Load Balancing of Autonomous Applications and Databases in a Cluster System

STÉPHANE GANÇARSKI
LIP6, University Paris 6, France

HUBERT NAACKE
LIP6, University Paris 6, France

PATRICK VALDURIEZ
LIP6, University Paris 6, France

Abstract

Clusters of PC servers make new businesses like Application Service Provider (ASP) economically viable. In the ASP context, hosted applications and databases can be update-intensive and must remain autonomous so they can be subject to definition changes to accommodate customer requirements. In this paper, we propose a new solution for load balancing of autonomous applications and databases. Our solution is similar to Distributed Shared Memory in that it provides a shared address space to applications with distributed and replicated databases. The main idea is to allow the system administrator to control the tradeoff between consistency and performance when placing applications and databases onto cluster nodes. Application requirements are captured through rules stored in a shared catalog. They are used (at run time) to allocate cluster nodes to user requests in a way that optimizes load balancing. They are also used with the database logs to detect and repair inconsistency problems.

Keywords

database, cluster architecture, transaction processing, load balancing, replication, consistency

1 Introduction

Clusters of PC servers now provide a cheap alternative to tightly-coupled multiprocessors such as Symmetric Multiprocessor (SMP) or Non Uniform Memory Architecture (NUMA). They make new businesses like Application Service

Provider (ASP) economically viable. In the ASP model, customers' applications and databases (including data and DBMS) are hosted at the provider site and need be available, typically through the Internet, as efficiently as if they were local to the customer site. Thus, the challenge for a provider is to fully exploit the cluster's parallelism and load balancing capabilities to obtain a good cost/performance ratio. The typical solution to obtain good load balancing in cluster architectures is to replicate applications and data at different nodes so that users can be served by any of the nodes depending on the current load. This provides also high-availability since, in the event of a node failure, other nodes can still do the work. This solution has been successfully used by Web sites such as search engines using high-volume server farms (e.g., Google). However, Web sites are typically read-intensive which makes it easier to exploit parallelism.

In the ASP context, the problem is far more difficult. First, applications can be update-intensive. Second, applications and databases must remain autonomous so they can be subject to definition changes to accommodate customer requirements. Replicating databases at several nodes, so they can be accessed by different users through the same or different applications in parallel, can create consistency problems [7, 6]. For instance, two users at different nodes could generate conflicting updates to the same data, thereby producing an inconsistent database. This is because consistency control is done at each node through its local DBMS. There are two main solutions readily available to enforce global consistency. One is to use a transaction processing monitor to control the access to replicated data. However, this requires significant rewriting of the applications and may hurt transaction throughput. A more efficient solution is to use a parallel DBMS such as Oracle Rapid Application Cluster or DB2 Parallel Edition. Parallel DBMS typically provide a shared disk abstraction to the applications [9] so that parallelism can be automatically inferred. But this requires heavy migration to the parallel DBMS and hurts database autonomy.

Ideally, applications and databases should remain unchanged when moved to the provider site's cluster. In this paper, we propose a new solution for load balancing of autonomous applications and databases which addresses this requirement. This work is done in the context of the Leg@Net project[1] sponsored by the RNTL between LIP6, Prologue Software and ASPLine. Our solution is similar to Distributed Shared Memory (DSM) [3] in that it provides a shared address space to applications with distributed and replicated databases. The main idea is to allow the system administrator to control the database consistency/performance tradeoff when placing applications and databases onto cluster nodes. Databases and applications can be replicated at multiple nodes to obtain good load balancing. Application requirements are captured (at compile time) through rules stored in a shared catalog used (at run time) to allocate cluster nodes to user requests. Depending on the users' requirements, we can control database consistency at the

[1] www.industrie.gouv.fr/rntl/AAP2001/Fiches_Resume/LEG@NET.htm

cluster level. For instance, if an application is read-only or the required consistency is weak, then it is easy to execute multiple requests in parallel at different nodes. If, instead, an application is more update-intensive and requires strong consistency, then an extreme solution is to run it at a single node and trade performance for consistency. Between these two extreme application scenarios, there are intermediate cases where copy consistency can be violated. Our solution exploits the database logs and the consistency rules to perform copy reconciliation.

This paper is organized as follows. Section 2 introduces our cluster architecture and discusses the various options for load balancing of applications and databases. Section 3 describes the way we can capture and exploit consistency rules about applications to obtain load balancing. Section 4 describes our execution model which uses these rules to perform load balancing and manage global consistency.

2 Cluster architecture

In this section, we introduce the conceptual architecture for processing user requests in our cluster system and discuss the main options available for placing applications, DBMS and databases in the system.

