BUILDING REPLICATED DATABASE SYSTEMS USING DISTRIBUTED SHARED MEMORY

NELSON DUARTE FILHO,

FERNANDO PEDONE

RESUMO

Este artigo apresenta uma abordagem para a construção de sistemas de base de dados replicados utilizando memória compartilhada distribuída. A arquitetura dsmDB, a qual implementa tal proposta, é apresentada. Vantagens e desvantagens da abordagem são elencadas e discutidas.

Palavras-chave: Sistemas de bases de dados replicados, Memória compartilhada distribuída

ABSTRACT

Current trends in main memory capacity and cost indicate that in a few years most performance-critical applications will have all (or most of) their data stored in the main mem- ory of the nodes of a small-size cluster. A few recent research papers have pointed this out and proposed architectures tak- ing advantage of clustered environments aggregating

Nelson Duaret FIlho: Centro de Ciências Computacionais - Universidade Federal do Rio Grande, dmtnldf@furg.br
Fernando Pedone: USI, Switzerland

power- ful processors equipped with large main memories. This position paper proposes yet another approach, which builds on Distributed Shared Memory systems (DSMs) introduced in the early 80's. We introduce the idea of the dsmDB, dis- cuss how its architecture could be organized, and elaborate on some of its algorithms. We conclude the paper with a discussion of some of its advantages and drawbacks.

Keyords: distributed memory, replicated database systems

1 INTRODUCTION

Many current high-end database applications have strin- gent performance and availability requirements. Commer- cial solutions to these applications typically rely upon spe- cialized hardware or proprietary software or both, often in- heriting their design from early centralized databases [10]. Increasing demand from high-end applications together with widespread availability of powerful clusters, built out of com- modity components, have recently led to alternative ap- proaches to implementing efficient and highly available data management systems.

In modern environments, in-memory architectures are more appropriate than on-disk solutions. Main memory capac- ity has been growing at a steady pace, approximately dou- bling every 18 months. There is evidence that most future performance-critical applications will have all (or most of) their data stored in the main memory of the nodes of a small cluster. These trends invalidate many fundamental design decisions of current systems and require a re-evaluation of data structures and algorithms adapted to the new environ- ment.

In-memory database systems (IMDBs) have exploited main memory trends focusing mostly on standalone architectures.

IMDBs provide high throughput and low response time by avoiding disk I/O. IMDBs were originally designed for spe- cific classes of applications (e.g., telecommunication), but have been recently used in more general contexts (e.g., web servers, trading systems, content caches). In most cases, ap- plications are limited by the memory capacity of the server running the IMDB (e.g., [7]).

In a clustered environment, the database can be parti- tioned into segments and stored in the main memory of sev- eral nodes. Applications are then limited by the aggregated memory capacity of the cluster, and not by the capacity of a single node. This approach has been exploited in recent research [1, 4, 10]. In these works, the database is parti- tioned and replicated either at the granularity of rows (e.g., horizontal partitioning) or tables. Queries either have to be broken into subqueries for execution at the appropriate node, or the workload has to be special enough so that the original query can be submitted to the node containing all needed information.

In this paper we argue for a different in-memory approach that does not suffer from the shortcomings of previous pro- posals: there are no limitations on the granularity of parti- tioning and replication, and queries do not have to be broken up for execution. Our architecture is based on Distributed Shared Memory systems (DSMs) of the early 80's. DSMs extend the notion of virtual memory to different nodes of a cluster. Instead of bringing a page from the local disk upon a page-fault, pages are brought from the main mem- ory of remote nodes. Early DSM systems were expensive to implement, as they offered strong consistency. Later pro- posals weakened the consistency semantics, improving per- formance, but required changes in the application programs. Interestingly, semantics that were considered weak in DSMs are enough to implement strong database isolation. In this paper we substantiate this claim with an implementation of one-copy serializability on top of a storage manager that ensures very weak consistency, even for DSM standards.

