A practical evaluation of a data consistency protocol for efficient visualization in grid applications* **

Gabriel Antoniu[1], Loïc Cudennec[2], and Sébastien Monnet[3]

[1] IRISA/INRIA, Gabriel.Antoniu@irisa.fr
[2] IRISA/INRIA, Loic.Cudennec@irisa.fr
[3] IRISA/University of Rennes I, Sebastien.Monnet@irisa.fr

Abstract. Data visualization is important in the context of grid applications, especially when successive refinements are iteratively realized based on intermediate results. We mainly focus on code coupling grid applications, structured as a set of distributed, autonomous, weakly-coupled codes. We consider the case where the codes are able to interact using the abstraction of a shared data space. In previous work, we have proposed an efficient visualization scheme by introducing a new operation called *relaxed read*, as an extension to the *entry consistency* model. This operation can efficiently take place without locking, in parallel with *write* operations. On the other hand, the user has to relax the consistency constraints, and accept slightly older versions of the data, whose "freshness" can however still be controlled. In this paper, we discuss and extensively evaluate the proposed consistency protocol, whose efficiency is clearly demonstrated by our experimental results.

Keywords

Data consistency, code-coupling applications, grid, visualization.

1 Introduction

With the growing demand of computing power, grid computing [11] has emerged as an appealing approach, allowing to federate and share computing and storage resources among multiple, geographically distributed sites (universities, companies, etc.). Thanks to this aggregated computing power, grids are typically useful to solve computationally intensive, parallel and/or distributed applications. In most cases, grids consist of a hierarchical federation of clusters. This hierarchy is defined in terms of hierarchical distribution, with a direct impact on the communication latency. Low-latency System-Area Networks (SANs), such as Giga Ethernet or Myrinet are often used to connect nodes within a given cluster. The various clusters may be interconnected through a higher-latency network, which can be a dedicated Wide-Area Network (WAN) whose bandwidth may reach 1 Gb/s or more.

A particular class of applications running on grids relies on the *code-coupling* paradigm: such an application is designed as a set of (usually) parallel codes, each of which runs on a different cluster. The computation is distributed in such a way that transfers between clusters are minimized. However, some data and synchronization messages still have to be exchanged among the clusters.

Code-coupling is used in high-performance computing. Computations can be very long, and it is generally impractical to wait for the end of the application to see if the results are correct. In order to monitor the progress of the application, it is often useful to have the ability to perform an efficient visualization of the running process, without degrading the overall performance of the computation. To allow the state of the computation to be monitored, pieces of data shared by different codes need to be accessed.

In grid environments, as in other distributed systems, data sharing is a crucial issue. Currently, the most widely-used approach relies on the *explicit data access model*, where clients have to move data to computing servers. A typical example is the use of the GridFTP protocol [3]. Though this protocol provides

* This work was supported by the GDS project (ACI MD - French Ministry of Research, INRIA, CNRS), by the RESPIRE project of the French National Research Agency (ARA MDMSA), by the Regional Council of Brittany and by Sun Microsystems

** Candidate to the Best Student Paper Award

authentication, parallel transfers, checkpoint/restart mechanisms, etc., it is still a transfer protocol which requires *explicit* data localization by the programmer. Such a low-level approach makes data management on grids rather complex. On the other hand, the concept of *transparent data access* in distributed systems through the illusion of a shared memory has intensively been studied in the context of distributed shared memory systems (DSM) since the late eighties ([12,10,4,9]). Nevertheless, DSM systems have been designed to address small scale physical architectures, usually made of tens (up to a hundred) of nodes and have usually been used on clusters. Furthermore, most of the data consistency models and protocols assume that the infrastructure is *static*, *without failures*. For instance, they often implicitly assume stable entities. These hypotheses are not longer valid within the grid context, where failures are part of the systems' properties. Therefore, *fault tolerance* and *volatility* increase the difficulty of designing a system providing transparent data access. The predominance of grid systems based on *explicit* transfers (GridFTP [3], IBP [8], etc.) demonstrates that transparent data sharing upon large scale architectures is still a real challenge.

