FAULT-TOLERANT DATA SHARING FOR HIGH-LEVEL GRID PROGRAMMING: A HIERARCHICAL STORAGE ARCHITECTURE

Marco Aldinucci, Marco Danelutto
Computer Science Department, University of Pisa
Largo Bruno Pontecorvo 3, 56127 Pisa, Italy
aldinuc@di.unipi.it

Gabriel Antoniu, Mathieu Jan
INRIA Rennes
Campus de Beaulieu, 35042 Rennes, France
gabriel.antoniu@irisa.fr, mathieu.jan@irisa.fr

Abstract Enabling high-level programming models on grids is today a major challenge. A way to achieve this goal relies on the use of environments able to transparently and automatically provide adequate support for low-level, grid-specific issues (fault-tolerance, scalability, etc.). This paper discusses the above approach when applied to grid data management. As a case study, we propose a 2-tier software architecture that supports transparent, fault-tolerant, grid-level data sharing in the ASSIST programming environment (University of Pisa), based on the JuxMem grid data sharing service (INRIA Rennes).

Keywords: Grid, shared memory, High-level programming, memory hierarchy, P2P.

1. Introduction

Grid computing has emerged as an attempt to provide users with the illusion of an infinitely powerful, easy-to-use computer, which can solve very complex problems. This very appealing illusion is to be provided (1) by relying on the aggregated power of standard (so, inexpensive), geographically distributed resources owned by multiple organizations; (2) by hiding as much as possible the complexity of the distributed infrastructure to users. However, the current status in most software grid infrastructures available today is rather far away from this vision. When designing programs able to run on such large-scale platforms, programmers often need to explicitly take into account resource heterogeneity, as well as the unreliability of the distributed infrastructure. In this context, the grid community converges towards a consensus about the need for a high-level programming model, whereas most of the grid-specific efforts are moved away from programmers to grid tools and run-time systems. This direction is currently pursued by several research initiatives and programming environments, such as ASSIST [17], eSkel [9], GrADS [14], ProActive [8], Ibis [16], Higher Order Components [11], etc., along the lines of CoreGRID's "invisible grid" approach to next generation grid programming models.

In this work, we explore the applicability of the above approach to data management for high-level grid programming. We consider three main aspects that need to be automatically handled by the data storage infrastructure: data access transparency, data persistence and storage fault-tolerance.

Transparent access to data across the grid. One of the major goals of the grid concept is to provide an easy access the underlying resources, in a *transparent* way. The user should not need to be aware of the localization of the resources allocated to the application submitted. When applied to the management of the data used and produced by applications, this principle means that the grid infrastructure should automatically handle data storage and data transfer among clients, computing servers and storage servers as needed. However, most projects currently still rely on the *explicit data access model*, where clients have to move data to computing servers. In this context, grid-enabled file transfer tools have been proposed, such as GridFTP [5], DiskRouter [12], etc. In order to achieve a real virtualization of the management of large-scale distributed data, a step forward has been made by proposing a *transparent data access model*, as a part of the concept of *grid data-sharing service*, illustrated by the JuxMem software experimental platform [6].

Persistent storage of data on the grid. Since grid applications typically handle large masses of data, data transfer among sites can be costly, in terms of both latency and bandwidth. Therefore, the data-sharing service has to provide persistent data storage. Data produced by one computation can be made available to some other computation through direct access via globally shared data identifiers, by avoiding repeated transfers of large volumes of data between the different components of the grid.

Fault-tolerant storage of grid data. Data storage on the grid must cope with events such as storage resources joining and leaving, or unexpectedly failing. Replication techniques and failure detection mechanisms are thus necessary, in order to enhance data availability despite disconnections and failures. Such mechanisms need to be coupled with checkpoint/restart techniques for application components with data replication mechanisms. This way, in reaction to faults, checkpointed application components could be migrated to available resources and restarted, as long as the application status and the application data remain available thanks to fault-tolerant storage.