According to Gray et al.'s terminology [3], the dsmDB be- longs to the RAPS category, Reliable Array of Partitioned Services. RAPS can be implemented in shared disk (e.g., Or- acle's Real Application Cluster [9]) and shared nothing (e.g., MySQL Cluster [5]) environments. While implementations in the former class require specialized hardware, that is not the case for implementations in the latter class, and there- fore for the dsmDB. As pointed out in [3], ideally, data in a RAPS should be automatically repartitioned when nodes are added to the system. In reality, however, this is difficult to achieve—MySQL cluster, for example, requires the en-tire cluster to be shutdown and all nodes to be synchronized offline when adding a new node [2].

As we claim in this pa- per, automatic reconfiguration is relatively simple and can be done on-the-fly in the dsmDB.

In Section 2 we review the concept of DSMs. In Section 3 we show how it can be used in the context of replicated databases. We conclude in Section 4 with a discussion of advantages and disadvantages of the approach.

2 DISTRIBUTED SHARED MEMORY

Distributed shared memory systems (DSMs) extend the notion of virtual memory to different nodes of a cluster [6]. With virtual memory, if an accessed page is not loaded in main memory, a page fault is triggered and the page is brought from disk. DSMs allow pages to be fetched from the main memory of other nodes in the cluster.

The DSM model improves performance since, with cur- rent technology, bringing a page from the main memory of a remote node in a cluster is faster than bringing the page from the local disk. It also allows programs running on different nodes to easily share pages. However, multiple nodes concurrently reading and writing shared pages raises the problem of consistency.

Many consistency criteria have been defined in the liter- ature. The most intuitive one is that a read should always return the last value written. However, in a distributed sys- tem the notion of the last value written is not well defined. Sequential consistency solves this problem by requiring the memory to appear to all nodes as if they were executing on a single multiprogrammed processor.

Although intuitive and well defined, sequential consistency is expensive to implement. To see why, consider that node N wants to write page P. Before the page can be written, N has to invalidate any remote copies of P. The write will be processed once all nodes have replied to N confirming that any future access to P will see the newest value writ- ten. Even though there are mechanisms to keep multiple copies of a page on several nodes, they involve some level of cooperation between nodes, and multiple communication steps until the operations can be processed.

Alternatively, weaker consistency criteria have been pro- posed, which require less synchronization and data move- ment, resulting in better performance [6]. Weaker consis- tency can been obtained, for example, by explicitly specify- ing, through synchronization operators, which parts of the application require strong consistency. Obviously, synchro- nization is no longer transparent, and to work properly some programs may have to be modified.

In the following sections we argue that the DSM model (with weak consistency) is an appropriate paradigm for build- ing high performance and high availability database systems.

3. THE DSM DATABASE APPROACH

In this section we briefly discuss the architecture of the DSM database approach (dsmDB), explain how database consistency criteria can be built on top of weak shared mem- ory guarantees, and present some of the algorithms involved.

3.1 The dsmDB Architecture

We assume a simplified database architecture composed of a Distributed Storage Manager, a Consistency Manager, and a Query Manager (see Figure 1).