In order to overcome these limitations and make a step forward towards a real virtualization of the management of large-scale distributed data, the concept of *grid data-sharing service* has been proposed [5]. The idea is to provide *transparent access* to distributed grid data: in this approach, the user accesses data via global identifiers. The service which implements this model handles data localization and transfer without any help from the programmer. It transparently manages data persistence in a dynamic, large-scale, distributed environment. The data sharing service concept is based on a hybrid approach inspired by Distributed Shared Memory (DSM) systems (for transparent access to data and consistency management) and peer-to-peer (P2P) systems (for their scalability and volatility-tolerance). The JuxMem (Juxtaposed Memory) platform [5] (described in more detail in Section 2) illustrates the grid data-sharing concept. JuxMem relies on JXTA [1], a generic P2P software platform initiated by Sun Microsystems. JuxMem also serves as an experimental framework for fault-tolerance strategies and data consistency protocols.

We focus on the problem of efficient data visualization within code-coupling applications designed for grid architectures. The goal is to modify the data consistency protocol behavior in order to efficiently support the presence of a visualization process (that we call *observer*). To this purpose we have proposed an *extension* of the *entry consistency* model and a corresponding protocol that allows efficient reads, *possibly concurrent* with writes to a given data. As a counterpart, the observer has to relax the consistency constraints, and accept slightly older versions of the data, whose "freshness" can however still be controlled. The approach underlying this work has first been introduced in [6]. In this paper, we discuss and evaluate the extension of the data consistency protocol. An implementation of this strategy has been integrated within the JuxMem platform and experimented on the Grid'5000 testbed [2]. Preliminary experimental results of this work show that this solution improves the performance of the visualization observer without degrading the performance of the application that keeps reading and writing the observed data.

The next Section introduces the JuxMem grid data sharing service. Section 3 briefly describes the consistency model and explains the proposed protocol extensions. An experimental evaluation is presented in Section 4. Finally, Section 5 discusses the contribution and the future work.

2 JuxMem : A decoupled architecture combining data consistency and fault-tolerance

2.1 JuxMem overview

To experiment our approach, we have used the JuxMem software experimental platform for grid data sharing, described in [5]. From the user's perspective, JuxMem is a service providing transparent access to persistent, mutable shared data.

JuxMem has a *hierarchical* software architecture, which mirrors a hardware architecture consisting of a federation of distributed clusters. Figure 1 shows the hierarchy of the entities defined in JuxMem, consisting of a network of peer groups (`cluster` groups A, B and C on the figure), which usually correspond to clusters at the physical level. 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*. Within each `cluster` group, the available providers are managed by a node called

cluster manager. Finally, a node which simply uses the service to allocate and/or access data blocks is called *client*. A node may at the same time act as a cluster manager, as a client, and as a provider. However, for the sake of clarity, each node only plays a single role on the figure.

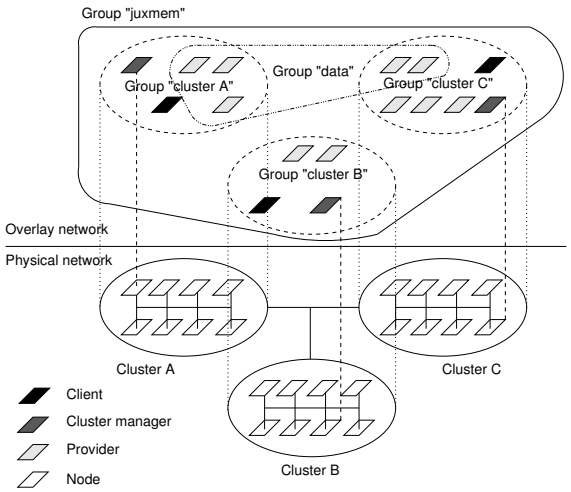

Fig. 1. Hierarchy of the entities in the network overlay defined by JuxMem.

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, associated to a group of peers called `data` group. The allocation primitive returns a global data ID, which can be used by the other nodes to identify existing data. To obtain read and/or write access to a data block, the clients only need to use this ID.

The `data` group is also hierarchically organized, as illustrated on Figure 2: the *Global Data Group (GDG)* gathers all provider nodes holding a replica of the same piece of data. These nodes may be distributed in different clusters, thereby increasing the data availability if faults occur. The GDG group is divided into *local data groups (LDG)*, which correspond to data copies located in the same cluster.