While the above properties are desirable in general for grid data sharing, in this paper we restrict our discussion to the more specific case of applications based on the ASSIST [3] programming environment, developed at the University of Pisa. ASSIST provides a programming model that enables transparent data sharing among distributed execution entities. However, its dedicated storage component (called ad-HOC [4]) does not enforce data persistence and is not tolerant to node failures. Such features are exhibited by the JUXMEM [6] grid data-sharing service developed at INRIA Rennes, which enables transparent access to grid-scale storage, based on P2P techniques. The goal of this paper is to study how ASSIST and JUXMEM could be integrated, in order to better support a high-level grid programming approach.

2. Analysis: using JUXMEM to enable grid-level, fault-tolerant storage in ASSIST

This section describes ASSIST's main features, then briefly introduces the data sharing facilities available with ad-HOC and JUXMEM (which exhibit complementary features); it finally proposes an integrated architecture which takes advantage of this complementarity.

2.1 Data sharing in ASSIST

The ASSIST programming model. ASSIST applications are described by means of a coordination language, which can express arbitrary graphs of software *modules*, interconnected by typed streams of data. Modules can be either sequential or parallel. A sequential module wraps a sequential function. A parallel module *(parmod)* can be used to describe the parallel execution of a number of sequential activities that run as *Virtual Processes* (VPs) on items coming from input streams. Each stream realizes a one-way, one-to-many asynchronous channel between sequential or parallel modules. VPs can be organized according to programmer-defined topologies (array, etc.). The ASSIST compiler implements VPs according to the underlying middleware. It selects an appropriate granularity by mapping a subset of a *parmod*'s VPs to the basic execution entity, e.g. a POSIX process/thread, a Globus Grid Service, or a Web Service. The full support for the forthcoming CoreGRID Grid Component Model (GCM) is currently under investigation [10, 2].

Data sharing in ASSIST. The ASSIST programming model enables data sharing both within VPs of the same module and between different modules by means of two different methods, respectively:

Attributes, which are global, typed variables of the module. Attributes are owned either by the module itself or by VPs; in particular arrays ownership can be partitioned among VPs according to user-defined rules.

References, which are global pointers in a shared memory space that is logically external to all application modules, thus can be accessed across different modules. ASSIST provides the programmer with an API to allocate, read, and write data structures in the shared space.

Note that, in both cases, ASSIST does not implement a built-in DSM to enable memory sharing; it rather provides hooks to attach ASSIST run-time support to an external DSM. This allows to test different solutions for data sharing, since the DSM can be replaced with almost no impact on the core code of the compiler and of its run-time support. Currently, ASSIST comes with a DSM built on top of a set of cooperating ad-HOC memory servers [4].

These variables adhere to the *transparent data access model*: programmers should neither care about data placement nor data splitting and distribution. These tasks are transparently and efficiently carried out by ad-HOC. However, in order to enforce fault-tolerance (and irrespectively of the checkpointing technique), ASSIST run-time needs to rely on permanent and robust data storage.

2.2 Existing building blocks: ad-HOC and JuxMem

Cluster-level sharing: ad-HOC. ad-HOC (Adaptive Distributed Herd of Object Caches), is a distributed *object* repository [4] developed at University of Pisa. It provides the programming environment designer with building blocks to set up client-server and service-oriented infrastructures for data storage management. Clients may access data in servers through different protocols, which are implemented on client-side within a *proxy* library.

A set of cooperating ad-HOC servers implements a permanent storage facility, i.e. a repository for arbitrary length, contiguous segments of data, namely *objects* since each data chuck is wrapped and stored and in an object homed in one of the servers [4]. Objects can be grouped in ordered *collections* of objects, which can be spread across different servers. Both objects and their collections are identified by *keys* with fixed length. In particular, the key of a collection specifies to which *spread-group* the collection belongs. Such a group specifies how adjacent objects in the collection are mapped across the set of servers. Both classes of shared objects described in the previous section (attributes and references) are implemented in ASSIST via ad-HOC object collections and can thus be stored in a distributed way. The ad-HOC API enables to `get/put/remove` an object, and to `create/destroy` a key for a collection of objects. Each ad-HOC manages an *object storage* area for server home objects and a write-back *cache* for objects with a remote home. Basic ad-HOC operations do not natively ensure data coherence of cached objects. Nevertheless, server operations can be extended via the special operation `execute` that enables application proxies to put a serialized C++ object[1] in a server and invoke its *run* method to issue a consistent access (e.g. lock/unlock, coherent put, etc).

