DTR: Distributed Transaction Routing in a Large Scale Network

Idrissa Sarr, Hubert Naacke, and Stéphane Gançarski

University Paris 6, LIP6 Lab, France
FirstName.LastName@lip6.fr

Abstract. Grid systems provide access to huge storage and computing resources at large scale. While they have been mainly dedicated to scientific computing for years, grids are now considered as a viable solution for hosting data-intensive applications. To this end, databases are replicated over the grid in order to achieve high availability and fast transaction processing thanks to parallelism. However, achieving both fast and consistent data access on such architectures is challenging at many points. In particular, centralized control is prohibited because of its vulnerability and lack of efficiency at large scale. In this article, we propose a novel solution for the distributed control of transaction routing in a large scale network. We leverage a cluster-oriented routing solution with a fully distributed approach that uses a large scale distributed directory to handle routing metadata. Moreover, we demonstrate the feasibility of our implementation through experimentation: results expose linear scale-up, and transaction routing time is fast enough to make our solution eligible for update intensive applications such as world wide online booking.

1 Introduction

Grid systems use a distributed approach to deal with heterogeneous resources, high autonomy and large-scale distribution. Thus, they present a real interest to important areas of enterprise information systems. For instance, Global Distribution Systems (GDS) like Amadeus [2], Sabre [16], Galileo [6] manage a huge amount of data for airline companies, hotels and travel agencies. For instance, Amadeus system information manages data for 62.000 travel agencies, 734 airline companies, 61560 hotels and covers more than 207 countries. The challenge for these systems is to ensure data availability and consistency in order to deal with fast updates. To solve this problem, these systems use expensive parallel servers. Furthermore data is located on single site, which limits scalability and availability. Mapping these GDS systems to a grid allows to overcome these limitations at a rather low cost. In such architecture, the data accessed by the GDS will be stored by the participants (hotels, airline companies, *etc.*) and can be shared. Thus, the data is distributed and parallel executions can be done so that load balancing is achieved.
In order to improve data availability, data is replicated and transactions are routed to the replicas. However, the mutual consistency can be compromised,

because of concurrent updates. Let us illustrate the problem with a concurrent update. Assume that we have two replicas R_1 and R_2 and we have two transactions T_1 and T_2 which are sent respectively by two applications (or travel agency) A_1 and A_2. Each transaction aims for making a reservation operation in the same flight (AF709) of Air France airline company. Assuming that only one seat is available and T_1 are routed to R_1 and T_2 to R_2, then the simultaneous execution of T_1 and T_2 produce a data inconsistency: one of travel agencies sales a non-existing seat. Another point is that some queries can be executed at a node which misses the latest updates. For instance, a request which computes the number of passengers of a flight can be executed in a (few loaded) stale node. To this end, two conditions must be satisfied: *(i)* staleness of the node (expressed in number of missing updates) does not exceed the quantity of overbooking the company is allowed and *(ii)* the request does not perform updates (for sake of consistency). In other words, controlling the freshness of nodes for executing read-only queries can help in improving performances through a better load balancing. Many solutions have been proposed in distributed systems for managing replicas [13], [11], [9], [8], [5] and [12]. Some of them include freshness control [15], [7], [10] and [1]. We base our work on the Leg@net approach [7], since it offers update anywhere and freshness control features and does not require any modification of the underlying DBMS nor of the application source code.

However, cluster systems deals with homogeneous nodes and are not suitable for systems which have heterogeneous and independent entities such as GDS. In order to make Leg@net system suitable to GDS applications, it is necessary to modify its architecture such that it becomes fully distributed on grid system. To reach this goal, the router and the metadata will be replicated at many sites of the grid.
In this paper, we aim to design a new system relying on the Leg@net principles to deal with transaction routing at a large-scale. Our main contributions are:

- A fully distributed transaction routing model which deals with update-intensive application. Our middleware, ensures data distribution transparency and does not require synchronization (through 2PC or group communication) while updating data.
- A large-scale distributed directory for metadata, highly available and easy to access. It enables to keep data consistency with few communications messages between routers.
- Experimental evaluation of our approach that show its feasability.