In order to access a piece of data, a client has to be attached to a specific LDG. Then, when the client performs the read/write and synchronization operations, the consistency protocol layer manages data synchronization and data transmission between clients, LDGs and GDG, with the strict respect of the consistency model.

2.2 Starting point: a hierarchical, fault-tolerant consistency protocol

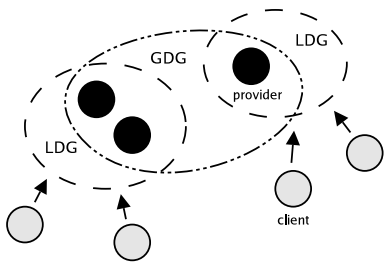

Fig. 2. JuxMem : a hierarchical architecture.

The entry consistency model To guarantee data consistency, JuxMem provides a consistency protocol that implements the entry consistency model. This model was first introduced in the Midway system [9]. As opposed to other relaxed models, it requires an explicit association of data to synchronization objects. This allows the model to leverage the relationship between a synchronization object that protects a critical section, and the data accessed within that section. A node's view of some data becomes up-to-date only when the node enters the associated critical section. This eliminates unnecessary data traffic, since only nodes that declare their intention to access data will get updated, and only the data which will be accessed will be updated. Such a concern for efficiency makes this model a good candidate in the context of scientific grid computing.

When using the entry consistency model, exclusive accesses to shared data have to be explicitly distinguished from non-exclusive accesses by using two different primitives: `acquire`, which grants mutual exclusion; `acquireRead`, which allows non-exclusive accesses on multiple nodes to be performed in parallel.

Adapting the entry consistency to the grid Existing protocols that implement the entry consistency model can not be applied directly to the grid. First, they have been designed for flat, small-scale architectures and do not cope with the hierarchical architecture of the grid (which implies a hierarchy in terms of communication latency, as previously explained). JuxMem addresses this aspect by implementing a hierarchical, consistency protocol that minimizes data traffic on long-distance, inter-cluster links. Second, traditional protocols have been designed for clusters and parallel machines, and often implicitly assume stable entities (e.g. a home node). However, failures and disconnections are part of grid's specifications. JuxMem implements a hierarchical, home-based protocol for entry consistency, where, to enhance fault tolerance, the critical role of the home is played by the a group (the LDG) at cluster level and by another group (the GDG) at global level. This protocol is described in detail in [7]. When using this protocol, if a client asks for a data access, its request may go through each level of the `data` group hierarchy, in order to be satisfied. For instance, when a client needs to acquire the read-lock, it sends a request to its associated LDG. If the LDG does not already have the read-lock, the LDG sends a request to the GDG. Then the lock is sent back from the GDG to the LDG and finally to the client. In this model, if a client owns a lock, its associated LDG owns the same lock. When the client modifies the data, the modifications are transmitted to the LDG when the client releases the lock, and they can further be transmitted to the GDG either immediately or lated, according to the desired level of fault-tolerance. These aspects are detailed in Section 3.

Finally, the consistency protocol gives priority to writers: a writer only has to wait that previous requests are satisfied, whereas a reader has to wait that no writer is asking for the lock. In its basic version, this strategy can cause readers starvation if two or more writers get alternatively the lock, postponing data access to readers. In order to guarantee that readers evetually access the date, a simple solution consists in setting a limit on the number of times writers actually use this priority.

3 Efficient visualization through concurrent reads and writes

3.1 Proposed enhancement: relaxed reads

We consider a scenario where an observer node reads some shared data for visualization purpose. The reads performed by this node should be efficient and low intrusive. The first idea is to favor access locality by taking advantage of the data copies located on the client node (if any), else fetch a data copy on its associated LDG, ikn tghe same cluster (if available). The second idea is to perform the read operation without acquiring a lock. This particular read operation provides the ability to have concurrent reads and writes as it does not lock the data.

The entry consistency model guarantees that the data is up-to-date only if the associated lock has been acquired. If the associated lock has not been acquired, no guarantees are provided. The approach highlighted in this paper proposes to enable relaxed reads (i.e. without acquiring a lock) for which the user application is able to keep control on the data "*freshness*". This implies that the consistency protocol implementing this extended model respects some bounds on the difference between the version of the data returned by the *rlxread* primitive and the latest version of the data (i.e. the one read after acquiring a lock). Note that this is an extenssion to the entry consistency model: the guarantees of the original model

are preserved under the same conditions (i.e. when using the regular synchronization primitives); besides, new guarantees are provided in some cases where the original model does not guarantee anything. This is detailed in Section 3.4.

