Routage Décentralisé de Transactions avec Gestion des Pannes dans un Réseau à Large Echelle *

Idrissa Sarr, Hubert Naacke, Stéphane Gançarski
Université de Paris 6, Laboratoire d'Informatique de Paris 6, France
Prénom.Nom@lip6.fr

Résumé

Les systèmes à large échelle tels que les grilles fournissent l'accès à des ressources massives de stockage et de traitement. Bien qu'elles furent principalement destinées au calcul scientifique, les grilles sont maintenant considérées comme une solution viable pour héberger les données des grandes applications. Pour cela, les données sont répliquées sur la grille dans l'optique d'assurer leur disponibilité et de permettre une rapide exécution des transactions grâce au parallélisme. Cependant, garantir à la fois la cohérence et la rapidité d'accès aux données sur de telles architectures pose des problèmes à plusieurs niveaux. En particulier, le contrôle centralisé est prohibé à cause de sa faible disponibilité et de la congestion que cela engendre à grande échelle. En outre, le comportement dynamique des noeuds, qui peuvent rejoindre ou quitter le système à tout instant et de manière fréquent peut compromettre la cohérence mutuelle des copies. Dans cet article, nous proposons une nouvelle solution pour la gestion décentralisée du routage des transactions dans un réseau large échelle. Nous transposons une solution de routage dédiée à un cluster de machines et l'améliorons afin que le routage devienne entièrement décentralisé et que les méta-données soient hébergées dans un catalogue distribué sur un réseau à large échelle. Ensuite, nous proposons un mécanisme de gestion des pannes, très adapté à notre contexte, prenant en compte la dynamicité et/ou les pannes des noeuds. Finalement, l'implémentation de notre modèle de routage et son évaluation expérimentale montrent la faisabilité de l'approche ; l'efficacité de la gestion des pannes est mesurée à travers une simulation. Les résultats révèlent une scalabilité linaire, et un temps de routage des transactions suffisamment rapide pour rendre notre solution applicable et adaptée aux applications à forte charge transactionnelle comme les systèmes de réservation en ligne. Les résultats montrent également que la gestion de la dynamicité augmente les performances du système en termes de débit tout en minimisant les coûts de communication.

Mots-clés : Bases de données répliquées, tolérance aux pannes, large échelle, traitement de transactions, équilibrage de charge, performance.

*"Ce travail a été financé partiellement par le projet Respire (ANR-05-MMSA-0011)"

1 Introduction

Large scale systems like Grids use a distributed approach to deal with heterogeneous resources, high autonomy and large-scale distribution. Thus, they are interesting in many areas of enterprise information systems. As an example of application, Global Distribution Systems (GDS) like Amadeus [4], Sabre [29], Galileo [13] manage a huge amount of data for airline companies, hotels and travel agencies. Amadeus GDS provides access to 95% of the global airline capacity, representing more than 500 airlines, 80 000 travel agencies, 70 000 hotels, 33 000 car rental companies, and 80 000 insurance companies. Moreover, each day, 79 500 travel agencies and 10 000 sales offices of airlines around the world connect to the Amadeus system to make 280 million of transactions which correspond to 5500 user requests per second. 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. Moreover, data is usually located on a single site, which limits scalability and availability. Implementing GDS systems onto a grid allows to overcome these limitations at a rather low cost. In such architectures, data accessed by the GDS is stored by the participants (hotels, airline companies, *etc.*) and can be shared. Thus, data is distributed and parallel executions can be performed 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 two problems: concurrent updates and node failures. To illustrate the first problem, let us assume that we have two replicas R_1^i and R_2^i of relation R^i, two transactions T_1 and T_2 which are sent respectively by two applications A_1 and A_2. Each transaction aims at making a reservation operation in the same flight (AF709) of Air France airline company. Assuming that only one seat is available and T_1 is routed to R_1^i and T_2 to R_2^i, then the simultaneous execution of T_1 and T_2 produces a data inconsistency: one of travel agencies sales a non-existing seat. Furthermore let us illustrate the problem due to node failure. Consider again replicas R_1^i and R_2^i of relation R^i and an application A which sends a transaction T for making a seat reservation operation. T is sent to R_1^i which executes it, but fails before sending the results to A. After a while, A sends again the transaction which is routed to R_2^i. If R_2^i performs successfully the execution of T, then T is executed twice. As a consequence, the agency sales two seats to a same customer.

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)* the 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.