The rest of this paper is organized as follows. We first present in Section 2 the global system architecture, the replication and freshness model. Section 3 describes our algorithm for transaction routing with freshness control. Section 4 presents experimental evaluations of our system and Section 5 concludes.

2 System Architecture and Model

In this section we describe how our system architecture and model are defined. We first present the global architecture which is needed for understanding our

solution. Then we describe the replication and freshness model used in order to preserve global consistency.

2.1 Architecture

The global architecture of our system is depicted on Figure 1. Transactions are sent by applications to any Transaction Manager (*TM*). TM uses metadata stored in a shared directory implemented into JuxMem [3], to route the transaction for execution on a data node (N^i) while maintaining global consistency. JuxMem provides the abstraction of a shared memory over a distributed grid infrastructure, by transparently handling consistency in a fault-tolerant way. Data nodes use a local relational DBMS to store data and performs local execution of transactions sent by TMs.

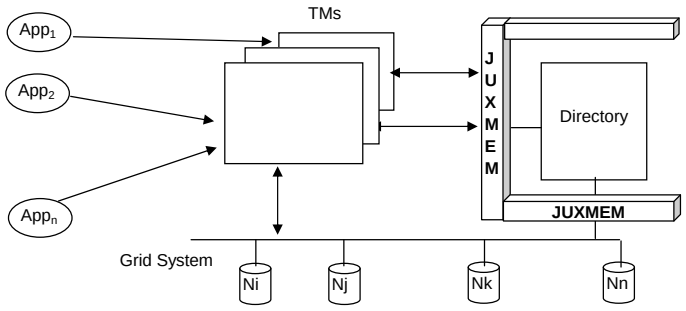

Fig. 1. Global architecture

2.2 Replication and Freshness Model

We assume a single database composed of relations $R^1, R^2...R^n$ that is fully replicated at nodes $N_1, N_2...N_m$. The local copy of R^i at node N_j is denoted by R^i_j and is managed by the local DBMS. We use a lazy multi-master (or update everywhere) replication scheme. Each node can be updated by any incoming transaction and is called the initial node of the transaction. Other nodes are later refreshed by propagating the transaction through refresh transactions. We distinguish between three kinds of transactions:

- Update transactions are composed of one or several SQL statements which update the database.
- Refresh transactions are used to propagate update transactions to the other nodes for refreshment. They can be seen as "replaying" an update transaction on another node than the initial one. Refresh transactions ar distinguished from update transactions by memorizing in the shared directory, for each data node, the transactions already routed to that node.

– Queries are read-only transactions, and thus need not be refreshed.

Let us note that, because we assume a single replicated database, we do not need to deal with distributed transactions, *i.e.* , each incoming transaction can be entirely executed at a single node.

2.3 Freshness Model and Metadata

Every transaction (update, refresh or query) reads a set of relations, every update and refresh transaction writes a set of relation. This information can be obtained by parsing transactions code, and is stored into the shared directory.

Queries may access to stale data, provided it is controlled by applications. To this end, application can associate a *tolerated staleness* with queries. Staleness can be defined through various measures [10]. In this paper, we only consider one measure, defined as the number of updated tuples, for each relation R_i accessed by a transaction T. More precisely, the staleness of R_j^i is equal to the maximum number of tuples of R^i already updated on any node but not yet updated on N_j. The tolerated staleness of a query is thus, for each relation read-accessed by the query, the maximum number of updates that can be missing on a node to be read by the query. Tolerated staleness reflects the freshness level a query requires to be executed on a given node. For instance, if the query requires perfectly fresh data, its tolerated staleness is equal to zero. This information is also stored in the shared directory. Note that, for consistency reasons, update (and thus refresh) transactions must read perfectly fresh data, thus their tolerated staleness is always equal to zero for every relation they access.