The extremely simple ad-HOC API is mainly aimed to implementation efficiency, which relies on non-blocking I/O. On each port, an ad-HOC can efficiently serve many clients, each of which supports thousands of concurrent connections while squeezing a close to ideal bandwidth even for fine-grained data accesses [4]. A set of ad-HOCs can efficiently cooperate across multi-tier networks and clusters with private address ranges, even when they are protected by firewalls. However, ad-HOC does not implement any form of fault-tolerance or automatic data replication: in the case a server permanently leaves the community implementing the data storage, some of the stored data will be lost. This problem is particularly severe if data structures are spread across all servers, since losing a single server may induce the corruption of the whole dataset.

Grid-level sharing: JuxMem. The JUXMEM [6] grid *data sharing service* developed at INRIA Rennes transparently manages data localization and persistence in dynamic, large-scale, distributed environments. The *data sharing service* concept is based on a hybrid approach inspired by Distributed Shared Memory (DSM) systems (for transparent and consistent data sharing) and peer-to-peer (P2P) systems (for scalability and volatility tolerance). The data sharing service can also transparently replicate data to ensure its persistence in case of failures. The consistency of the various replicas is then implemented through adequate fault-tolerant consistency protocols [7]. This approach will serve as a basis to the architecture proposed in this paper.

The JUXMEM API provides the users with classical functions for allocating (`juxmem_malloc`) and mapping/unmapping memory blocks (`juxmem_mmap`, etc.) When allocating a memory block, the client can choose to replicate data to enhance fault-tolerance, and then has to specify: (1) on how many clusters the data should be replicated; (2) on how many providers in each cluster the

[1]Object data is serialized at run-time and send to the server, while code must be provided (in advance) to the server as dynamically linkable library.

data should be replicated; (3) the consistency protocol that should be used to manage this data. This results into the instantiation of a set of data replicas (associated to a group of nodes), called *data group*. The allocation operation returns a global data ID. This ID can be used by other nodes in order to identify existing data. It is JUXMEM's responsibility to localize the data and perform the necessary data transfers based on this ID. This is how JUXMEM provides a transparent access to data. To obtain read and/or write access on a data, a process that uses JUXMEM should acquire the lock associated to the data through either `juxmem_acquire` or `juxmem_acquire_read`. This permits to apply consistency guarantees according to the consistency protocol specified by the user at the allocation time of the data. Note that `juxmem_acquire_read` allows multiple readers to simultaneously access the same data.

The general architecture of JUXMEM mirrors a federation of distributed clusters and is therefore *hierarchical*. The goal is to accurately map the physical network topology, in order to efficiently use the underlying high performance networks available on grid infrastructures. Consequently, the architecture of JUXMEM relies on node sets to express the hierarchical nature of the targeted testbed. They are called *cluster groups* and correspond to physical clusters. These groups are included in a wider group, the *juxmem group*, which gathers all the nodes running the data-sharing service. Any cluster group consists of *provider* nodes which supply memory for data storage. Any node may use the service to allocate, read or write data as *clients*, in a peer-to-peer approach. This architecture has been implemented using the JXTA [1] generic P2P platform.

Table 1. Properties for cluster-level and grid-level sharing.

	Cluster sharing (ad-HOC)	**Grid sharing (JuxMem)**
Throughput	High	Medium/High
Latency	Low	High
Fault-tolerance	No	Yes
Protocols for data consistency	No	Yes

ad-HOC and JuxMem: a quick comparison at hand. Due to different design goals, ad-HOC distributed storage server and of the JUXMEM data-sharing service exhibit different features (as summarized in Table 1).

ad-HOC provides transparent and permanent access to distributed data. It enables the distribution and the parallel access to collections of objects (e.g. arrays). It is robust w.r.t. node additions and removal, which should be driven by a proper node agreement protocol to guarantee data safety. It does not checkpoint/replicate data, thus is not fault-tolerant. Data accesses exhibit low

