intensively been studied within the context of DSM systems [21]. However, an overwhelming majority of protocols assume a *static* configuration where nodes do not disconnect nor fail. It is clear that these assumptions do not hold any more in the context of a *large-scale*, *dynamic* grid infrastructure. In such a context, consistency protocols cannot rely any more on entities supposed to be stable, as traditionally was the case.

4.3.1 Fault tolerant consistency protocols. JUXMEM takes a new approach to this problem by putting scalability and fault-tolerance into the core of the design. The `data` groups use atomic multicast to perform a pessimistic replication. Therefore, critical protocol entities can be implemented using these replication groups. For instance, a large number of protocols associate to each data a node holding the most recent data copy. This is true for the very first protocols for sequential consistency [19], but also for recent *home-based* protocols implementing lazy release consistency [24] or scope consistency [17], where a *home node* is in charge of maintaining a reference data copy. It is important to note that these protocols implicitly assume that the home node never fails. Implementing the *home entity* using a replication group like JUXMEM's `data` groups allows the consistency protocol to assume that this entity is stable. Actually, any home-based consistency protocol can become fault-tolerant using this decoupled architecture (see Figure 5).

4.3.2 A scalable consistency protocol. As we are targeting a grid architecture, multiple clients in different clusters may share a same piece of data. In such a situation, it is important to minimize the inter-cluster communications, since they may have a high latency. Atomic multicast in a flat group spread over multiple physically distributed clusters would be inefficient. A hierarchical approach to consistency protocol design is then necessary. For each piece of data, a home entity should be present in every cluster that contains a potential client.

As a proof of concept, we have developed a protocol implementing the *entry consistency model* in a fault-tolerant manner. Starting from a classical home-based, *non fault-tolerant* protocol, we use replication to tolerate failures as described above. Then, in order to limit inter-cluster communications, the home entity is organized in a hierarchical way: *local homes*, at cluster level, act as clients of a *global home*, at grid level. The global home is a logical entity,

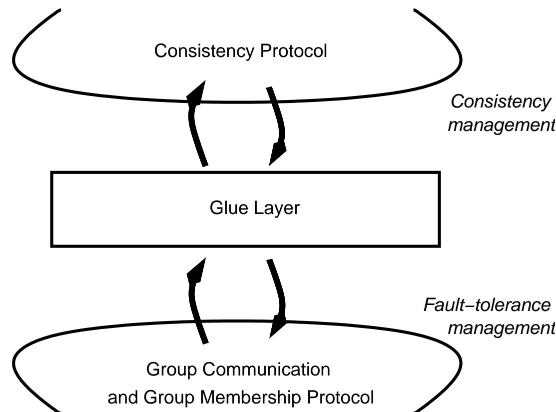

Figure 5. JUXMEM's decoupled architecture for fault tolerance and data consistency.

which is implemented by a replication group, whose members are the local homes. But note that local homes are logical entities as well, implemented as replication groups of physical nodes! A detailed description of this protocol can be found in [3].

5. Putting all elements together

The elements presented in the previous sections allow us to define an architecture for a *data-sharing service* as a hybrid approach combining the DSM and P2P paradigms, while leveraging algorithms and mechanisms studied in the field of fault-tolerant distributed systems. Previous results in each of these areas are obviously very good starting points; however, note that they cannot be directly applied in a grid context. For instance, DSM systems provide transparent access to data and interesting consistency protocols, but neglect fault tolerance and scalability. P2P systems provide scalable protocols and cope with volatility, but generally deal with read-only data and therefore do not address the consistency issue. Finally, fault-tolerant algorithms have often been subject to theoretical validations, but they have rarely been evaluated experimentally, on real large-scale testbeds. Our approach builds on these existing efforts, while taking into account *simultaneously* all these constraints inherent to a grid architecture.