To compute the staleness of a relation copy R_j^i, we store in the shared directory, for each update transaction T writing R^i, the maximum number of tuples T may update on R^i. We also store the system global state, *i.e.* for each update transaction, the nodes where it has been already executed. This allows for computing a lower bound of R_j^i's staleness, which is lower or equal to the actual staleness. This guarantees that, when executing a query with tolerated staleness ts on a node with an estimated staleness $s \leq ts$, then the actual freshness of the node is sufficient to fullfil the query requirement.

The shared directory also stores, for each transaction T, the estimated time of processing T, which is a moving average based on previous executions of T. It is initialized by a default value obtained by running T on an unloaded node. It serves at computing the cost function used for transaction routing and load balancing (see Section 3.1).

2.4 Global Consistency

In a lazy multi-master replicated database, the mutual consistency of the database can be compromised by conflicting transactions executing at different nodes. To solve this problem, update transactions are executed at database nodes in compatible orders, thus producing mutually consistent states on all database replicas. Queries are sent to any node that is fresh enough with respect to the query

requirement. This implies that a query can read different database states according to the node it is sent to. However, since queries are not distributed, they always read a consistent (though stale) state. To achieve global consistency, we maintain a graph in the shared directory, called global precedence order graph. It keeps track of the conflict dependencies among active transactions, *i.e.* , the transactions currently running in the system but not yet committed. It is based on the notion of potential conflict: an incoming transaction potentially conflicts with a running transaction if they potentially access at least one relation in common, and at least one of the transactions performs a write on that relation. This pre-ordering strategy, already used in Leg@net, is comparable to the one of [4]. The main difference is that the global ordering graph is also used for computing nodes freshness

3 Transaction Routing with Freshness Control

In this section, we describe how transactions are routed in order to improve performance. First, we present the routing algorithm, directly inspired from [7]. Then, we discuss the specific issues raised by the use of a shared directory.

3.1 Routing Algorithm

Our routing strategy is cost based and uses late synchronization, thus it takes into account the cost of refreshing a node before sending a transaction to it. As mentioned in [7], the routing complexity is linear in the number of active transactions and the number of nodes, which makes our approach scalable. The cost-based routing algorithm evaluates, for each node N_j:

- N_j's load. This cost is computed by evaluating the remaining execution time of all running or waiting transactions at node N_j.
- the cost of refreshing N_j enough (if necessary) to meet a transaction T freshness requirements. To this end, it computes a refresh sequence S for N_j: the minimal sequence of refresh transactions to be executed on N_j to make it fresh enough *wrt.* T's requirement. In other words, after applying the refresh sequence on N_j, its staleness *wrt.* each relation read-accessed T is lower than the respective staleness tolerated by T (remember that this tolerated staleness is always 0 if T is an update). The cost is the estimated time needed to execute the sequence S.
- the cost of executing T itself.

Then it chooses the node which minimizes the cost, *i.e.* the sum of the preceding three costs, sends the corresponding sequence to this node and finally sends it the transaction T. It also updates the shared directory: all the transactions in S (plus T if T is an update) are dropped from the set of transactions waiting to be executed on N.

In order to ensure global consistency, refresh transactions are inserted in the refresh sequence according to the global serialization order: whenever a refresh

transaction is inserted, all its predecessors not yet executed on the node are also inserted, in the appropriate order, so that the sequence order is compatible with the global precedence order (see Section 2.4)

3.2 Concurrent Access to the Shared Directory

As opposed to the centralized version of [7], where the single router is interacting sequentially with the directory, we must here take into account the concurrency problem due to the presence of several routers, thus to simultaneous access the metadata. We decided to solve this problem using traditional two phase locking (locks on metadata are kept until the end of the routing process), based on two observations: (1) the routing process is very fast compared to the execution of the refresh sequence and of the transaction itself, thus locks are released rather quickly, and (2), locking is provided by JuxMem, which makes the implementation straightforward. In order to validate this choice, we ran experiments to measure the overhead due to concurrent access to the shared directory (see next Section).