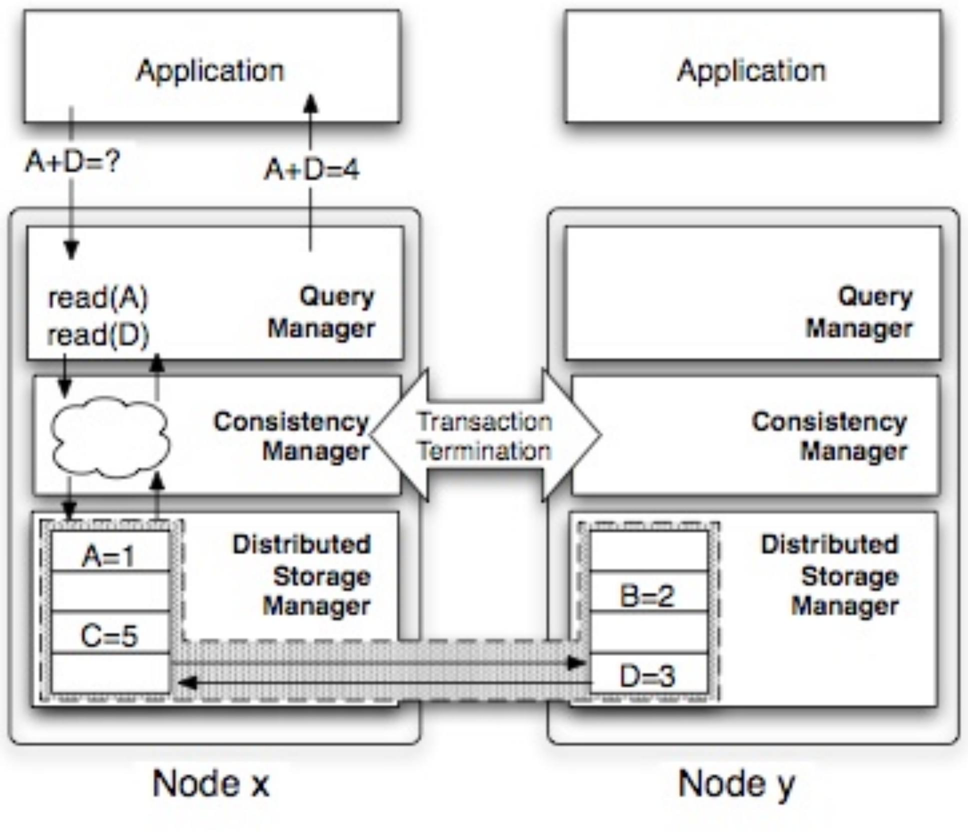

Figura 3: Architecture of the dsmDB

The Storage Manager executes read and write operations on the data, providing transactional access to a virtual stor- age that consists of the complete database space. This il- lusion is implemented by local and remote memory. Storage Managers may synchronize local access to the virtual database space, in which case two transactions executing on the same node would be mutually consistent. However, transactions executing on different nodes are not synchro- nized during their execution by Storage Managers.

The Consistency Manager ensures that the execution is globally consistent according to some correctness criteria. It uses an optimistic mechanism to guarantee global consis- tency, similarly to the deferred update replication model [8]: Transactions execute locally on some node and, as part of the transaction termination protocol, are validated. If the transaction passes the validation phase, it is locally commit- ted by the Storage Manager; if it does not pass validation it is aborted. From the perspective of the Consistency Man- agers, the execution of each transaction is local. In reality, some data may be fetched from remote nodes on demand.

The Query Manager is responsible for receiving high level client requests (e.g., SQL statements), pre-processing them and possibly breaking them up into lower level operations to be executed by the Storage Manager. It receives the re- sults from the Consistency and Storage Managers and passes them to the clients, possibly after some post-processing.

In the example in Figure 1, an application program has submitted a request to compute the sum of items A and D to the Query Manager of node x. This request is parsed and results in two read operations, passed to the Storage Manager. The value of A is stored locally and is available immediately. Reading D triggers an exception since it is not stored locally. After D is fetched from a remote node, the Storage Manager replies to the Query Manager, which com- putes the result and returns it to the application program.

3.2 Distributed Storage Manager

The Distributed Storage Manager provides access to a vir- tual storage that consists of the complete database space, which hereafter we assume to be a set of blocks. There aref our simple operations: read(T, B), write(T, B, V), commit(T), and abort(T), where T is a transaction, B a unique block id, and V a value. These primitives provide weak guarantees:

(Read committed.) Every transaction can only read com- mitted data or its own uncommitted writes.

(Data freshness.) If a node x executes a committed write on block B in isolation and no other node writes B at a later time, then eventually every node that successively reads B will see x's write.

Providing weak consistency (w.r.t. traditional consistency guarantees of Distributed Shared Memory systems) is key to the dsmDB: First, it allows applications to tune the level of global consistency needed through the Consistency Man- ager. Second, weak consistency can be efficiently imple- mented by the Storage Manager.