Figure 1 shows the main elements involved for processing a user request coming, for instance, from the Internet.

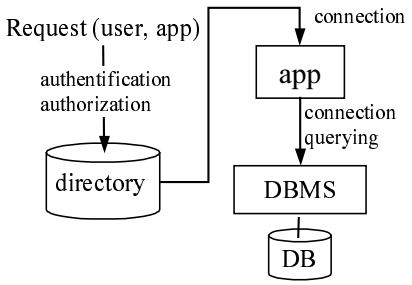

Figure 1: Cluster conceptual architecture

First, the user is authenticated and authorized using the directory which captures information about users and applications. If successful, the user gets a connection to the application (possibly after instantiation) which can then connect to the DBMS and issue queries for retrieving and updating database data.

We consider a cluster system with similar nodes, each having one or more processors, main memory (RAM) and disk. There are various ways to organize the elements of the conceptual architecture in the cluster system. For simplicity,

we ignore the directory element which is used to route requests to nodes. We present three main organizations to obtain parallelism and discuss their consistency/performance trade-offs. The first and simplest one is client-server DBMS connection (see Figure 2) whereby a client application at one node connects to a remote DBMS at another node (where the same application can also run). The client node only needs connectivity software to access the remote DBMS. This organization allows for application parallelism and yields strong consistency, since all database accesses go through the same DBMS. However, remote database access is much less efficient than local access. Furthermore, there is no database parallelism. This organization is well-suited when strong consistency is important or applications are not data-intensive.

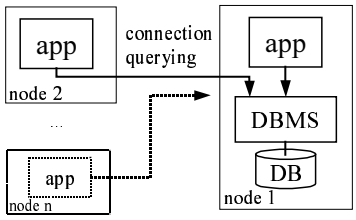

Figure 2: Client-server DBMS connection

The second organization is peer-to-peer DBMS connection (see Figure 3) whereby a client application at one node connects to a local DBMS which transparently accesses the same DBMS at another node using a distributed database capability. This organization also allows for application parallelism and yields strong consistency, since all database accesses go through one database which is under the control of the distributed DBMS. Furthermore, the DBMS at the client node can cache data and thus reduce the overhead of remote access. This organization is well-suited when strong consistency is important or applications have strong locality of reference. However, remote database access can still be quite inefficient and there is no database parallelism.

The third organization is replicated database (see Figure 4) whereby the database is replicated across several nodes. We use multi-master (or symmetric) replication [7] whereby each (master) node can perform updates to the copy it holds. When a node updates a copy, it also updates (or synchronizes) the other copies in either synchronous mode (within the same transaction) or asynchronous mode (in separate transactions). There is both application and database access parallelism, and no need for remote access between nodes. Thus, this is much more efficient than the previous organizations. However, conflicting updates to the database from two different nodes can yield to consistency problems (e.g. the same data get different

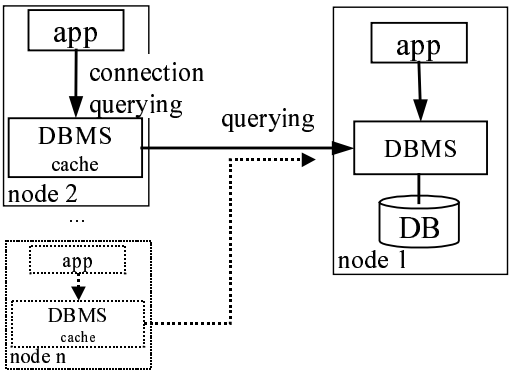

Figure 3: Peer-to-peer DBMS connection

values in different copies) which must then be detected and corrected. Correction can be based on priority assigned to one of the copies, called reference copy. This organization is well-suited when consistency can be relaxed or applications are read-intensive.

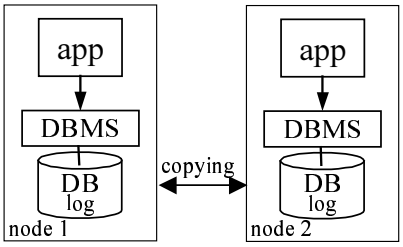

Figure 4: Replicated database

These three organizations are interesting alternatives which can be combined to better control the consistency/performance trade-off of various applications and increase load balancing. For instance, an application at one node could do client-server connection to one or more replicated databases, the choice of the replicated database being made depending on the load.

3 Trading consistency for load balancing

Applications update the database through transactions. In this section, we discuss transaction parallelism and show how we can capture and exploit consistency rules about applications in order to obtain transaction parallelism.