Therefore, for each relaxed read operation, the application specifies (as a parameter of the *rlxread* primitive) an upper bound on the difference between the latest version and the one returned by the *rlxread* primitive call.

3.2 Controlling data freshness

Specifying the difference between the latest version and the one returned by the *rlxread* primitive is not a trivial problem. The hierarchical aspect of the data consistency protocol does not provide the ability to retrieve the latest version in one step. For some given data, different LDGs may store different versions indeed. The LDG that owns the lock associated to the data hosts the lastest version of this data while the other ones may host an older version (as LDGs do not necessarily propagate every data update to the GDG). Furthermore, even client nodes attached to the same LDG may host different versions of a given data according to the last time they access this data: the data stored by a client node is only updated when it accesses the data (using the consistency protocol primitives).

To express the difference between the latest version and the version returned by the *rlxread* primitive, we introduce two parameters that take into account the two layers of the hierarchical consistency protocol.

- The D parameter is a constant attached to each piece of data.
- The w parameter (also called *reading window*) is specified for each call to the *rlxread* primitive.

The D constant corresponds to the number of times a LDG can give the exclusive lock to its locally-attached client nodes without sending updates to the GDG. The D parameter is set when the data is allocated by the service. Setting D to a small value forces the LDG to spread updates frequently, offering the possibility to get fresher data from the other LDGs. However, this solution adds an overhead due to frequent GDG updates (releasing the lock, sending update messages, etc.). Alternatively, using a larger value lets the writers perform writes within the same cluster (associated to a given LDG), without wasting time in frequent GDG updates. The counterpart is that the data versions returned by the relaxed read in other LDGs may be a bit older. For instance, if $D = 0$ LDGs have to spread their modifications to the GDG after each release of the exclusive lock by a client. In this case, all LDGs have the same version of the data (the latest). The D parameter has been inspired by the hierarchical synchronization protocol described in [5].

The w parameter is the *reading window*. It is specified for each call of the *rlxread* primitive. It defines an upper bound on the distance between the latest version of the data and the version returned by the relaxed read. Therefore, w must be greater than or equal to D. Considering the smallest value for w (i.e. $w = D$) implies that the relaxed read returns the LDG's version. This solution offers fresher data but it also implies more network traffic when data updates occur frequently (and therefore less efficient relaxed reads). Relaxing the read (i.e. using a greater value for w), enhances the observer access speed by reducing the network traffic but the relaxed read primitive may return older versions of the data.

Note that distances D and w are positive or null and w must be greater than or equal to D. The difference $w - D$ indicates the upper bound between the version of the data stored on the client's LDG and the one returned by the relaxed read primitive on the client's node. For instance, if $D = 3$ then all the LDG can successively give the lock up to 3 times without updating the GDG. If $w = 4$ then the version of the data read by the client is either the LDG's version of the data or the previous version.

For a given data, if a client stores version V_C of the data and if V_{LDG} is the version stored on its LDG, the client can use its own version V_C as long as the following condition is satisfied (α):

$$V_c \geq V_{LDG} - (w - D)$$

This condition is checked by the LDG each time a client node performs a relaxed read.

Efficient visualization relies on the correct tuning of both D and w parameters. Therefore, a smart combination of D and w parameters has to be used depending on the type of application that is monitored and the visualization accuracy that is required.

3.3 Example

Figure 3 illustrates the roles played by w and D within the hierarchical architecture of the protocol. The d data is available in 3 different versions stored on client nodes or LDGs (V_a in one cluster, V_b and V_c in a second cluster). Several clients acquire the lock, write the data, release the lock and send updates to LDG A, increasing the V_a version (1). Every D_d lock releases within LDG A, data updates are sent to the other LDGs (i.e. to the GDG) (2). At the same time, in the second cluster, Client C performs relaxed reads, using a window w as a parameter of each access. A relaxed read request is sent from Client C to LDG B. This request contains 2 pieces of information: 1) the w parameter and 2) V_c: the version of the data owned by client C (3). Depending on the evaluation of the α condition, the LDG B sends back either its V_b version of the data or a message that allows the client to use its own version (4).