The contribution of this paper is namely to propose an approach to *transparent access to data*, while addressing three important issues: *persistence*, *fault tolerance*, *consistency*. The proposed architecture (illustrated on Figure 6) fits the hierarchical architecture of a grid defined as a federation of SAN-based clusters interconnected by high-bandwidth WANs. Note that this hierarchy is taken at all levels of the architecture. The DIET computing infrastructure and

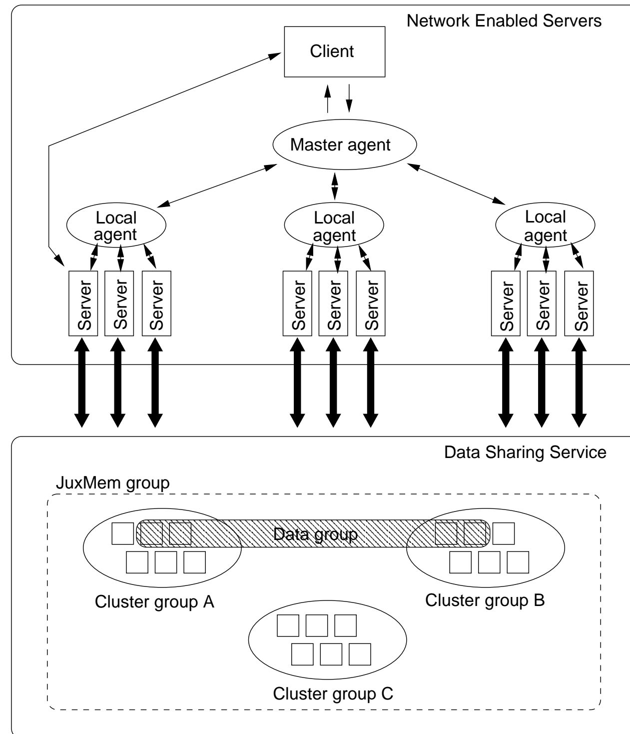

Figure 6. Overall architecture of the grid data-sharing service.

the JUXMEM entities are mapped onto the underlying resources available in the various clusters, each of which may have specific properties and policies. The failure detector used by JUXMEM is hierarchical as well, for scalability reasons.

An implementation of this integrated architecture is under way within the GDS [26] project of the French ACI MD Research Program. The hierarchical failure detector described in Section 4.2.3 has already been integrated into the JUXMEM. It is used by JUXMEM's fault-tolerant components (consensus, atomic multicast, group membership), on which rely the consistency protocols. The protocol described in Section 4.3.2 is fully operational and has been subject to a preliminary evaluation [3].

6. Conclusion

The concept of *grid computing* was initially proposed by making an analogy with the *power grid*, where the electric power is *transparently* made available to the users. No knowledge is necessary on the details of how and where elec-

tric power is produced and transported: the user just plugs in its appliance! In a similar way, in an ideal vision, using computational grids should be totally transparent: it should not be required that the user explicitly specify the resources to be used and their locations!

An important area where transparency needs to be achieved concerns data management. As opposed to most of the current approaches, based on *explicit* data localization and transfer, we propose in this paper an architecture for a *data-sharing service* providing *transparent access to data*. The user only accesses data via global identifiers. Data localization and transfer are at the charge of the service. The service also applies adequate replication strategies and consistency protocols to ensure data persistence and consistency in spite of node failures. These mechanisms target a large-scale, dynamic grid architecture, where nodes may unexpectedly fail and recover.

The modular character of the proposed architecture opens many experimentation possibilities. Various algorithms can be evaluated and tuned at the level of each layer (failure detection, replication strategies, consistency protocols, etc.). Different possible interactions between the fault tolerance layer and the consistency layer can also be experimented. The final goal is to be able to put into practice *adaptive* strategies, able to select the most adequate protocols at each level. This could be done according to some given *performance/guarantees* trade-off transparently reached by matching the application constraints with run-time information on the characteristics of the available resources.

The architecture presented in this paper is currently being connected to the DIET NetWork Enabled Server environment. The transparency of data management, data consistency, and fault tolerance are mandatory features to get the best performance at a large scale for this kind of grid middleware. The data management scenarios provided by the TLSE application offer interesting use cases for the validation of JUXMEM.

As a further step, in order to take into account the efficiency constraints expressed by the applications, one crucial issue to handle is the efficiency of *data transfers*. In this context, it is important to be able to fully exploit the potential of high-performance networks available in the grid clusters: System-Area Networks (SANs) and Wide-Area Networks (Wans). Existing high-performance frameworks for networking and multi-threading can prove helpful. PadicoTM [12] is an example of such an environment able to automatically select the adequate communication strategy/protocol in order to best take advantage of the available network resources (zero-copy communications, parallel streams, etc.). Integrating such features would be another step forward in the direction of transparency!

References

[1] William Allcock, Joseph Bester, John Bresnahan, Ann Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5):749–771, 2002.