In [30], we proposed DTR, a solution which takes into account the problem due to concurrent updates and controls the freshness in order to improve performances. However, DTR does not deal with node failures (or dynamicity) which are very frequent in large scale systems. Dealing with dynamicity is very challenging since it requires designing efficient algorithms for managing node failures and always ensure a certain level of availability for trying again the executions of some

failed transactions. Many solutions have been proposed in distributed systems for managing replicas, such as [24, 22, 23]. Some solutions include freshness control, for instance [28, 14, 19, 2, 27]. Some other, tackle the node failures to ensure fault tolerance, such as [6, 15, 16]. We base our work on the Leg@net approach [14], 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, Leg@net has been designed for a PC cluster. It is dedicated to homogeneous parallel systems. Transposing Leg@net to heterogeneous systems which have independent entities such as GDS, is not straightforward. 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 a grid. To reach this goal, the router and the metadata are replicated at many sites of the grid. Furthermore, we deal with node failures in order to face the dynamic behaviour of nodes which can join and leave the system frequently.

In this paper, we aim to design a new system relying on the Leg@net approach to deal with transaction routing at a large-scale. Our main contributions are:

- A fully distributed transaction routing model which deals with update-intensive applications. Our middleware, ensures data distribution transparency and does not require synchronization (through 2PC or group communication) while updating data. Furthermore, it allows freshness relaxing for read-only queries, which improves the load balancing.

- 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.

- A fault-management mechanism well suited to a large scale system. To this end, we propose an approach which relies on a selective detection and recovery algorithm. On opposite with most of the other approaches, it imply only nodes which participate to the execution.

- An implementation and an evaluation of our approach, both on a medium scale experimental infrastructure and on a large scale simulator. It demonstrates the feasibility of our solution and quantifies its performance benefits.

The rest of this paper is organized as follows. We first present in Section 2 the global system architecture together with the replication and freshness model. Section 3 describes our transaction routing algorithm with freshness control. Section 4 deals with node dynamicity. Section 5 presents experimental evaluations of our distributed routing approach through implementation. Section 6 deals with performance evaluation of our fault management mechanism through simulation. Section 7 presents related work. Section 8 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 for better understanding our solution. Then, we describe the replication and transaction model in Section 2.2. The freshness model and global consistency management are presented respectively in Section 2.3 and Section 2.4. Finally, our failure model is described in section 2.5.

2.1 Global Architecture

The global architecture of our system is depicted on Figure 1.

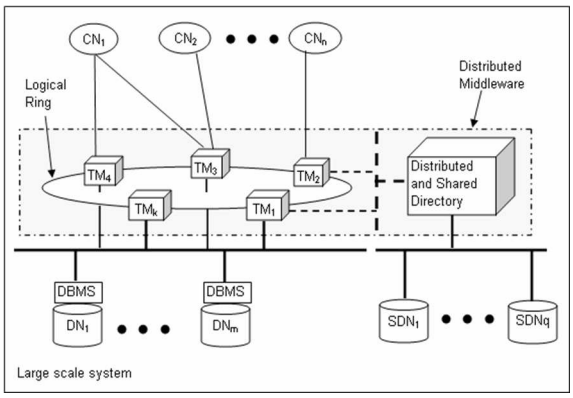

Figure 1: Global architecture

All the nodes are identified through their IP address and communicate through asynchronous messages. We distinguish four kinds of nodes:

- *Client Nodes* (*CN*) send the transactions to any TM. A CN assign to each transaction a global unique identifier (GId) which is required to prevent ambiguity during subsequent routing. The Gid is a pair (ClientId, SeqN) such that ClientId is the client identifier and SeqN is the local transaction sequence number.

- *Transaction Manager Nodes* (*TM*) route the transaction for execution on data nodes while maintaining global consistency. The TMs use metadata stored in the shared directory. The TMs are gathered into a logical ring [17] in order to facilitate the collaborative detection of failures. To this end, each TM knows k predecessors and k successors. The set of successors of TM_i is given by $Suc(TM_i) = \{TM_{((i+j) \ mod \ n)}, 1 \leq j \leq k\}$ where n is the total number of TMs.

- *Data nodes* (*DN*) use a local DBMS to store data and execute the transactions received from the TMs. They return the results directly to the CNs.

- *Shared Directory nodes* (*SDN*) are the building blocks of the shared directory. They are implemented into JuxMem [5]. JuxMem provides the abstraction of a shared memory over a distributed grid infrastructure, by transparently handling consistency in a fault-tolerant way. The shared directory contains detailed information about the transactions processed on the data nodes, *i.e.* for each relation and each node, the list of running and executed transactions. It is used to compute the relation staleness as described in Section 2.3. 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 3.2).

2.2 Replication and Transaction 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 are 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. Thus, they do not need to 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

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.

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 [19]. In this paper, we only consider one measure, defined as the number of missing transactions. More precisely, the staleness of R_j^i is equal to the number of transactions writing R^i already performed on any node but not yet executed 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 that 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. 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.

We compute the staleness of a relation copy R_j^i based on the system global state stored in the shared directory that gives detailed information about committed and running transactions.