3.4 Discussion

The relaxed read proposes an *extension* of the consistency model. Entry consistency is still preserved and guarantees that clients read an up-to-date version of the data, provided they acquire the associated lock. Besides, the entry consistency model is extended by a new feature: some controls are now available when processing a read without acquiring the lock.

Note that setting $D = 0$ and $w = 0$ is not equivalent to the classic sequence of performing a read after getting a read-lock. First, during the relaxed read, the lock can be acquired by another client which can modify the data. This is not allowed in the original entry consistency model. Second, between the moment when the LDG sends the data to the client and the moment when the data is returned by the *rlxread* primitive, new versions can be produced (as the protocol allows writes to continue). Therefore, the user has to know that this approach does not offer *strict* guarantees on data freshness. Providing more guarantees would require that the LDG wait for a client acknowledgment before accepting new updates. Such an approach would however be less efficient. Furthermore, these guarantees are not necessarily needed for the problem of efficient visualization within code-coupling applications.

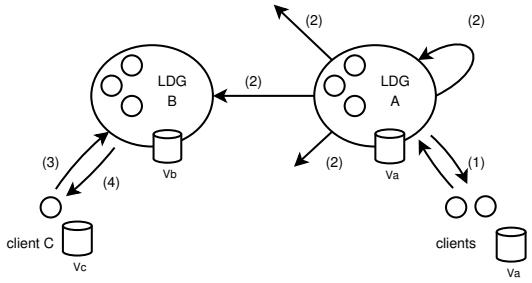

Fig. 3. A relaxed read overview.

4 Evaluation

To perform an experimental evaluation of the proposed protocol, we used the Grid'5000 platform [2], which gathers 9 clusters geographically distributed in several cities of France. These clusters are connected together through the Renater Education and Research National Network (1 Gb/s). For these experiments, we used from 9 to 25 nodes in 3 of these cities (Orsay, Rennes and Toulouse). In each of these 3 clusters, nodes are locally interconnected through a Giga-Ethernet network (1 Gb/s).

Note that we do not use more nodes, as these experiments aim at evaluating the cost of the observation of a single piece of data. Even if a grid application may involve hundreds or thousands of nodes, a single piece of data is rarely accessed by more than a few tens of nodes.

4.1 A visualization scenario

We consider a synthetic code-coupling application running across 2 clusters located in Rennes (Cluster $C1$) and Toulouse (cluster $C2$). As illustrated by Figure 4, Cluster $C1$ runs processes that iteratively write the shared piece of data. We call these processes *writers* thereafter. On Cluster $C2$, some processes (called *readers*) perform read operations. Finally, a third cluster, located in Orsay (Cluster $C3$) is used to run a visualization process, called *observer*.

The experiments are configured as follows: each *writer* performs 50 writes, and each *reader* performs 50 reads concurrently on the same piece of data. At the same time, the *observer* on Figure 4 performs 50 observations of the piece of data. Note that the data is replicated: there is one copy in each cluster. In this example, the size of the LDGs is reduced to 1 (i.e. there is only one copy of the data in each cluster). The 3 LDGs compose the GDG for this data. The main reason that motivated this choice is that fault-tolerance is not the main goal of these experiments. Furthermore, a high replication degree would not be really relevant here, as it has a low impact on read and relaxed read operations.

The goal of these experiments is to evaluate the impact of the consistency model extension upon the visualization process. Therefore, each test is performed twice, by relying on two mechanisms for visualization: 1) using the *acquireRead/release* primitive (called *acquireRead-based visualization* thereafter); 2) using the *rlxread* primitive described in this paper, with no lock synchronization. Finally, we vary the visualization constraints by tuning w and D parameters, and measure how the visualization cost evolves.

In order to evaluate the impact of the data size in our experiments, we use 4 different sizes: 1 KB, 512 KB, 1 MB and 10 MB.

Initially, we use a single *writer* and a single *reader*. Then, in order to vary the communication patterns the number of *writers* and *readers* is gradually increased (up to 9 readers performing $9 * 50$ reads and 9 writers performing $9 * 50$ writes).