We now briefly describe one possible implementation of the Distributed Storage Manager.

The virtual storage abstraction provided by a node is im- plemented using its local memory and the memory of remote nodes. At any time, each data block B can be stored by any number of nodes, but it is surely stored by all home nodes of B, determined by a deterministic function with B's id as sole input. Thus, every node keeps locally a set of per- manent blocks, the ones for which it is home, and a set of temporary blocks, blocks that are not permanent and can be discarded at the node's will.

Therefore, we divide the local memory in the Distributed Storage Manager of a node into permanent and temporary. Permanent storage keeps permanent blocks; temporary stor- age keeps temporary blocks and ongoing writes of executing transactions. For durability purposes, permanent storage can be located both in main memory and on disk. Alterna- tively, if blocks have multiple home nodes, then only main memory may suffice to guarantee durability.

Read operations are served immediately from the node's local storage (i.e., from the permanent or temporary storage) or fetched from the block's home node. Write operations are executed locally in temporary storage. At commit time, they may become permanent, temporary, or simply be discarded from the node's memory. To guarantee data freshness, from time to time a node invalidates all blocks in its temporary storage.

The Storage Manager can be optimized in a number of ways. For example, it can discard all write operations of blocks for which it is not the home node. This results in faster response for committing transactions, since fewer op- erations have to be executed, and leaves more space in the memory buffer for ongoing and new transactions.

3.3 Consistency Manager

Storage Managers implement weak consistency semantics. Stronger guarantees can be obtained by a separate mecha- nism implemented by the Consistency Manager. In the fol- lowing we will discuss how One-Copy Serializability (1SR) can be implemented. We adopt an approach similar to the validation process used by the Database State Machine (DBSM) [8].

1SR specifies that any concurrent execution of transac- tions in a possibly replicated setting should be equivalent to a serial execution of the same transactions in a single node. As a consequence, if two conflicting transactions ex- ecute concurrently on different nodes, only one transaction can be allowed to commit. Two transactions conflict if they access the same database block and at least one transaction writes the block.

The Consistency Manager uses an atomic broadcast ab- straction, defined by the primitives broadcast(m) and deliver(m), where m is a message. Every message broadcast by the Con- sistency Manager of a node is delivered by all operational (i.e., nonfaulty) nodes in total order. More formally:

1. (Agreement.) If node x delivers message m, then every nonfaulty node also delivers m.

2. (Total order.) If nodes x and y deliver messages m and m′, they do so in the same order.

During the local execution of T , the Consistency Manager captures its read and write operations, that is, its readset and writeset. Notice that readsets and writestes contain an indication of the data blocks read and written by the trans- actions, not the contents of the blocks. When the commit operation is requested, T’s readsets, writesets, and the val- ues written are broadcast to all nodes.

Upon delivery, each node x proceeds with the validation of T, which checks whether all blocks read by T are still up to date, i.e., they were not written by another transac- tion during T’s execution. This is performed by checking T ’s readset against the writesets of all transactions that ex- ecuted concurrently with T and have already committed.

If T passes validation then it can be committed against the local Storage Manager. At the node where T executed, it suffices for the Consistency Manager to send the commit(T) operation to the Storage Manager and reply to the appli- cation;

at the other nodes, the local Consistency Manager first executes all writes against the local Storage Manager and then commits the transaction.

4. CONCLUDING REMARKS

This short note discusses the dsmDB, a novel approach to building high performance clustered database systems in- spired by DSMs. The dsmDB approach enhances transac- tion processing in two ways: first, it provides in-memory access of executing transactions; second, it allows nodes to reduce the number of updates to be processed when trans- actions commit, improving performance. This last property is achieved by the Storage Managers, which can drop write requests for data blocks they are not responsible for (i.e., the node is not the block's home node).

At the top two layers, Query Manager and Consistency Manager, the dsmDB is similar to the Database State Ma- chine (DBSM) [8]. In fact, the Storage Managers implement the illusion of a fully replicated database. Differently from the DBSM, and other fully replicated database protocols, the dsmDB has more flexibility to handle storage.