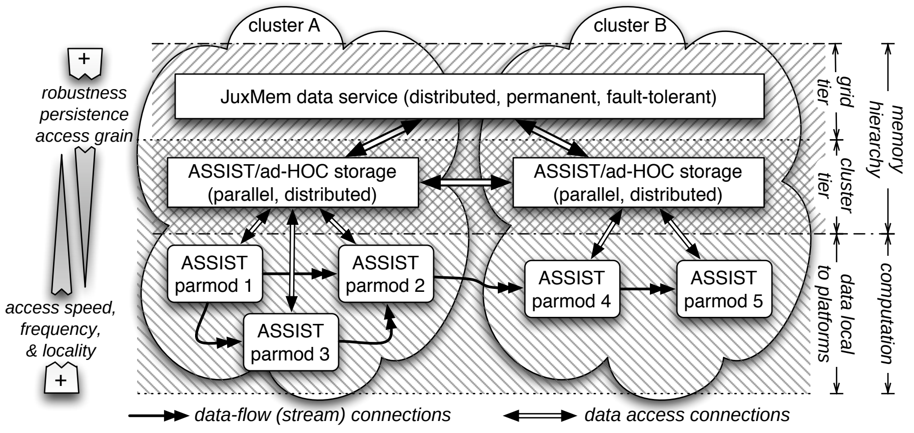

Figure 1. Target architecture illustrating the interaction of ad-HOC and JuxMem.

latency, high-bandwidth, high-concurrency and scalability. It is mainly targeted to *cluster-level* sharing. It is developed in C++.

JUXMEM provides transparent and permanent access to distributed data. It is fully fault-tolerant w.r.t. node additions and fail-stop by means of data consistent replication. Data accesses exhibit a higher latency, medium to high-bandwidth, and scalability. It relies on the JXTA generic P2P framework. It is targeted to *grid-level* sharing. It is developed in C and Java.

3. Proposal: an integrated 2-tier architecture

Memory hierarchy is an effective way to enhance data management efficiency while respecting data access transparency. We propose a 2-tier memory hierarchy which trades off storage performance vs. robustness: it enables ASSIST modules to rely on a fault-tolerant grid storage while still exploiting low-latency/high-bandwidth data accesses, which are typical of a fast, cluster-level DSM. The big picture is sketched in Fig. 1: the two levels of hierarchy are called *grid tier* and *cluster tier*. The hierarchy is connected to an additional tier where computation is figured out (at the bottom on the figure). This third level (represented at the bottom) also includes data storage that is local to platforms[2]. The top two tiers of the memory hierarchy are specified as follows:

The grid tier enables data sharing across multiple *parmods* running on *different* clusters. It is implemented via JUXMEM. Accesses to data stored

[2]Register file level in classic memory hierarchies.

in JUXMEM are assumed to be *less frequent, coarse-grained, with less performance constraints.*

The cluster tier enables data sharing among multiple parallel codes (VPs, *parmods*) within a single cluster. It is implemented via an ad-HOC-based storage component. Accesses to data stored in ad-HOC are assumed to be *frequent, fine-grained, with high-performance constraints.*

In the following, we define why and when the data stored at the cluster level should be copied/stored at the grid level and vice-versa. The answers come directly from the analysis of hierarchy goals.

Sharing across multiple clusters. A natural interaction between the two levels of the hierarchy will take place whenever data produced and stored on a cluster (at the ad-HOC level) has to be made available on another cluster (multi-site deployment). At the programming level this may be triggered by passing a reference across the clusters via streams. Notice that this functionality can also be implemented at ad-HOC level by "horizontally" connecting ad-HOC storage components of different clusters. However, the hierarchical solution has some advantages: (1) data can be shared also among different applications thanks to persistence of grid level; (2) ASSIST does not need any more to handle the co-allocation of resources (possibly via job schedulers) for all *parmods* distributed on different clusters, since JUXMEM can serve as temporary storage; (3) data shared at this level can be considered safe w.r.t. node and networks faults (as this aspect is handled by the JUXMEM-based grid-level).