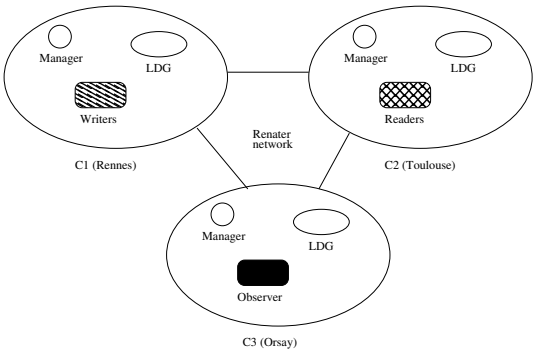

Fig. 4. Experiments configuration

4.2 Results analysis

Benefits of the extension The goal of these first set of experiments is to evaluate the impact of the protocol extension even when parameters D and w are set to 0. As explain in section 3.4, this is not equivalent to reading the data through the *acquireRead* primitive, as no lock is acquired. However, this corresponds to the maximal freshness degree that is allowed by the *rlxread* primitive.

Figure 5 illustrates the impact on the visualization process. The improvement by approximately 80% is mainly explained by the fact that the visualization does not need to wait for a lock. The benefit is growing with the data size: the larger the data, the longer the time to update the data and release the lock. The benefit even reaches 94% for a 10MB piece of data (not displayed on the figure for the sake of readability).

The visualization process is not the only one to take advantage of the *rlxread* primitive. The application itself shows a small improvement as it no longer has to wait for the visualization process to release its lock.

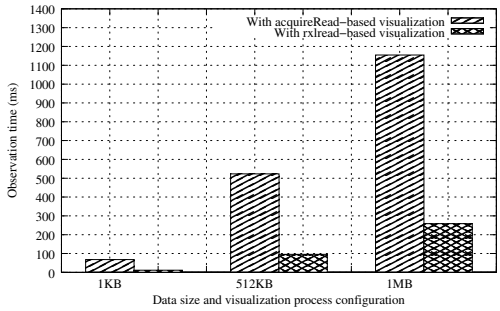

Fig. 5. Improving the observation cost

Figures 6 and 7 respectively illustrate the gain for the writer and the reader. However, the improvement is small: in the case of the *acquireRead-based* visualization, the impact on the application is already low as the read lock is shared between the application reader and the visualization process.

Consequently, the main improvement concerns the visualization process, as shown on Figure 8, which summarizes the benefits for the reader, the writer and the observer.

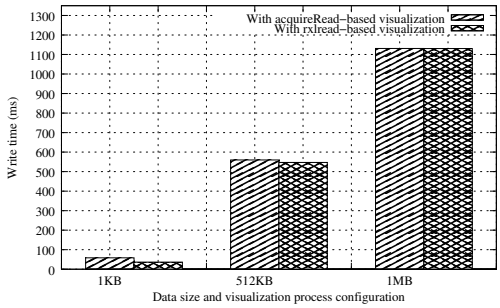

Fig. 6. Impact on the writing cost

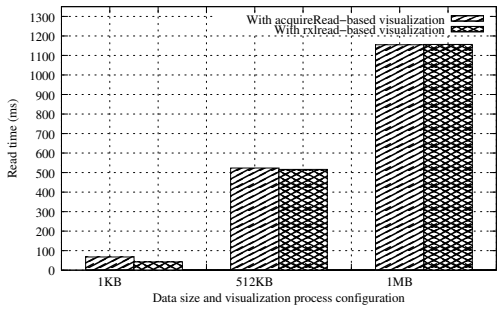

Fig. 7. Impact on the reading cost

Influence of D and w In order to evaluate the impact of the D and w parameters upon the visualization and the application, we have run a second set of experiments, setting $D = 2$ and $w = 3$.