4 Experimental Validation

In this section we evaluate the performances of our solution through experimentation. In [7], the Leg@net router was demonstrated to perform better than well known routing strategies such as round robin or least loaded node routing. Since our solution relies on the same cost based routing algorithm, we focus here on comparing the distributed version of the routing algorithm with the Leg@net centralized one.

The experiments follow two goals. First, we need to check that the distributed router is not a bottleneck, *i.e.* , it routes every transaction fast enough. Second, we want to assess if the distributed router brings some global benefit for the applications *i.e.* , if it improves transaction response time.

4.1 Experimental Setup

We run all the experiments on a 20 nodes (P4, 3GHz, 2GB RAM) cluster with 1Gb/s inter node connection as well as some desktop computers from the laboratory to host end user applications. The router is implemented in C language and relies upon JuxMem services, which are built on top of Sun JXTA layer. JuxMem provides a grid-wide RAM access. Our router acts as a middleware; it provides a transaction processing interface for the applications. A cluster node has two roles: it acts as a router node and/or a DBMS node.

4.2 Distributed Directory Access Overhead

The first set of experiments focuses on the routing step itself. It measures the overhead of using a distributed directory to manage router metadata. The workload is made of an increasing number of applications, each of them is sending one

transaction per second to a single router. We measure the resulting throughput (in transaction/second) that the router achieves. Figure 2 shows that a single router can process up to 40 transactions per second. This threshold is satisfying considering that more routers would be able to handle higher workloads.

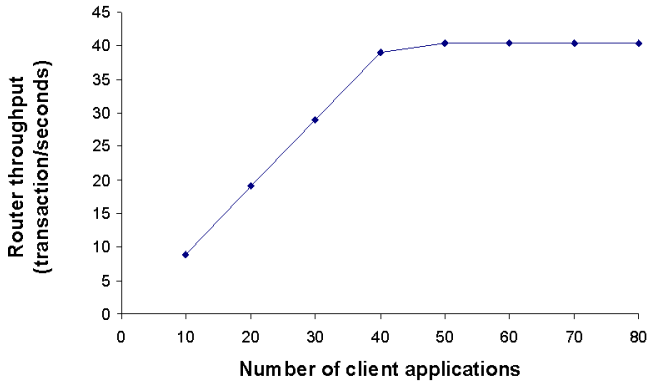

Fig. 2. Middleware throughput

A part of the routing process is to access the distributed directory. In order to quantify the directory access overhead and then to know if our approach can scale out, we increase the directory size by adding database replicas, since more replicas imply more metadata. We report on Figure 3, the output workload that a router achieves in 3 cases: small , medium and large directory size (respectively 5 , 50, 100 replicas). We measure a slowdown of less than 20% for a large directory that has a replication degree of 100. For a smaller replication degree of 50, the slow down is only 5%. Since most of the applications, in our context, require a replication degree lesser than 10, we conclude that the distributed directory access is not a performance brake.

Furthermore, we study the impact of multiple routers concurrently accessing the distributed directory. The workload is made of the same applications as the former experiment, but the transactions are sent to 2 routers (such that half of the workload goes to each router). The results of Figure 4 are obtained in the worst case (*i.e.* all transactions access to the same data leading the routers to do so with metada) and they shows a maximal throughput of 20 transactions/second that is half of the standalone throughput. Indeed, waiting for locks is decreasing the router throughput. Thus, in the worst case where each directory access is delayed by a concurrent access to the same metadata, the router is still able to provide reasonable throughput. We note that, in our context, concurrent situations are not frequent since metadata is fragmented and the probability of concurrent access to the same fragment is weak. Nevertheless, ongoing experi-