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 [7]. The main difference is that the global or-

dering graph is also used for computing nodes freshness

2.5 Failure Model

In this paper, we deal with systems which have only two kinds of components: nodes which process the transactions (CN, TM, and DN), and network communications. Each of these components can fail when the system runs, leading to node or communication failure. In this paper, we focus on the following failure types.

2.5.1 Node Failure

When a node fails, its processes stop abnormally and can lead to inconsistencies. We assume that a node is always either working correctly or not working at all (it is down). In other words, we assume fail-stop failures and do not deal with Byzantine failures.

2.5.2 Communication Failure

A communication failure occurs when a node N_i is unable to contact node N_j, even though none of the two nodes is down. When such a failure happens, no message is delivered.

In our context, communication is asynchronous. Thus, each message received by a node must be acknowledged. Without this acknowledgment, we assume that the message is lost due to a communication failure or a node failure (see Section 2.5).

2.5.3 Failure Detection

Usually, the failures are detected either periodically by heartbeat messages [1], or on demand by ping-pong messages [17]. [10] presents the principles of collaborative detection targeted to large scale systems. We use heartbeat and ping-pong messages according to the node type. Indeed, the failure detection requirements for DNs differ from those for TMs. First, the number of TM is very small compared to the number of DN. Second, the TMs collaborate with the others to detect both DN failures and communication failures between TM and DN.

Because the number of DN is high, periodic detection would potentially use a lot of network bandwidth, compromising the overall performance. Thus, we decide to detect DN failure without additional cost by integrating failure detection into the routing protocol. A failed DN is detected only when a TM attempts to send a transaction to that DN. This detection mechanism does not prevent the case where a DN is actually down without being detected, while no transaction is sent to it. However, this would have minor impact on the overall routing.

The collaboration between the TMs to handle DN failure detection requires to minimize the occurrence of undetected failed TMs because it would be misinterpreted as a communication failure. Thus, TM failure detection is managed by periodic heartbeat messages.

3 Transaction Routing with Freshness Control

In this section, we describe how the transactions are routed in order to improve performance. First, we present the routing algorithm, directly inspired from [30] and [14]. Then, we discuss the specific issues raised by the use of a shared directory.

3.1 Routing Process Specification

By simplifying the approach described in [30], the transaction processing can be split in three steps.

1. **Setup phase**: During this phase, a CN

contacts a TM in order to submit a transaction. We assume that any CN knows some of the TMs (not necessary all). A CN chooses a TM by round robin among the TMs it knows.

2. **Routing phase**: It is the phase where a TM performs the routing algorithm (see Section 3.2) for sending the incoming transaction to a data node. This phase finishes when the chosen data node receives the transaction.

3. **Execution phase**: This last phase starts after a data node has received a new transaction and lasts until it sends results to the CN which initialized the transaction and sends to the TM a notification of the successful or failure end of execution. Results are sent directly to the CN, since the TM transmits also the Client identifier (its IP address) to the chosen DN.

3.2 Routing Algorithm

In this section, we describe the routing algorithm handled by any TM when it receives an incoming transaction. 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 [14], 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 N which minimizes the cost, *i.e.* the sum of the preceding three costs, and sends to N the sequence S followed by 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)

3.3 Concurrent Access to the Shared Directory

As opposed to the centralized version of [14], 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. Results are shown in Section 5.

4 Dealing with Node Dynamicity

We describe the approach used to deal with a node joining or leaving the system during the execution of a transaction. We assume that any node (CN, TM or DN) joining the system is able to locate one available TM. The contacted TM is then responsible for including the new node by updating either the ring (TM join) or the shared directory (CN or DN join).

Since our main goal is to preserve consistency whenever a node leaves, we consider the disconnection of TMs and DNs (if a CN leaves the system, this does not compromise data consistency since the CN delegates transaction processing to the TM). Then, we distinguish two situations: predictable and unpredictable disconnection.

4.1 Predictable Disconnection

A predictable disconnection occurs when a node deliberately decides to leave the system.

Predictable Disconnection of a TM. When a TM decides to leave the system, it informs its predecessors (resp. successors) which update the logical ring by removing it from their list of successors (resp. predecessors). During the disconnection, the TM simply ignores the incoming messages it receives.

Predictable Disconnection of a data node. If a DN wants to disconnect, it sends a message called *Disconnection request* to the last TM which sent it a transaction and is still available. This TM, when receiving this message, removes the DN from the list of available DNs. This prevent any other TM from routing an incoming transaction to this leaving DN. Then, the TM sends to the DN a message called *Disconnection accepted*. Thus, the DN is considered as disconnected until subsequent notification to join the system.

4.2 Unpredictable Disconnection

In order to deal with unpredictable disconnection, we detail each of the three phases defined in Section 3. We assume that there is at least always one available node (TM or DN) on which we can rely whenever we detect a node failure.