3.1 Transaction parallelism

The replicated database organization may increase transaction parallelism. For simplicity, we focus on inter-transaction parallelism, whereby transactions updating the same database are dynamically allocated to different master nodes. The choice of a node for a given transaction depends on the current load of the cluster; intuitively, a transaction should be sent to one of the least loaded nodes. But it also depends on the application requirements in terms of consistency. If the application requires strong consistency, client-server or peer-to-peer organizations are better. If the application requires weaker consistency, it is possible to run transactions in parallel at different master nodes. Even with the replicated database organization, strong consistency can be achieved by simply allocating transactions to the reference master node, which always holds the up-to-date consistent copy.

3.2 Using consistency rules

Application consistency requirements are expressed in terms of rules. Examples of consistency rules are data-independency between transactions, integrity constraints [1], access control rules, etc. They may be stored explicitly by the system administrator or inferred from the DBMS catalogs. They are primarily used by the system administrator to place and replicate data in the cluster, similar to parallel DBMS [5, 9]. They are also used by the system to decide at which nodes and under which conditions a transaction can be executed.

To illustrate how we can use consistency rules, let us consider two transactions T1 and T2 on the same database executed in parallel at master nodes N1 and N2, respectively. There are three interesting cases.

First, assume T1 and T2 are data independent, i.e. data accessed by T1 and T2 are disjoint sets. In this case, copy consistency can be simply achieved by propagating the changes made by T1 to N2 and those made by T2 to N1. Data independency between transactions may be statically stored in the directory, either explicitly stated or inferred at compile-time. It may also be deduced at run-time, according to the user profile (e.g. access authorization) or by analyzing database logs. The same policy can be applied when T1 and T2 are not data independent but are commutative, following the idea of multi-level transactions [11]: propagating T1 to N2 after the execution of T2 leads to the same database state as propagating T2 to N2 after the execution of T1. However, commutativity between non data-

independent transactions should be explicitly stated, since it seems difficult to infer it from both static and dynamic knowledge.

The second case is when T1 and T2 perform conflicting non-commutative writes, such as booking the same hotel room. In this case, conflict resolution must take into account two kinds of rules, namely priority and resolution mode. Priority expresses which result (of T1 or T2) should be kept. For instance, the changes made by the conflicting transaction with the highest priority should be propagated to the other sites. Priority may be inferred from static rules, such as user profile or the fact that a node is a reference master for the copy. It may also take into account dynamic features, such as transaction timestamps and even use random ranking in case of undecidability. The resolution mode defines the policy to restore consistency. When the changes made by a transaction with lower priority cannot be propagated, the system must know what to do. For instance, it may notify the user that the update is being overwritten. It may also compensate the whole transaction, e.g. canceling the room reservation may lead to cancel the car reservation which is now useless. It may also retry part of the transaction, e.g. to get a room in a separate transaction. In this case, the system must be able to notify the user of which part of its transaction was successfully performed (here the car reservation) and which part has to be retried (here the room reservation). To this end, the system analyzes the database logs and detects that only the room reservation has been overwritten. The system may also, in some cases, automatically retry any (partially) aborted or overwritten transaction.

The third case deals with read/write consistency. Compensating a transaction T in case of conflict resolution can effect another transaction T' reading data written by T, which become dirty. For the sake of consistency, T' should in turn be aborted or compensated. In some cases, this cascading rollback can be avoided using application knowledge. For instance if T and T' are both booking rooms, the system can infer that, although T and T' may access the same data, the effects of compensating T (i.e. canceling a reservation) cannot lead to compensate T'. Consistency rules may also take into account the conflict resolution frequency. For instance, transaction T requires consistent data, but can operate on data that was consistent less than 10 minutes ago since the system knows that the database state could not vary sufficiently in 10 minutes to change the behavior of T.

3.3 Managing consistency rules

Consistency rules (CR) are stored in the *CR base* (see Figure 5), and maintained by the system administrator. They are expressed in a declarative language. Implicit rules refer to data already maintained by the system (e.g. users authorizations). Hence, they include queries sent to the database catalog to retrieve the data.