[2] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. JuxMem: Weaving together the P2P and DSM paradigms to enable a Grid Data-sharing Service. *Kluwer Journal of Supercomputing*, 2005. To appear. Preliminary electronic version available at URL `http://www.inria.fr/rrrt/rr-5082.html`.

[3] Gabriel Antoniu, Jean-François Deverge, and Sébastien Monnet. Building fault-tolerant consistency protocols for an adaptive grid data-sharing service. In *Proceedings of the ACM Workshop on Adaptive Grid Middleware (AGridM '04)*, Antibes Juan-les-Pins, France, September 2004. Held in conjunction with PACT 2004.

[4] Dorian Arnold, Sudesh Agrawal, Susan Blackford, Jack Dongarra, Micelle Miller, Kiran Sagi, Zhiao Shi, and Sthish Vadhiyar. Users' Guide to NetSolve V1.4. Computer Science Dept. Technical Report CS-01-467, University of Tennessee, Knoxville, TN, July 2001.

[5] 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 *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '02)*, pages 194–201, Berlin, Germany, May 2002. IEEE.

[6] Marin Bertier, Olivier Marin, and Pierre Sens. Performance analysis of hierarchical failure detector. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '03)*, pages 635–644. IEEE Society Press, June 2003.

[7] 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. Springer-Verlag, August 2002.

[8] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[9] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comp. Surveys*, 33(4):427–469, December 2001.

[10] Olivier Coulaud, Michael Dussère, and Aurélien Esnard. Toward a computational steering environment based on corba. In G.R. Joubert, W.E. Nagel, F.J. Peters, and W.V. Walter, editors, *Parallel Computing: Software Technology, Algorithms, Architectures & Applications*, volume 13 of *Advances in Parallel Computing*, pages 151–158. Elsevier, 2004.

[11] Michel Daydé, Luc Giraud, Montse Hernandez, Jean-Yves L'Excellent, Chiara Puglisi, and Marc Pantel. An Overview of the GRID-TLSE Project. In *Poster Session of 6th International Meeting (VECPAR '04)*, pages 851–856, Valencia, Espagne, June 2004.

[12] Alexandre Denis, Christian Pérez, and Thierry Priol. PadicoTM: An Open Integration Framework for Communication Middleware and Runtimes. *Future Generation Computer Systems*, 19(4):575–585, May 2003.

[13] Frédéric Desprez, Martin Quinson, and Frédéric Suter. Dynamic Performance Forecasting for Network Enabled Servers in a Metacomputing Environment. In *Int. Conf. on Parallel and Dist. Proc. Tech. and Applications (PDPTA '2001)*. CSREA Press, 25-28 June 2001.

[14] Bruno Del Fabbro, David Laiymani, Jean-Marc Nicod, and Laurent Philippe. Data management in grid applications providers. In *Proceedings of the 1st IEEE Int. Conf. on Dist. Frameworks for Multimedia App. (DFMA '05)*, February 2005. To appear.

[15] Gilles Fedak, Cécile Germain, Vincent Neri, and Franck Cappello. XtremWeb : A generic global computing system. In *Proceedings of the IEEE Workshop on Global Computing on Personal Devices (GCPD 'O1)*, pages 582–587, Brisbane, Australia, May 2001.

[16] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The Int. Journ. of Supercomp. App. and High Perf. Comp.*, 11(2):115–128, 1997.

[17] 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.

[18] Tevfik Kosar and Miron Livny. Stork: Making data placement a first-class citizen in the grid. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS '04)*, pages 342–349, Tokyo, Japan, March 2004.

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

[20] Hidemoto Nakada, Mitsuhisa Sato, and Satoshi Sekiguchi. Design and Implementations of Ninf: towards a Global Computing Infrastructure. *Future Generation Computing Systems, Metacomputing Issue*, 15(5-6):649–658, 1999.

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

[22] Keith Seymour, Craig Lee, Frédéric Desprez, Hidemoto Nakada, and Yoshio Tanaka. The End-User and Middleware APIs for GridRPC. In *Proc. of the Work. on Grid App. Progr. Interfaces (GAPI '04), In conjunction with GGF12*, September 2004.

[23] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In Manish Parashar, editor, *Proceedings of the 3rd International Workshop on Grid Computing (GRID '02)*, volume 2536 of *LNCS*, pages 274–278, Baltimore, MD, USA, November 2002. Springer.

[24] 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.

[25] The JXTA project. `http://www.jxta.org/`, 2001.

[26] The GDS project: Grid Data Service. `http://www.irisa.fr/GDS/`.