The first point allows the user to use command scripts implementing a functionality similar to UNIX pipes in grid distributed fashion, while sharing data among the different processes through JUXMEM. The second point addresses a very hot topic in grid middleware design: multi-site co-allocation. Let us suppose as an example that modules mapped on cluster A (see Fig. 1) start well before modules on cluster B. Data coming on stream from parmod 2 can be transiently buffered on grid storage up to the moment parmod 4 starts. This scenario can be extended to all cases in which data dependencies among clusters can be described by a Direct Acyclic Graph (co-allocation is not logically needed). The third point is discussed in details in the next paragraph.

Fault tolerant data storage/checkpointing. Since the ad-HOC provides efficient access latency, it can be used as a cache for intra-cluster data sharing. Since it is not fault-tolerant, the ad-HOC can periodically save application data to JUXMEM. This feature could also be used for application checkpointing in general, as the computation status can be saved together with the data. Checkpointing is driven by ASSIST run-time support, which stores checkpoint information on the cluster tier, and then triggers a flush of "dirty" objects to the grid tier. Basically the cluster tier is used as a write-back cache. In the same way, this information could also be used to migrate *parmods* from one cluster to another. In such a case, ad-HOC data can be saved to JUXMEM on the initial cluster and then read from JUXMEM after migration on the second cluster.

Locality. The use of a memory hierarchy adequately supports clustered locality. Grids are generally structured as federations of clusters, where each cluster is pretty homogeneous and exhibits high-bandwidth/low-latency connectivity due to the spatial proximity and to the reduced security constraints, but also to the use of high-performance System Area Networks, such as Myrinet or Infiniband. High-level parallel languages and their run-time environments aim at enhancing clustered locality (1) by providing programmers with language paradigms (constructs, skeletons, design patters, etc.) leading to known and regular interaction patterns among concurrent activities; (2) by statically and/or dynamically mapping/enforcing clustered locality of a groups of activities demanding frequent interactions. A few examples are: the iterated halo-swap paradigm and the block data distribution [9], the Divide&Conquer paradigm supported by dynamical load balance based on hierarchical work-stealing [15]. Component technology and related design methodologies further enforce clustered locality by encouraging application designer to aggregate related activities within a (possibly compound) component. As result, most of the shared memory accesses have a limited scope in terms of accessing entities, and those entities are preferably mapped and deployed on the same cluster, thereby improving the local-to-remote access ratio.

4. ASSIST and fault-tolerance: a sample scenario

The simplest scenario showing how to exploit the benefits of the integrated 2-tier architecture leverages the fault tolerant data storage/checkpointing features described above. In this scenario, parmods are considered in insulation. The goal is to protect them against a site fail-stop (due either to node crashes or to connectivity failures). The nature of the parmod construct suggests that there exist a quite strong relationship among the parallel activities within it, whereas activities in different parmods definitely exhibit a weaker relationship. A parmod is therefore usually deployed within a single site. The parmod status, which is normally stored in the ad-HOC tier, can periodically be snapshotted and saved into the JUXMEM tier. As we shall see in the next section, this can be done by leveraging already existing mechanisms that enforce the generation of a coherent snapshot of the status. The snapshot can be enriched with all the information needed to restart (in some other site) the parmod from the snapshot that can retrieved from a JUXMEM-managed backup copy living on a still alive site. Other parmods can detect the site crash through a broken stream event; among those, a leader elected via JUXMEM tier support, can restart a new copy of the parmod by instanciating the snapshotted status. If necessary, the

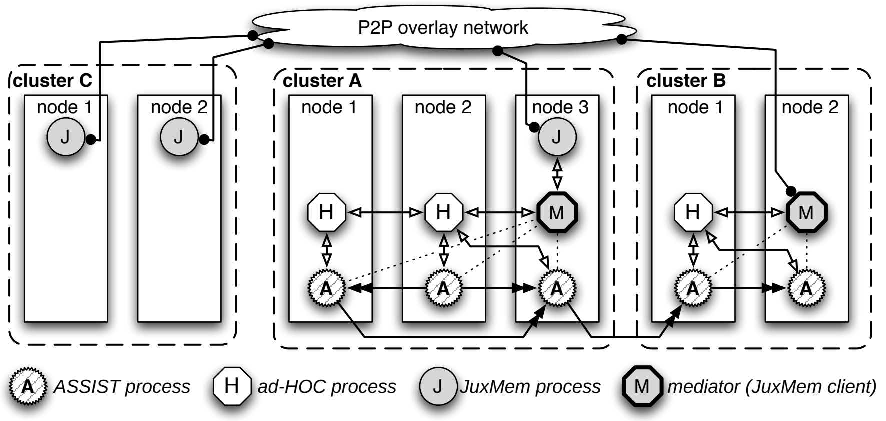

Figure 2. Implementation of proposed architecture using ad-HOC and JUXMEM (the application is sketched in Fig. 1). Dashed lines represent synchronizations inducing mediator actions.

parmods directly connected with the restored one should update their status before proceeding.