Fault-Management during Setup Phase. After sending a transaction to a TM, the CN uses two timeouts: δ_{s1} for ensuring that the transaction reaches its destination, and δ_{s2} for being aware that the execution is done. When δ_{s1} elapses, if there is no acknowledgement from the TM previously contacted, the CN deduces that a communication or a TM Node failure occurred. Two cases can be identified at this point: (i) The TM was unreachable due to either a communication failure or its own failure. (ii) The TM has received the transaction, has sent an acknowledgement to the CN and routed the query to a chosen data node. Unfortunately, a communication failure occurred before the reception of the acknowledgement by the CN.

In the first case, the CN retransmits the transaction with the same global identifier. In order to give to the retransmission a successful outcome, the CN increases the number of

targeted TMs while sending again the transaction. More precisely, the CN appends one more TM to its destination list, each time it renews its attempt to send the transaction. Candidate TMs are chosen in a round robin fashion among the TMs known by the CN. Notice that consistency can not be compromised since the transaction is not delivered to any DN for execution. Even if several TMs receive the same transaction, this will be sent to only one data node, thanks to the use of global identifier and of a shared directory.

In the second case, the transaction can be already performed or currently running. To avoid performing the transaction twice, any TM which receives the retransmission from one CN, checks if another transaction with the same global identifier is already existing in the shared data and, if it is the case, notifies the CN. Each CN, upon receiving this kind of notification, stops any retransmission of the same transaction.

When the second timeout δ_{s2} elapses, if the CN did not receive the results of the transaction, it contacts a TM for sending again the query. Two cases can be identified with respect to the retransmission issue: (i) The transaction is executed, but the CN failed after the transaction has committed, thus could not be reached. In this case, the TM order the data node to send again the results to the CN. (ii) The transaction is in progress, the CN is invited to wait and to reinitialize its second timeout δ_{s2}.

In order to reduce useless and frequent retransmissions, timeouts values are based on the network latency and average time to process transactions. Then, $\delta_{s1} \geq \ 2 * \lambda_N$ and $\delta_{s2} > \delta_{s1} + Avg(T)$ where λ_N is the network latency and *Avg (T)*, the average time to process transaction T.

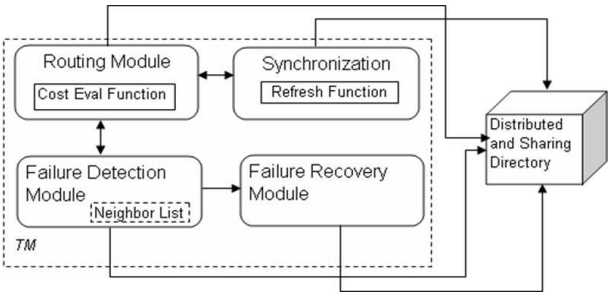

Figure 2: TM architecture

Fault-Management in Routing Phase. Each TM_i initializes a timeout δ_r after sending a transaction to a node. When δ_r expires, and TM did not receive a response from data node DN_i, it chooses the least loaded data node DN_j without considering DN_i. In parallel, it invokes the *Failure Detection Module*(*cf.* figure 2) which checks if DN_i is available or not. To this end, it contacts some of its predecessors and successors. Each of these TMs, try to contact the suspicious node DN_i by sending it a message. The results of these handshaking are returned to the original TM. If all the results are negative it concludes that DN_i has failed and append it to the Node Failure list, called *NFList*, stored in the shared directory. Conversely, if one of these results is positive, the TM assumes that there is a temporary communication failure between itself and DN_i. Thus, it can consider DN_i like a potential candidate for later processing. The failure manager, called *Failure Recovery Module* (*cf.* Figure 2), is responsible for checking the recovery of any failed node, and to remove it from the *NFList* as it is done in [10].

Fault-Management in Execution Phase. A data node performs transactions sent by TMs, and notifies the end of their executions. It sends results back to the original CN. In

order to notify the TM about the end of a transaction, a DN sends a message *'Up Data'* if transaction validates, or *'No Data'* otherwise (*i.e.* when the transaction execution has failed). After sending the message *'Up Data'*, the DN initializes a timeout δ_e and waits an acknowledgement from the TM. If DN does not receive this acknowledgement when the δ_e expires, it adds this message to a buffer for sending it subsequently. Messages stored in the buffer can be sent again by a data node in two possible ways: *(i)* using *piggybacking* while sending termination notification, or *(ii)* sending them periodically to the least loaded TM. We currently use piggybacking for sending the buffered messages, since it reduces the number of messages sent to the TMs.

When a TM receives a message *'No Data'* from a DN, it chooses another DN. When it receives messages *'Up Data'*, it updates the shared directory and all corresponding transactions are considered as processed on the DN.

5 Experimental Evaluation

In this section we evaluate the performances of our solution through experimentation. In [14], 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.