We enumerate next a number of reasons we believe the dsmDB approach is promising.

• Location transparency. The dsmDB offers a transpar- ent mechanism to manage memory and disk, either lo- cal or remote. As a consequence, adding nodes to and removing nodes from a cluster can be done on-the-fly, with the resulting performance benefits. It also pro- vides simple system reconfiguration (e.g., changing the unit of partitioning).

• Concurrency control "a` la carte". By providing weak consistency at the storage level, the dsmDB allows an

efficient implementation. Stronger requirements can be built on top of it in a modular way. We have com- mented on how 1SR can be guaranteed in this context. We conjecture that other strong database consistency criteria (e.g., snapshot isolation) could be also imple- mented on top of Storage Managers.

• Simple load balancing. The dsmDB could be com- bined with a load balancer in order to group trans- actions that access common portions of the database on specific nodes, favoring locality. This technique has proved to be quite efficient in similar contexts [4]. The work in [4] assumes that the workload can be divided at the table level, and each table placed on a sepa- rate replica. Since the dsmDB provides location trans- parency, it does not need such an assumption.

• Flexible data partitioning and replication. As another consequence of location transparency, the dsmDB al- lows data to be freely partitioned and replicated. This is an important feature if the database is large but the hot spot portion of the data fits the aggregated main memory of the cluster nodes.

• Incremental recovery. Traditional database systems have to undergo a recovery procedure before becom- ing operational after a crash. This procedure involves operations such as bringing the database state from disk, undoing the effects of unfinished transactions, and making sure that all committed transactions have their effects reflected on the database state. This op- eration can be sped up by bringing the current state from remote nodes, but in any case new transactions can be only accepted once the complete state has been recovered. The dsmDB allows incremental recovery by simply restarting the crashed node with an empty stor- age. New transactions can be accepted immediately and data read by these transactions will be fetched from remote nodes on demand.

We have also tried to anticipate the drawbacks of the dsmDB. So far, we identified the following ones.

• Loss of semantics. Since the virtual database space is implemented by the Storage Managers, after high level operations are translated into lower level block opera- tions by the Query Manager, some information avail- able for sharing data may be lost. In this sense, mov- ing the virtual shared space to higher modules (e.g., the Query Manager) could be wiser. It’s unclear to us though whether the complexity involved in such a design could bring any real benefits.

• Difficulty to re-engineer into existing database engines. We are currently building a prototype of the dsmDB by modifying an existing database (Berkeley DB). De- spite the relatively simplicity of the Storage Manager, modifying a storage engine to integrate the ideas de- scribed in this paper turned out to be more complex than we initially expected. One reason is that although it is easy to identify the portions of the database that handle storage, dependencies across layers make it dif- ficult to introduce the needed modifications.

5. REFERENCES

[1] L. Camargos, F. Pedone, and M. Wieloch. Sprint: A middleware for highperformance transaction processing. In Eurosys, 2006.

[2] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: The gaps between theory and practice. In SIGMOD ’07: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, 2006.

[3] B. Devlin, J. Gray, B. Laing, and G. Spix. Scalability terminology: Farms, clones, partitions, and packs: Racs and raps. Technical report, Microsoft, 1999.

[4] S. Elnikety, S. Dropsho, and W. Zwaenepoel. Tashkent+: Memory-aware load balancing and update filtering in replicated databases. In Eurosys, 2006.

[5] Mysql cluster architecture overview, April 2004. Online Technical White Paper.

[6] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. IEEE Computer, 24(8):52–60, August 1991.

[7] Oracle TimesTen products and technologies, February 2006. Online White Paper.

[8] F. Pedone. The Database State Machine and Group Communication Issues. PhD thesis, E´colePolytechnique F´ed´erale de Lausanne, Switzerland,1999. Number 2090.

[9] Oracle real application cluster 11g, April 2006. Online White Paper. [10] M. Stonebraker, S. Madden, D. Abbadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it’s time for a complete rewrite). In VLDB, 2006.