5. Design and Implementation

A prototypal implementation has recently been completed and is currently being tested. It will mainly be used to analyze the hierarchy's efficiency and limitations in order to refine the design phase. The current implementation relies on a set of external *mediator* processes that are started with ad-HOC servers by the ASSIST launcher. The mediator can take into account remote commands (through a TCP port), which trigger data copying from the cluster tier to the grid tier and vice-versa. Available commands include copying a set of ad-HOC objects (which can be spread across the nodes of the distributed ad-HOC server) to new or to existing JUXMEM memory areas, and vice-versa. In both cases, the data exchanged between the two tiers flow through the mediator. In one case, it reads a series of data objects from ad-HOC, collapses them in a single data chunk, and writes it into JUXMEM; in the other case, it fetches a data chunk from JUXMEM, splits and spreads it to ad-HOC servers of the cluster. Such an interaction is represented in Fig. 1.5, where J processes act as JUXMEM storage providers and M processes (representing the mediators) act as JUXMEM clients. Mediators and providers can be hosted on the same node, on different nodes in the same cluster or on different clusters.

These tasks leverage existing features of the ASSIST compiler. Since the compiler can produce dynamically adaptable code, it already identifies within *parmod* so-called *reconf-safe* points and instruments it with the required agreement protocols for coordinated checkpointing. In these points, the compiler can ensure a coherent and known state of application processes and the shared memory. This information can be dumped, as it can serve to restart the whole application or *a proper subset* of it from last *reconf-safe* point. This technique has been successfully used to enable dynamic migration of VPs and *parmods* [10].

Fault-tolerant storage. The proposed architecture can transparently be used to provide fault-tolerant storage for ASSIST by using the primary-backup approach, where the cluster tier behaves as a primary replica and the grid tier as backup. In particular, *attributes* can be safely stored in *reconf-safe* points. In these points ASSIST run-time triggers an update of backup copies via the mediator process. A similar technique is used for *references*. However since these are managed directly by programmers in the user code, the triggering is achieved by extending the API with a `safe_write` operation (alternatively, the `write` operation can be transparently turned into a `safe_write` operation).

Checkpoint/rollback. A fault-tolerant storage is necessary but not sufficient to make ASSIST fault-tolerant. However, a *parmod* can easily be made fault-tolerant by triggering a checkpoint of its data *and status* into JUXMEM. Starting from one of these checkpoints, a whole *parmod* can be rolled back by using already existing primitives for dynamic adaptation (see [10] for details).

Ensuring fault-tolerance of a complete ASSIST application is slightly more complex. This requires to equally taking into account the global status of streams established among *parmods*. Classic techniques based on message logging could than be used, while relying on the proposed memory hierarchy as a stable storage. This point is under investigation.

5.1 Preliminary Experiments

The main goal of preliminary experiments conducted on the 2-tier architecture aims to assess and quantify the behaviour of the two tiers, independently considered, on the same running environment. At this end we run a set of read/write benchmarks between the mediator and the two tiers on a cluster composed of Intel Xeon5150@2.66GHz 8GBytes RAM nodes wired via a GigaEthernet. Benchmarks consist in the writing or reading of 1000 objects of a given size. Tested configurations and results of write throughput benchmark are shown in Fig. 3 for the ad-HOC tier, and Fig. 4 for the JUXMEM tier. Each tier is tested for its own peculiar features, i.e. objects distribution onto a set of parallel servers for ad-HOC; and objects replication onto a set of memory providers for JUXMEM. Both sets of tests include the extreme case of a single server (no distribution, no replication) where the two tiers behave identically. Tests show that the two tiers achieve a very good throughput.