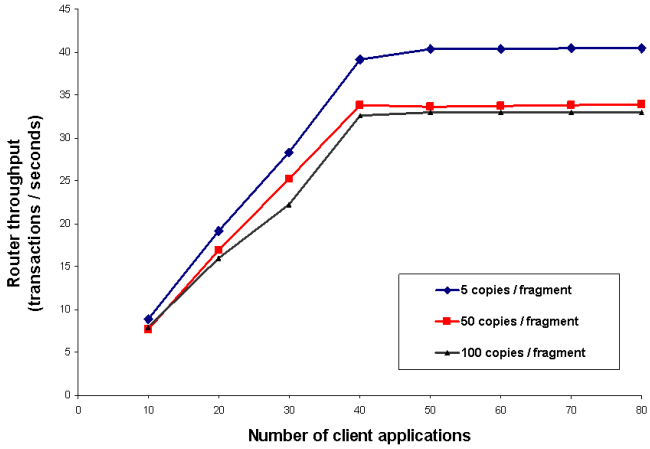

Fig. 3. Directory size overhead

mentations aim to evaluate precisely the slowdown led by concurrent access to the distributed directory *wrt* concurrency degree. In other words, we will vary the concurrency degree between 0% and 100% and measure the variations of the performances.

4.3 Overall Routing Performance

This experiment focuses on the overall transactional performance of the distributed routing (DR). We measure the increase in throughput compared to the centralized routing (CR) of [7]. The workload is made of N applications of 3 kinds (N/3 apps of each kind). Each kind of application is accessing a distinct part of the database and is connected to a distinct router. In other words, there is no concurrency between routers when accessing the directory We measure the output throughput when N is varying from 15 to 150 applications. On Figure 5, we compare theses results with a case where a single router receives the whole identical workload. As n is increasing, the gap between DR and CR is expanding. For a heavy workload of 150 applications, DR outperforms CR by a ratio of 3. The main reason is that centralized routing quickly reaches its performance limit due to the time required to route each transaction. We note that the benefit ratio equals the number of routers: that demonstrates a linear scale up. Ongoing experimentations are conducted to assess up to which number of routers our solution scales linearly.

4.4 Dealing with Scale Up

In order to deal with large scale network, our experiments must take into account the databases and directory replication at large scale such as grid systems.

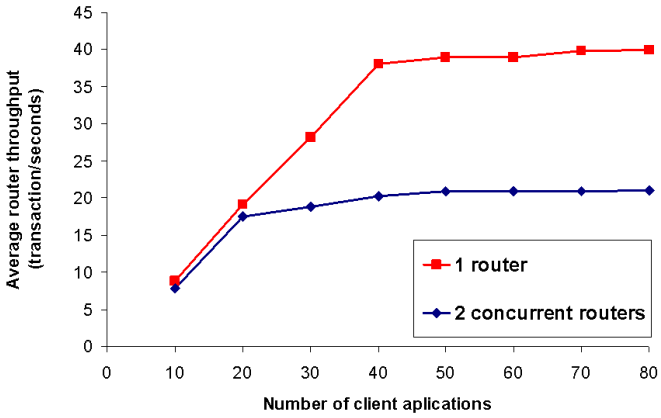

Fig. 4. Concurrent access

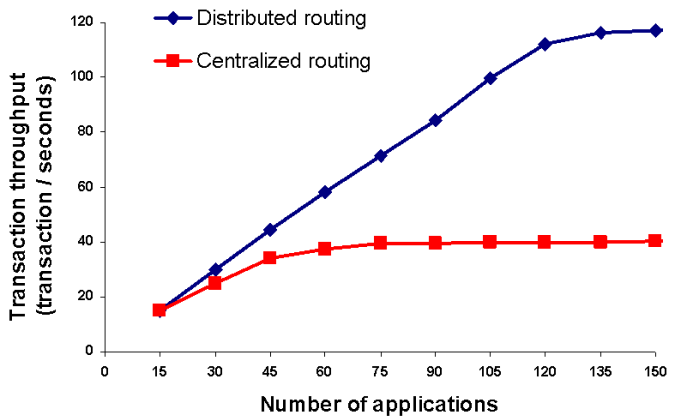

Fig. 5. Distributed vs centralized algorithm

However, in this paper, our main goal is to demonstrate the performance benefit that we achieve by distributing the routing protocol. To this end, replicating our middleware over few nodes, at least one router per cluster, is sufficient. More precisely, we note that JuxMem experiments reported the time to write metadata stored on a remote cluster that belongs to the Grid5000 [14] infrastructure: more than 90 milliseconds per write. In such environment, the routing time of our system would be around 100 milliseconds. Then, the router throughput falls to 10 transactions per second.