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

5.1 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. We note that our solution weakly depends on JuxMem, since the separation between our implementation and JuxMem is clearly defined by the JuxMem API. We could use any software (*e.g.* [11]) that provides the same functionalities as JuxMem, however the performances might differ. 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.

5.2 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 3 shows that a single router can process up to 40 transactions per second.

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 5.2, the output workload that a router achieves in 3 cases:

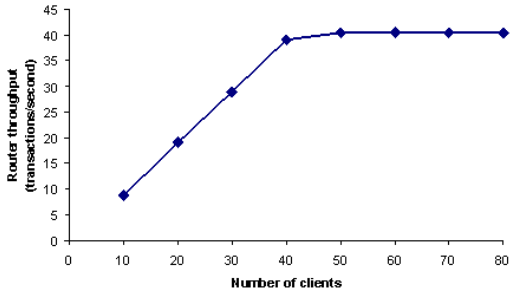

Figure 3: Middleware throughput

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 slowdown 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.

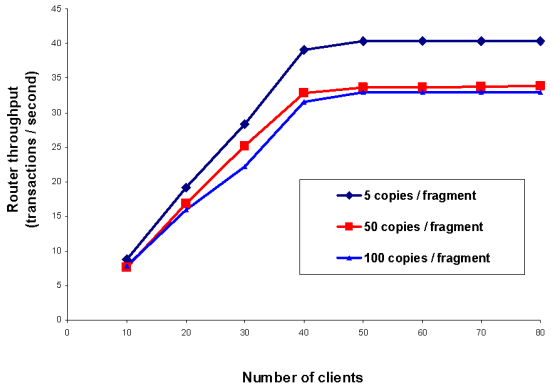

Figure 4: Directory size overhead

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 5 are obtained in the worst case (*i.e.* all transactions access to the same data leading the routers to do so with metadata) 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 experimentations 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. For the best case (concurrency degree equals 0) it is expected that the global throughput equals to the maximal throughput of a router (40 transactions/second) multiplied by the number of routers as it is shown in the next section.

5.3 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 [14]. 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

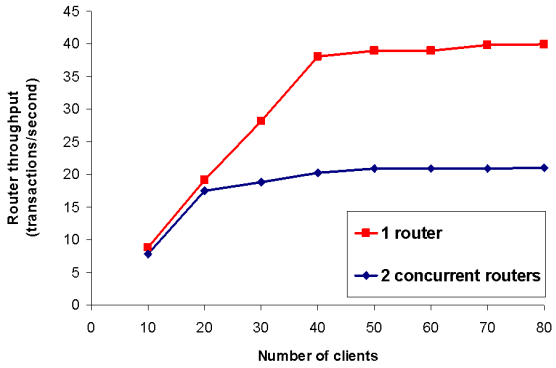

Figure 5: Concurrent access

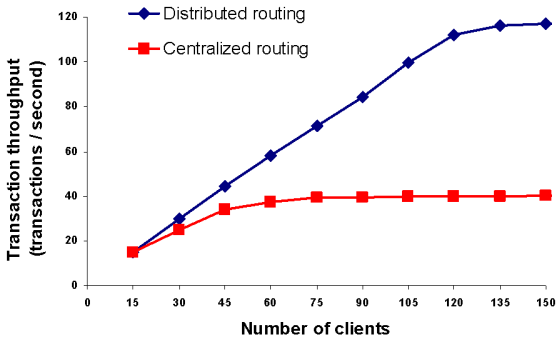

Figure 6: Distributed vs. centralized algorithm

no concurrency between routers when accessing the directory (concurrency degree equals 0). We measure the output throughput when N is varying from 15 to 150 applications. On Figure 6, we compare theses results with a case where a single router receives exactly the same 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. In order to support a transactional load equivalent to that of a GDS like Amadeus, it would require less than 200 routers (assuming a low conflict rate). Ongoing experimentations are conducted to assess up to which number of routers our solution scales linearly.

5.4 Impact of Relaxing Freshness

In this section, we study the influence of tolerated staleness on performances. We measure how much relaxing the freshness can impact the response time of transaction and load balancing. We choose a medium size setup: 40 applications (20 for update transactions and 20 queries) and 20 DNs. We vary the tolerated staleness of queries. Figure 7 and 8 shows the response time of transactions and query versus their tolerated staleness (expressed as a number of transactions). The results show that increasing the tolerated staleness improves significantly the query and transaction response time. This is mainly due to the fact that increasing the tolerated staleness gives more flexibility to postpone synchronization, and thus to execute transactions more quickly. We note that for a staleness value greater than 15, the response time does not increase. This is because, the lowest average time to process a transaction is reached.

Of course, freshness relaxing comes at the cost of node getting stale. Figure 9 shows the global staleness of the system at the end of the experiment, according to the tolerated staleness of queries. To maintain nodes at a reasonable level of freshness, we plan to use the background refresh strategies presented in [18].