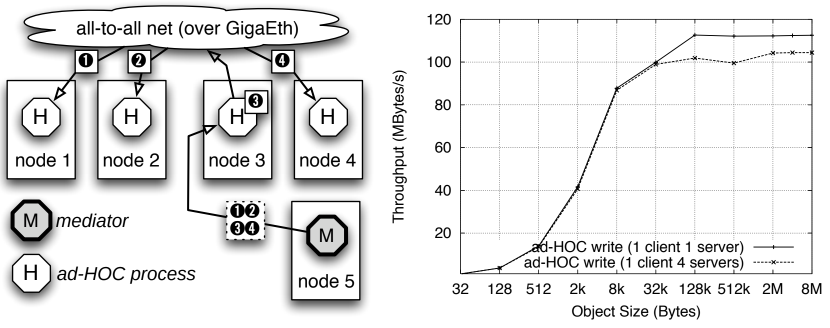

Figure 3. Preliminary results: the tested architecture and the mediator throughput on ad-HOC side. In the first experiment objects are stored on a single ad-HOC server, in the second objects are distributed onto 4 ad-HOC servers.

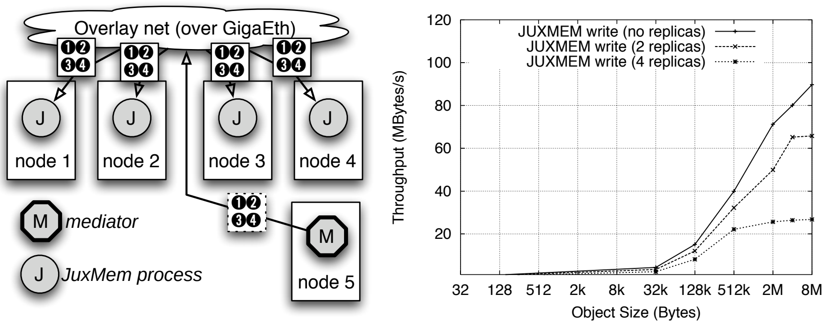

Figure 4. Preliminary results: the tested architecture and the mediator throughput on JUXMEM side. In the first experiment objects are stored on a single JUXMEM provider, in the second and the third objects are replicated across 2 and 4 JUXMEM providers, respectively.

As mentioned above, the ad-HOC does not include any protocol to manage consistent data replication, thus it implements a lighter data access protocol with respect to JUXMEM. This enables ad-HOC to achieve the same throughput of JUXMEM for a smaller object size. Read throughput benchmarks, which are not reported in the paper, gives almost identical figures. Notably, the ad-HOC exploits a slightly better throughput when objects are not distributed (single server case). This is due to the single connection between the mediator and the ad-HOC servers. The bottleneck can be removed by connecting the mediator with more than one ad-HOC server [4].

Differently from ad-HOC, JUXMEM can transparently and consistently manage object replication: this enables the definition of a permanent, fault-tolerant tier. Obviously, consistent data replication has a cost. As shown in Fig. 4, this cost linearly depends from the number of replicas in the case of write operation. On the contrary, read operations exhibit a throughput similar to write with no replicas independently of the number of replicas.

Overall, the preliminary experiments confirm the complementarity of the two tiers, which deliver their optimal data throughput at different object size. Experimental results enable to reason about the size increase and frequency reduction the mediator should perform between the two tiers. These values do not depends only from platforms and networks speed, but also from distribution and replication degree chosen for the two tiers, respectively.

6. Conclusion

This paper addresses the problem of how to support of high-level grid programming by means of a software infrastructure that automatically and transparently handles low-level, grid-specific issues, such as multi-site resource distribution, fault-tolerance, etc. It proposes a hierarchical grid data storage architecture whose goal is to provide the ASSIST grid programming environment with grid-scale, fault-tolerant data-sharing facilities, as provided by the JUXMEM grid data-sharing service. This work is a specific integration effort between existing researches carried out by two partners within the Institute on Programming Model (WP3) of the CoreGRID NoE. We mainly focus on architecture design, however a prototypal implementation has recently been completed and preliminary experiments are under way. The work in progress concerns the assessment of the performance of memory accesses between the two tiers of the hierarchy.

