Extending the entry consistency model to enable efficient visualization for code-coupling grid applications

Gabriel Antoniu, Loïc Cudennec, Sébastien Monnet

Extending the entry consistency model to enable efficient visualization for code-coupling grid applications

Gabriel Antoniu, Loïc Cudennec and Sébastien Monnet
IRISA/INRIA and University of Rennes 1
Campus de Beaulieu, 35042 Rennes, France
{Gabriel.Antoniu,Loic.Cudennec,Sebastien.Monnet}@irisa.fr

Abstract

This paper addresses the problem of efficient visualization of shared data within code coupling grid applications. These applications are structured as a set of distributed, autonomous, weakly-coupled codes. We focus on the case where the codes are able to interact using the abstraction of a shared data space. We propose an efficient visualization scheme by adapting the mechanisms used to maintain the data consistency. We introduce 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. We discuss the benefits and the constraints of the proposed approach.*

1 Introduction

With the growing demand of computing power, grid computing [8] 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. A particular class of such applications relies on the *code-coupling* paradigm: the applications are 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 data transfers between clusters are minimized. In general, these computations can be very long, and it is impractical to wait for the end of the application to see if the results are correct. In order to see 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. Currently, most grid environments use an *explicit data access model*, where data location and transfer across the distributed infrastructure is explicitly handled by the application or the service that needs the data. Such a low-level approach makes data management on grids rather complex. 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 [2]. The idea is to provide a *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 [9, 7, 6] (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 [2] (briefly described 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.

In this paper, we address the problem of efficient data visualization within code-coupling applications designed for grid architectures. In order to efficiently support the presence of a visualization process (that we call *observer*), we extend the *entry consistency* model and propose a 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.

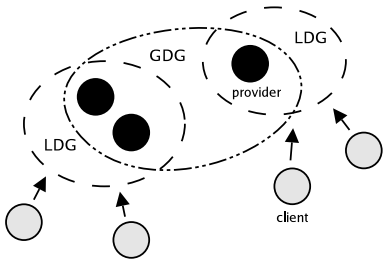

Figure 1. JUXMEM : a hierarchical architecture.

2 JUXMEM : A hierarchical architecture for replicated data

To experiment our approach, we have used the JUXMEM software experimental platform for grid data sharing, described in [2]. From the user's perspective, JUXMEM is a service providing transparent access to persistent, mutable shared data. 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 hierarchically organized, as illustrated on Figure 1: the *Global Data Group (GDG)* gathers all provider nodes holding a replica of the same piece of data. These nodes can 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 a 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, within the strict respect of the consistency model.

To guarantee data consistency, JUXMEM provides a *hierarchical*, *fault-tolerant* consistency protocol that implements the *entry consistency* model. This model was first introduced in the Midway system [6]. 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 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.

JUXMEM implements a hierarchical, home-based protocol for entry consistency, where the role of the home (i.e. the node storing the latest version of the data) is played by the LDG at cluster level and by the GDG at global level. This protocol is described in detail in [5]. 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.

3 Efficient visualization through concurrent reads and writes

3.1 Proposed enhancement: relaxed reads

We consider a scenario where an observer node reads a shared data for visualization purpose. The reads performed by this node should be efficient and low intrusive. The first idea is to enhance locality by using a data copy that is already on the client node or, else, by fetching one from a close node (within the same cluster). The second idea is to introduce a particular read operation (*rlxread*) that can be performed without acquiring a lock, thus allowing read operations and write operations to be concurrently performed.

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 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).

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

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. It corresponds to the number of times a LDG can give the exclusive lock to attached client nodes without sending updates to the GDG. This scheme has been inspired by the hierarchical synchronization protocol described in [3]. The D parameter is set when the data is allocated by the service.**The w parameter**, also called the *reading window*, is specified for each call to 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 larger than or equal to D.

For a given data, if a client has V_C as data version 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.

3.3 Scenario

As a piece of motivation for the proposed extension of the entry consistency protocol, we can imagine a scenario as described in figure 2. We consider 2 clusters (A and B) running a code-coupled application and 1 cluster (C) in charge of the observation of a shared replicated data (3 copies in each cluster). Some clients belonging to cluster A are writing the data that is accessed by readers within the cluster B. From the application side, entry consistency constraints have to be fully preserved in both clusters A and B. This is not the case of cluster C, which only has an observation role. Copies of the data stored on clusters B and C are updated either each D successive writes from cluster A, or when the cluster B acquires the *read-lock*. Depending on the *reading window* w and on the evaluation of the (α) condition, the observer can use its own copy of the data without acquiring the lock, hence without triggering any action for consistency maintenance. Thus, it avoids to degrade the overall application performance.

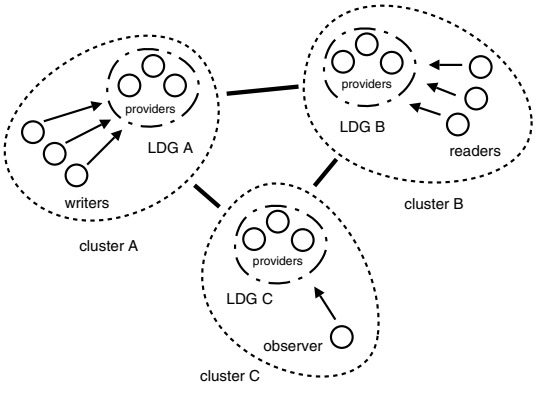

Figure 2. A scenario for relaxed reads.

3.4 Discussion

Efficient visualization relies on the correct tuning of both D and w parameters, according to the type of application that is monitored and the visualization accuracy that is required.

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 GDG updates (releasing the lock, sending update messages, etc.). On the other hand, using a greater value let the writers performing writes within the 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).

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 larger 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 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. 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).

4 Conclusion

In this paper, we present an extension of the entry consistency model that enables efficient relaxed reads concurrently to the application reads and writes. This provides the ability to perform an efficient, and still rather accurate visualization. The *relaxed read* operation is introduced as 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 user can use relaxed reads without locking, while still allowing the data "freshness" to be controlled.

This approach does not offer *strict* guarantees on data freshness. However, the degree of control made available may suffice for visualizations purposes. The proposed approach can also be useful in the context of distributed database applications, where it can be acceptable to process requests on slightly "older" versions of the database.

An extensive evaluation of the efficiency gain obtained thanks to the enhanced protocol is given in [4], based on a visualization scenario inspired by a code-coupling application.

References

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

[2] 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, Nov. 2005. Extended version to appear in Kluwer Journal of Supercomputing.

[3] G. Antoniu, L. Bougé, and S. Lacour. Making a dsm consistency protocol hierarchy-aware: an efficient synchronization scheme. In *Proc. Workshop on Distributed Shared Memory on Clusters (DSM 2003)*, pages 516–523, Tokyo, May 2003. Held in conjunction with the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2003), IEEE TFCC.

[4] G. Antoniu, L. Cudennec, and S. Monnet. Extending the entry consistency model to enable efficient visualization for code-coupling grid applications. Research Report RR-5813, INRIA, IRISA, Rennes, France, January 2006.

[5] 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*, (17), Sept. 2006. To appear.

[6] B. N. Bershad, M. J. Zekauskas, and W. 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, Feb. 1993.

[7] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 152–164, Pacific Grove, CA, Oct. 1991.

[8] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Supercomputer Applications*, 15(3):200–222, Mar. 2001.

[9] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov. 1989.