Furthermore we evaluate the improvement of the load balancing achieved by relaxing freshness. To this end, we measure the *un-*

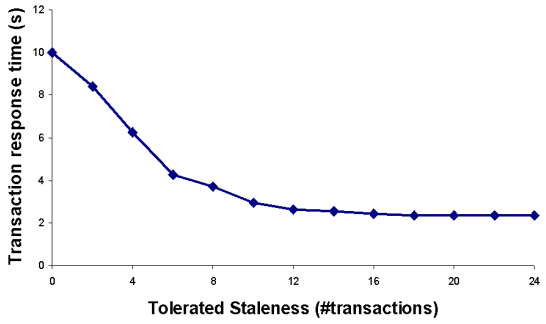

Figure 7: Transaction response time vs. tolerated freshness

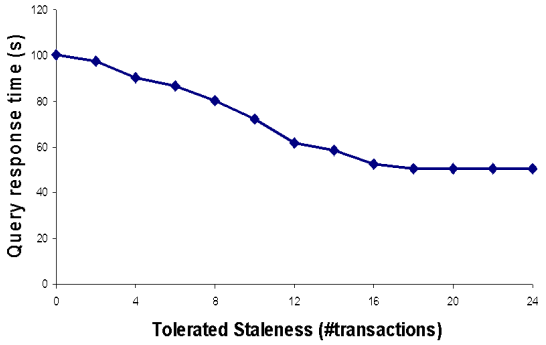

Figure 8: Query response time vs. tolerated freshness

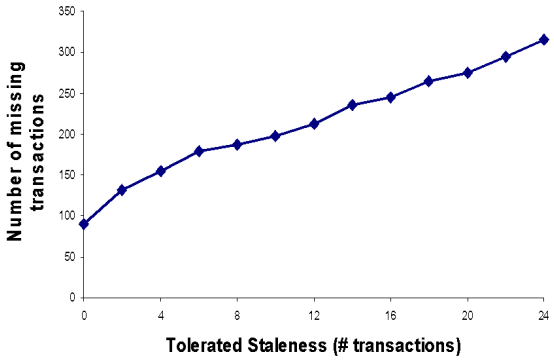

Figure 9: Number of missing updates vs. tolerated freshness

balance rate (τ) of the load. The unbalance rate shows the deviation of the current load balancing *wrt* a perfect balance. To get the deviation, we divide the standard deviation (σ) by the average load (E): $\tau = \frac{\sigma}{E}$. The results on Figure 10 show that tolerated staleness reduce by a factor of 2 the initial deviation, *i.e.* when queries do not tolerate any staleness. In other words, we get a better balance when tolerated freshness increases, due to the weaker requirements of queries

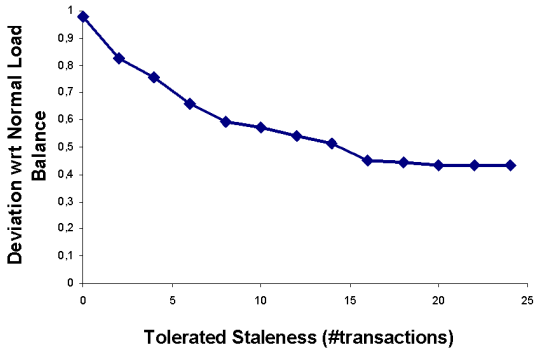

Figure 10: Unbalance rate vs. tolerated freshness

5.5 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. 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 [20] reported the time to write metadata stored on a remote cluster that belongs to the Grid5000 [26] infrastructure: more than 90 milliseconds per write. In such en-

vironment, the routing time of our system would be around 100 milliseconds. Then, the router throughput fall 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.

6 Validation Through Simulation

Now, we evaluate the performance of our fault management protocol through simulation. This section is organized as follows. Section 6.1 describes the simulation setup and Section 6.2 presents the improvement yield by detecting failures with our approach. We ran two distinct simulations to evaluate the occurrence of TM failures and of DN failures.

6.1 Simulation Setup

In order to evaluate our solution at large scale with hundreds of nodes, we implement our protocol using PeerSim[25] simulation tool. Since we aim to measure the impact of node failures, we use the cycle-based engine of PeerSim, which provides scalability, and hides the details of the transport layer in the communication protocol stack. The behaviour of each node (CN,TM, and DN) is plugged into the Peersim framework as specific protocols implemented in Java. We also implemented the shared distributed directory used by TMs to route queries.

6.2 DN Failure Detection

To simulate the dynamicity of the DNs, we vary the number of failed DNs from 0 to 10% (we assume that 10% is representative of the worst case in real applications). Table 1 shows the simulation parameters.