However, if every router access only a part of the distributed directory that is managed locally on the same cluster, the throughput performance (up to 40 transaction/second) is still observed, even if the databases are replicated over many remote clusters. In this case, the response time slightly increases depending on network latency between clusters.

5 Conclusion

This paper presents an ongoing work towards the design and implementation of a grid-based large scale data management system. This system extends Leg@net, a previous work designed for clusters, to the grid context. It uses JuxMem, a shared main memory system designed for grids to implement a shared, distributed directory which stores metadata useful for transaction routing and freshness control. The experimental evaluations led on a first version of the system show that the overhead due to accessing to the distributed directory is rather low. They also shows that, using the distributed directory, we can implement several instances of the router in the network. For heavy workloads, this increases significantly the global throughput of the system with respect to the centralized version of the router used in Leg@net.

Ongoing experimentations are conducted to find out the optimal number of router instances with respect to the heaviness of the workload. After, we plan to evaluate precisely the slowdown led by multiple routers concurrently accessing the distributed directory *wrt* to concurrency degree. We will also take into account the grid heterogeneity (intra-cluster links faster than inter-cluster links) in our cost estimations, in order to improve node choice and thus to yield better performances. Next, we will run the same experimentation as the one led with Leg@net to observe the benefits in terms of response time queries can obtain with respect to the staleness they tolerate. Furthermore, we plan to deal with fault tolerance.

References

1. F. Akal, C. Türker, H. Schek, Y. Breitbart, T. Grabs, and L. Veen. Fine-Grained Replication and Scheduling with Freshness and Correctness Guarantees. In *Int. Conf. on Very Large DataBase (VLDB)*, pages 565–576, 2005.
2. Amadeus. http://www.amadeus.com/index.jsp.

3. 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, 2005.
4. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
5. S. Choi, M. Baik, J. Gil, C. Park, S. Jung, and C. Hwang. Group-Based Dynamic Computational Replication Mechanism in Peer-to-Peer Grid Computing. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, page 7. IEEE Computer Society, 2006.
6. Galileo. http://www.galileo.com.
7. S. Gançarski, H. Naacke, E. Pacitti, and P. Valduriez. The Leganet System: Freshness-aware Transaction Routing in a Database Cluster. *Journal of Information Systems*, 32(2):320–343, 2006.
8. R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(40):68–74, 1997.
9. R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, 1987.
10. C. Le Pape, S. Gançarski, and P. Valduriez. Refresco: Improving Query Performance Through Freshness Control in a Database Cluster. In *Int. Conf. On Cooperative Information Systems (CoopIS)*, pages 174–193, 2004.
11. E. Pacitti, C. Coulon, P. Valduriez, and T. Özsu. Preventive Replication in a Database Cluster. *Distributed and Parallel Databases*, 18(2):223–251, 2005.
12. E. Pacitti, P. Minet, and E. Simon. Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases. *Int. Conf. on Very Large DataBase (VLDB)*, 1999.
13. M. Patino-Martinez, R. Jimenez-Peres, B. Kemme, and G. Alonso. MIDDLE-R, Consistent Database Replication at the Middleware Level. *ACM Transactions on Computer Systems*, 28(4), 2005.
14. Grid'5000 Project. http://www.grid5000.org.
15. U. Rohm, K. Bohm, H. Sheck, and H. Schuldt. FAS - a Freshness-Sensitive Coordination Middleware for OLAP Components. *Int. Conf. on Very Large DataBase (VLDB)*, 2002.
16. Sabre. http://www.sabre.com/.