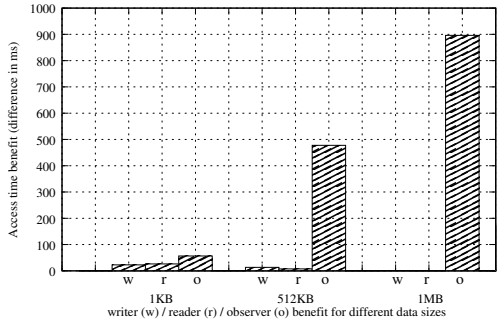

Fig. 8. Overall benefit

According to these values:

- the LDG located in Cluster $C1$ propagates updates at least every 3 writes. Therefore, the degree of fault tolerance is lower here: the latest version of the piece of data may be lost if a failure occurs in Cluster $C1$ between two update propagations. The D parameter provides the ability to tune the tradeoff between fault tolerance and data access performance.
- the LDG in Cluster $C2$ sends back the data to the observer only if the difference between its version and the observer's version is larger than 1 ($w - D$). That allows the observer not to transfer the data each time a new version is available on its LDG. Therefore, it increases performance and decreases network load while providing a sightly less accurate observation.

Figure 9 shows that relaxing the constraints on the data freshness results in an improvement for the visualization (33% for a data size of 1MB). Setting $w = 3$ reduces the probability for the observer to transfer the data. Therefore the improvement increases with the data size. On the other hand, the data returned by the *rlxread* primitive is a little bit less up-to-date.

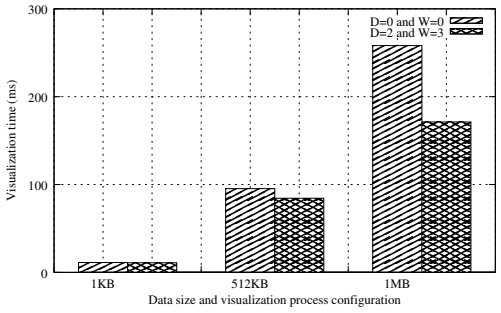

Fig. 9. Improving the observation cost (D=2 W=3)

The impact on the application is really low (almost null), as shown by figures 10 and 11.

Varying communication patterns Finally, the number of writers in Cluster $C1$ and the number of readers in Cluster $C2$ is increased in order to evaluate the impact of the number of readers and writers. Each test is run with both the *acquireRead-based* visualization (using the *acquireRead* primitive) and with the *rlxread-based* visualization. The size of the data is 1 KB. The results presented in Figure 12 show that the latency of the *rlxread* primitive is constant (and lower than in the case of *acquireRead-based* visualization): it does not depend on the number of writers and readers. The *rlxread* primitive only induces communications between

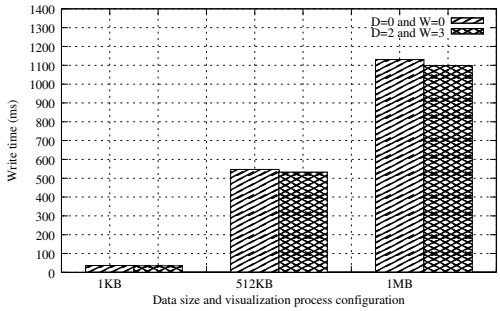

Fig. 10. Impact on the writing cost (D=2 W=3)

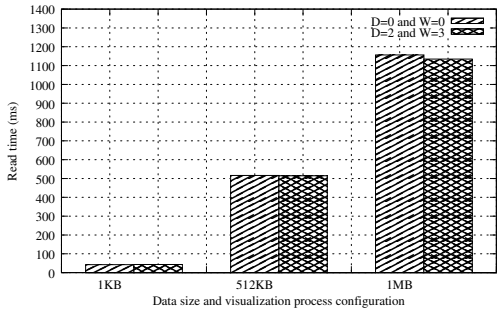

Fig. 11. Impact on the reading cost (D=2 W=3)

the visualization process and its LDG. The latency of the *acquireRead-based* visualization decreases while the number of readers increases: a high number of readers increases the probability that a read lock as already been given in the system. In this case, there is no need to wait for a release, the read lock can be shared by the numerous readers, providing a lower read latency.

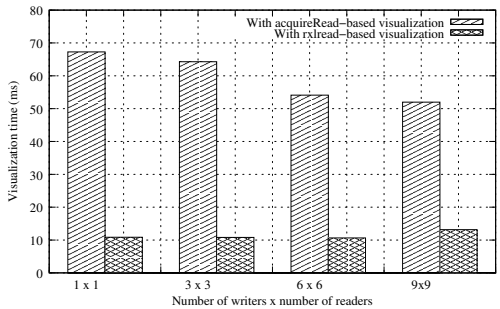

Fig. 12. Impact on the observation cost

However, as the number of writers and readers increases, the average write time grows. As the write lock is exclusive, the probability to wait for a release increases with the number of processes accessing the data using lock synchronization (i.e. except the ones using the *rlxread* primitive). However, Figures 13 and 14 show that using the *rlxread* primitive provides a significant improvement even increasing the number of writers and readers.