Parameter	**Values**
Number of DNs	500
Failed DNs	10%
Number of transactions/cycle	40

Table 1: Simulation parameters

In our solution, DN failure detection is integrated into the routing algorithm (called DTR2) of the TM. Our objective is to compare DTR2 with the former DTR routing algorithm proposed in [30]. DTR neither detects node failure nor manages them: any transaction sent to a failed DN does not terminate. We compare the performance of DTR2 versus DTR in terms of number of transactions executed in one run. The workload is made of 40 applications. Each application sends one transaction per cycle. A simulation lasts 60 cycles. We use 10 TM nodes which are availaible during the whole simulation. We vary the number of DN from 50 to 200. We report on Figure 11 the number of executed transactions. We observe that DTR2 outperforms the basic DTR by a factor of 140% when the number of DN is greater than 100. Two reasons explain this improvement. First, with DTR2, a node that has already been detected as unavailable will not receive any more transaction. Second, if a DN fails while executing a transaction, the TM will switch to another DN in order to terminate the transaction. We note that both DTR and DTR2 performance does not increase when the number of replicas reaches 180. Indeed, adding more DN will not im-

prove performance because there is already enough DN to process the incoming workload. Furthermore, the gap between DTR and DTR2 remains constant because DTR does not detect unavailable node.

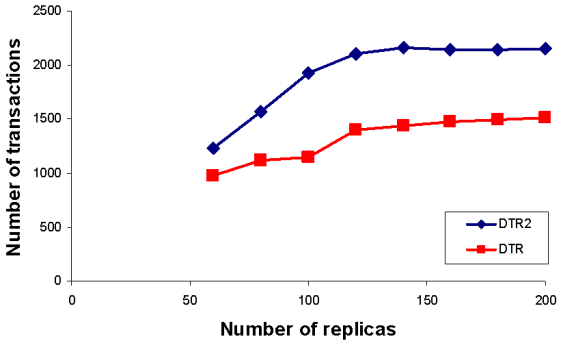

Figure 11: Throughput vs. number of replicas

6.3 TM Failure Detection

In this Section, we compare DTR2 with the approach proposed in SWIM [12], in terms of communication cost (*i.e.* the total number of messages needed to detect failures). In SWIM, detection is done by periodically sending a "ping-pong" message to a random subset of 4 nodes. In DTR2, each TM collaborates with 4 others TMs to handle failure detection.

We start the simulation with 18 available TMs and 2 failed TMs. Then, the number of failed TMs varies from 2 to 18. We report on Figure 12 the number of detected failures.

Figure 12 shows that with a high number of occurred failures, DTR2 yield better performances than SWIM in terms of node failures detected. The main reason is that any failed TM is detected by at least one of its successors, then it is excluded from the ring. On the

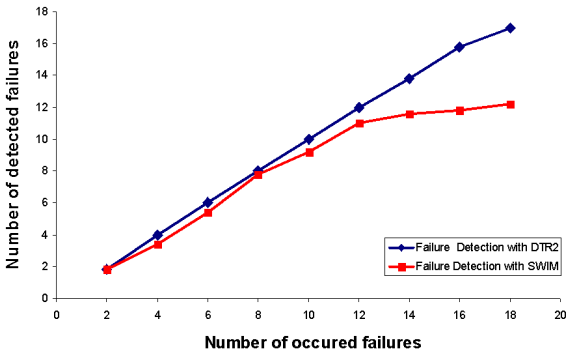

Figure 12: Detected failures vs. occurred failures

contrary, with SWIM, a failed TM may not be detected since it can stay a while without being randomly chosen by other TMs.

Furthermore, we compare the communication cost led by detecting failures with DTR2 and SWIM approach. We compute the total number of messages sent by any node in order to detect failures occurrences.

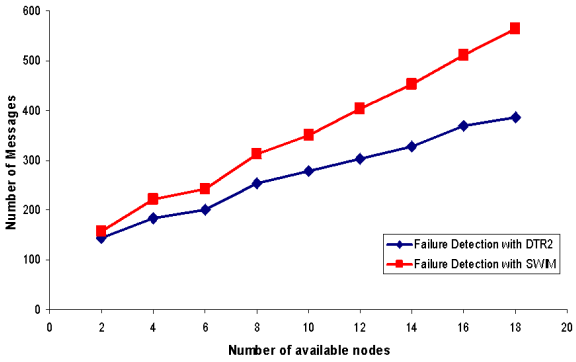

Figure 13: Number of messages vs. occurred failures

The results obtained in Figure 13 shows that the total number of messages for detecting failures increases when the number of available nodes is heavy. In fact, with 20 TMs

the total number of messages is greater than 400 during one simulation, since any TM periodically contacts its successors. Moreover, we notice that DTR2 always requires fewer messages than SWIM for detecting failures. Indeed, the logical ring is restructured as soon as a failed node is detected, avoiding to contact a failed node more than once. In SWIM, a failed node may still be contacted by other nodes that are not yet aware of the failure, since failure notification is not immediate.