Long-term aims. Many approaches to data management on grid aim to optimize the access to very large, quite localized, mostly read-only scientific data [13]. However, in many real applications, data is inherently distributed across application components. We believe that a mature programming model for the grid should provide designers with an abstract view of the data, and a high-level

API to access it. Due to the hierarchical nature of typical grid platforms, any implementation of data management system will benefit from a clear understanding of qualitative and quantitative aspects governing a distributed memory hierarchy on the grid. These aspects may be used to enhance and automatize the distributed management of data in development tools for the grid. The proposed architecture appears flexible enough to investigate the benefits and overheads of hierarchical data management, and the extent to which data can be transparently moved (horizontally and vertically) across the tiers of the hierarchy.

Acknowledgments. We are grateful to Massimo Torquati and Daniele Buono who contributed to software development and experimentation. This work has been partially supported by the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265), and by the Region of Brittany (France).

References

[1] The JXTA (juxtapose) project. `http://www.jxta.org`.

[2] M. Aldinucci, C. Bertolli, S. Campa, M. Coppola, M. Vanneschi, L. Veraldi, and C. Zoccolo. Self-configuring and self-optimizing grid components in the GCM model and their ASSIST implementation. In *Proc of. HPC-GECO/Compframe (held in conjunction with HPDC-15)*, IEEE, pages 45–52, Paris, France, June 2006.

[3] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*, chapter 10, pages 230–256. Springer, Jan. 2006.

[4] M. Aldinucci and M. Torquati. Accelerating apache farms through ad-HOC distributed scalable object repository. In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *Proc. of 10th Intl. Euro-Par 2004 Parallel Processing*, volume 3149 of *LNCS*, pages 596–605. Springer, Aug. 2004.

[5] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke, and I. Foster. Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing. In *Proc. of the 18th IEEE Symposium on Mass Storage Systems (MSS 2001), Large Scale Storage in the Web*, page 13, Washington, DC, USA, 2001. IEEE Computer Society.

[6] G. Antoniu, L. Bougé, and M. Jan. JuxMem: An adaptive supportive platform for data sharing on the grid. *Scalable Computing: Practice and Experience*, 6(3):45–55, Sept. 2005.

[7] G. Antoniu, J.-F. Deverge, and S. Monnet. How to bring together fault tolerance and data consistency to enable grid data sharing. *Concurrency and Computation: Practice and Experience*, 2006. To appear.

[8] F. Baude, D. Caromel, and M. Morel. On hierarchical, parallel and distributed components for grid programming. In V. Getov and T. Kielmann, editors, *Proc. of the Intl. Workshop on Component Models and Systems for Grid Applications*, CoreGRID series, pages 97–108, Saint-Malo, France, Jan. 2005. Springer Verlag.

[9] M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[10] CoreGRID NoE deliverable series, Institute on Programming Model. *Deliverable D.PM.02 – Proposals for a Grid Component Model*, Nov. 2005.

[11] J. Dünnweber and S. Gorlatch. HOC-SA: A grid service architecture for higher-order components. In *IEEE International Conference on Services Computing, Shanghai, China*, pages 288–294. IEEE Computer Society Press, Sept. 2004.

[12] G. Kola and M. Livny. Diskrouter: A Flexible Infrastructure for High Performance Large Scale Data Transfers. Technical Report CS-TR-2003-1484, University of Wisconsin-Madison Computer Sciences Department, Madison, WI, USA, 2003.

[13] E. Laure, H. Stockinger, and K. Stockinger. Performance engineering in data Grids. *Concurrency & Computation: Practice & Experience*, 17(2–4):171–191, 2005.

[14] S. Vadhiyar and J. Dongarra. Self adaptability in grid computing. *Concurrency & Computation: Practice & Experience*, 17(2–4):235–257, 2005.

[15] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *PPoPP '01: Proc. of the 8th ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 34–43, New York, NY, USA, 2001. ACM Press.

[16] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a flexible and efficient Java-based grid programming environment. *Concurrency & Computation: Practice & Experience*, 17(7-8):1079–1107, 2005.

[17] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, Dec. 2002.