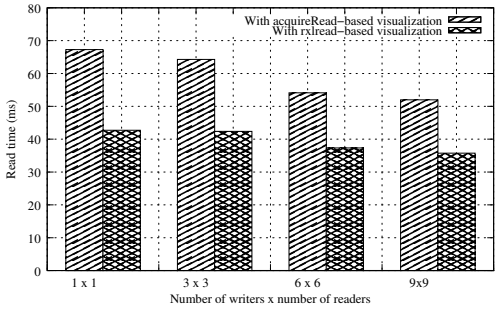

Fig. 13. Impact on the reading cost

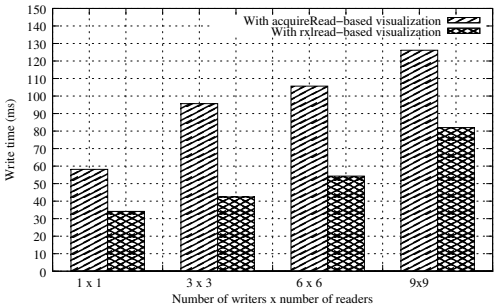

Fig. 14. Impact on the writing cost

As for the *acquireRead-based* visualization, the latency of the read operation decreases while the number of readers increases. There again, the improvement offered by the *rlxread* primitive is significant.

5 Conclusion

Visualization is an useful feature in the context of code-coupling applications, as it may help tuning the application dynamically, while also allowing to get preliminary results, to perform demos, etc. This paper presents and evaluates an extension to the entry consistency model. We introduce the concept of *relaxed read*, that can be performed concurrently to the data accesses performed by the application. This provides the ability to achieve an efficient, and still rather accurate visualization.

Preliminary results obtained on the Grid'5000 testbed show that using the new operation (*rlxread*) is a lot more efficient and slightly less intrusive than using lock-based synchronization (e.g. through the *acquireRead* operation provided by the entry consistency model). The data version returned by the *rlxread* operation is not necessarily the most recent, however its "freshness" can be controlled and should be sufficient for visualization purposes.

We plan to further refine the approach proposed in this paper. A step forward towards transparency and self-adaptivity would consist in considering the w parameter as a hint (e.g. *not accurate*, *accurate* or *very accurate*), according to the needs of the visualization process. JuxMem may then automatically decide what exactly the w parameter should be (which expresses the "freshness degree"), by taking into account parameters like the network load or the data update rate.

References

1. The JXTA project. `http://www.jxta.org`.
2. Projet Grid'5000. `http://www.grid5000.org`.

3. Bill Allcock, Joe Bester, John Bresnahan, Ann L. 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 Comput.*, 28(5):749–771, 2002.
4. Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.
5. Gabriel Antoniu, Luc Bougé, and Mathieu Jan. JuxMem: An adaptive supportive platform for data sharing on the grid. *Scalable Computing: Practice and Experience*, 6(3):45–55, November 2005. Extended version to appear in Kluwer Journal of Supercomputing.
6. Gabriel Antoniu, Loïc Cudennec, and Sébastien Monnet. Extending the entry consistency model to enable efficient visualization for code-coupling grid applications. In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'2006)*, May 2006. To appear.
7. Gabriel Antoniu, Jean-François Deverge, and Sébastien 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. Alessandro Bassi, Micah Beck, Graham Fagg, Terry Moore, James S. Plank, Martin Swany, and Rich Wolski. The internet backplane protocol: A study in resource sharing. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 194, Washington, DC, USA, 2002. IEEE Computer Society.
9. Brian N. Bershad, Mattew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the 38th IEEE International Computer Conference (COMPCON Spring '93)*, pages 528–537, Los Alamitos, CA, February 1993.
10. John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 152–164, Pacific Grove, CA, October 1991.
11. Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Supercomputer Applications*, 15(3):200–222, March 2001.
12. Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.