Incoming transactions are managed by the execution manager. It retrieves consistency rules associated to a given transaction and defines a run-time policy for

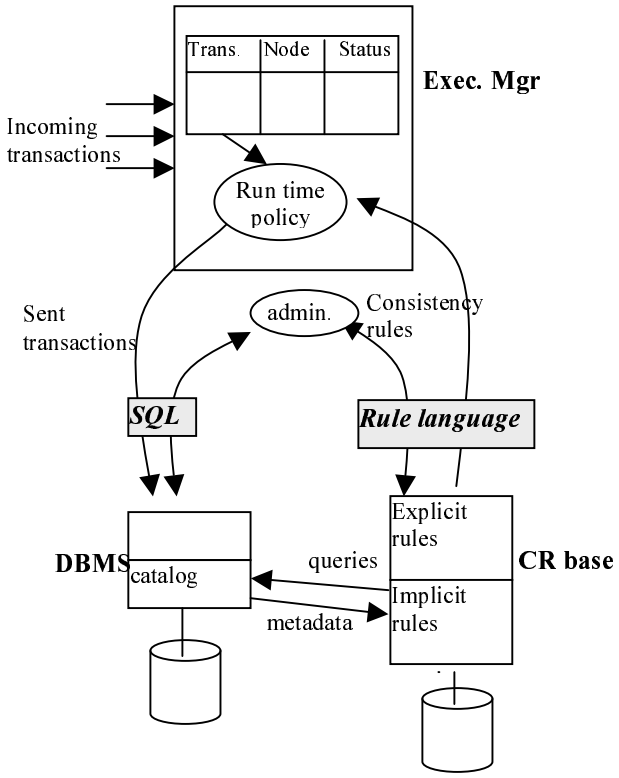

Figure 5: Consistency rule management

the transaction. The run-time policy controls the execution of the transaction on a given node, at the required level of consistency.

4 Execution model

In this section, we present the execution model for the cluster architecture. The objective is to increase load balancing following the consistency rules. The problem can be reduced as follows: given the cluster's state (nodes load, running transactions, etc.), the cluster's data placement, and a transaction T with a number of consistency rules, choose the node(s) where T should be executed with minimal cost (including the cost of synchronizing replicas). Choosing the optimal node is

done in two steps: (1) choosing the data access method and the synchronization mode, and (2) choosing the optimal node among all the nodes that support the data access method chosen at step (1).

4.1 Choice of data access method

Choosing the data access method involves deciding whether the transaction accesses reference data (local or remote) or replicated data based on consistency rules. We have the following possible methods:

Access to reference data. This method is necessary to preserve data consistency when several transactions update the same data. It serializes updates, which has the negative effect of increasing contention as the degree of concurrency increases.

Access to synchronous replicated data. This method accesses a replica which is not the reference copy, while preserving copy consistency. Synchronization is done immediately at transaction commit time, within the transaction. Conflicts occurring during synchronization must be resolved to yield a consistent state of all copies. If conflict resolution is not allowed by the transaction, then it must be aborted and restarted until there is no conflict. This method is appropriate for heavy transactions (to avoid overloading the reference master node) and when the chance of conflict is low. Synchronization is not necessary if the absence of conflicts can be inferred from the transaction log or consistency rules.

Access to asynchronous replicated data. This method accesses also a replica which is not the reference copy, but performs copy synchronization after the transaction commitment at specified times. If a conflict occurs during synchronization, since the transaction has been committed, the only solutions are to either compensate the transaction or notify the user or administrator. This method is appropriate when T is disjoint from all transactions that accessed reference data since the last synchronization.

Algorithm. The choice of the candidate access methods (CAM) is made as follows (RD stands for reference data, SD for synchronous replicated data, and AD for asynchronous replicated data):

if T is compensatable or probability of conflict = 0
or weak consistency is required
then CAM = {RD, SD, AD}
else if T is commutable or conflicts are resolvable
  then CAM = {RD, SD}
  else CAM = {RD}
endif
endif

When there are several candidate access methods, the choice is made at the next step, when choosing the best node.

4.2 Choice of optimal node

This is an optimization problem. The initial search space is the set of all the nodes which support the access methods chosen at step (1). The execution cost of T at each candidate node must be estimated based on a cost function which takes into account the cluster load, the delays due to the serialization of concurrent transactions and possibly the cost of synchronizing replicas. Thus the choice of the optimal node depends on the accuracy of the cost function. Alternatively, the system administrator can overload cost functions with specific ones in the directory for important transactions.

4.3 Transaction execution

After steps (1) and (2), the execution manager sends the transaction to the chosen node with the chosen access method (see Figure 5) and registers the transaction in its table. Periodically (or on user's demand), it performs copy synchronization. To get enough details about the current state of copies, the execution manager reads the DBMS log (which keeps the history of update operations) of all the nodes that have executed transactions since the last synchronization. Then, it resolves conflicts based on priority, timestamps, or user-defined reconciliation rules. If automatic conflict resolution is not possible, the execution manager fires a notification alert to the user or the administrator with details about conflicting operations.

5 Conclusion

In this paper, we proposed a new solution for load balancing of autonomous applications and databases in the context of ASP. Our solution is similar to Distributed Shared Memory in that it provides a shared address space to applications with distributed and replicated databases. The main idea is to allow the system administrator to control the consistency/performance tradeoff when placing applications and databases onto cluster nodes.

We studied three organizations for placing applications and databases in a cluster system: client-server, peer-to-peer, and (multi-master) replicated database. These three organizations are interesting alternatives which can be combined to better control the consistency/performance trade-off of various applications and increase load balancing.

Application requirements are captured through rules stored in a shared catalog (used at run time). They are used to choose the best organization for applications and databases, to prevent, detect and repair copy inconsistency; and optimize load

balancing by estimating the processing cost of transactions, including conflict resolution cost.

In the near future, we plan to implement the proposed solution on LIP6's cluster architecture running Linux and Oracle 8i. Then, we will experiment with application samples provided by our partners in Leg@Net in our cluster. We will also develop a simulation model, calibrated with our implementation, to study how our solution will scale-up to very large cluster configurations.

Other interesting projects deal with DSM for data management in cluster architectures. The PowerDB project at ETH Zurich deals with the coordination of the cluster nodes in order to provide a uniform and consistent view to the clients. Its solution fits well for some kinds of applications, such as XML document management [2] or read-intensive OLAP queries [8]. However, it does not address the problem of seamless integration of legacy applications. The Trapp project at Stanford University [4] addresses the problem of precision/performance trade-off. It is close to us since it shows that performance gains can often be achieved if precision requirements (exact consistency of replicated data) can be relaxed. However, the focus is on numeric computation of aggregation queries. The GMS project at Washington university uses global information to globally optimize page replacement and prefetching decisions over the cluster [10]. However, it mainly addresses system-level or Internet applications (such as the Porcupine mail server). In summary, none of them addresses the problems of leaving databases and applications autonomous and unchanged as in our research.

References

[1] DOUCET, A., GANÇARSKI, S., LEÓN, C., AND RUKOZ, M. Checking integrity constraints in multidatabase systems with nested transactions. In *Int. Conf. On Cooperative Information Systems (COOPIS)* (Trento (Italy), 2001).

[2] GRABS, T., BÖHM, K., AND SCHEK, H.-J. Scalable distributed query and update service implementations for XML document elements. In *IEEE RIDE Wshp. on Doc. Mgt. for Data Intensive Business and Scientific Applications.* (2001).

[3] HILL, M., LARUS, J., REINHARDT, S., AND WOOD, D. Cooperative shared memory: Software and hardware for scalable multiprocessors. *ACM TOCS 11*, 4 (1993).

[4] OLSTON, C., AND WIDOM, J. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *VLDB International Conference* (2000).

[5] ÖZSU, T., AND VALDURIEZ, P. Distributed and parallel database systems - technology and current state-of-the-art. *ACM Computing Surveys 28*, 1 (1996).

[6] ÖZSU, T., AND VALDURIEZ, P. *Principles of Distributed Database Systems (2nd edition)*. Prentice Hall, 1999.

[7] PACITTI, E., AND VALDURIEZ, P. Replicated databases: concepts, architectures and techniques. *Network and Information Systems Journal 1*, 3 (1998).

[8] RÖHM, U., BÖHM, K., AND SCHEK, H.-J. Cache-aware query routing in a cluster of databases. In *IEEE Conference on Data Engineering* (2001).

[9] VALDURIEZ, P. Parallel database systems: open problems and new issues. *Int. Journal on Distributed and Parallel Databases 1*, 2 (1993).

[10] VOELKER, G., ANDERSON, E., KIMBREL, T., FEELEY, M., CHASE, J., KARLIN, A., AND LEVY, H. Implementing cooperative prefetching and caching in a global memory system. In *ACM Sigmetrics Conference on Performance Measurement, Modeling, and Evaluation* (1998).

[11] WEIKUM, G. Principles and realization strategies of multilevel transaction management. *ACM Transaction on Database System 16*, 1 (1991).

Stéphane Gançarski is Assistant Professor of Computer Science at University Pierre et Marie Curie (Paris 6), and researcher at LIP6 lab. E-mail: Stephane.Gancarski@lip6.fr

Hubert Naacke is Assistant Professor of Computer Science at University Pierre et Marie Curie (Paris 6), and researcher at LIP6 lab. E-mail: Hubert.Naacke@lip6.fr

Patrick Valduriez is currently a Professor of Computer Science at University Pierre et Marie Curie (Paris 6), on leave from INRIA, the national research center for computer science in France. He is the author or co-author of over 100 technical papers and several books in computer science, among which "Principles of Distributed Database Systems" published by Prentice Hall in 1991 and 1999 (second edition). E-mail : Patrick.Valduriez@lip6.fr