We plan to run further simulations to investigate the impact of combined failures *i.e.* when both TM and DN node involved in the same transaction simultaneously fail. Although this case would not compromise data consistency, it would slowdown the transaction response time.

7 Related Work

Our work fits in the context of transaction processing over distributed and replicated databases. We distinguish existing solutions for query and transactions processing in that context, depending on the replication framework upon which these solutions rely. First, replication type depends on the number of updatable copies *i.e.* either mono or multi-master replication (*a.k.a. primary copy* and *update anywhere*). Second, replica update strategy is either synchronous or not. Most of existing solutions have functionalities close to our solution. In order to compare ourselves with theses solutions we enumerate their main functionalities:

- Autonomy: to preserve existing databases without customizing them, in order to seamlessly integrate operational sources.

- Consistency enforcement: to consistently execute simultaneous transactions. We distinguish 3 consistency levels. Strong consistency level guaranties that read and write operations are always consistent (1 copy serializability). Outdated read consistency level guaranties consistent writes but allows for reading outdated versions of the same data. Weak consistency level allows for write operations to be lost.

- Required freshness: to guaranty that the query result always satisfies the freshness required by the client. This allows to query outdated data, under some limit, and to postpone replica propagation in order to save computing resources.

- Load balancing: to spread work over replicas to improve response time and/or throughput.

- Availability (or Failure Management): to handle some typical cases of execution failure due to data sources joining or leaving the system. We distinguish between solutions that assume every part of their architecture to be fully available, and solutions that consider potential unavailability of each components. In between, [3] achieves availability of the timestamp service used for transaction ordering, but do not studies the case of unavailable data sources.

- Scale: to deal with a very large number of databases distributed over a wide area network. Medium scale solutions are dedicated to cluster systems, assuming a homogeneous and stable execution environment. Large scale solutions take into account dynamic environments such as the varying communication latency.

Next, we summarize related solutions before comparing them.

Middle-R [24] is a middleware oriented solution that focuses on dynamic adaptation to failures and workload variation. Synchronous replication guaranties data consistency. It improves former work on active replication such that [15] and [31]. The major drawback is the overhead implied by group communication used to synchronize replicas: this requires high speed interconnects and thus is restricted to cluster systems.

C-JDBC[9] is a Java middleware designed as a JDBC driver. It offers transparent transaction processing on a cluster of replicated databases. Routing strategy aims to be simple and efficient: round robin routing for query, send each SQL statement everywhere. In asynchronous mode, consistency is not guaranteed since the first database that responds is optimistically designated for reference, without taking care of the other databases. Thus, this solution is restricted to a stable environment.

FAS [28] is a coordination middleware which takes into account the node freshness *wrt.* query's requirement. It sends all updates to a designated OLTP node and routes any incoming query to the least loaded of the remaining cluster nodes (called OLAP nodes). It propagates updates to OLAP nodes by deferred bulk refresh transactions. It is not suited for update-intensive applications because of its mono master configuration. Moreover, if no node is fresh enough, the query will simply wait for a node to be refreshed, consequently overloading a node, as soon as it will be fresh enough. In that particular case, immediately refreshing an idle node would have yield better response time.

[3] deals with data currency in replicated DHT and it provides a scalable solution for data sharing in P2P systems. Mutual consistency is guaranteed by using a timestamp service which finds efficiently the current replica. The failure of any node providing the timestamp service is detected; automatic recovery is performed by choosing another node. However, availability of nodes which store replicas is not addressed. Then, if the node which receives the last update leaves the system before propagating it to the other nodes, this update becomes unavailable and can lead to inconsistencies.

The RepDB [22] solution provides strong consistency in replicated database and uses FIFO reliable multicast which is not easy to achieve in a large scale network. In order to guarantee consistency, transactions are performed according to a timestamp scheduling and committed only after a delay which depends on network latency. Thus, this solution is restricted to cluster where network is fast and reliable.

Sprint [8] is a middleware infrastructure for high performance and availability data management. Consistency is ensured by ordering transactions and thus avoiding distributed locks. However, Sprint uses a voting protocol and requires each node to implements the termination protocol. Thus, node autonomy is compromised and whenever a node fails, the vote process fails and leads to transaction abort.

Leg@net[14], is a freshness-aware transaction routing in a database cluster. It uses multi-master replication and relaxes replica freshness to increase load balancing. It targets updates-intensive applications and keeps the databases autonomous. However, it is not suited to large scale systems since the middleware and metadata are centralized.

While surveying related work, we pointed an interesting approach to recover from failures using replication, in the domain of object interoperability [21]. However this solution is not targeting data management or transaction processing.

Finally, we check in Table 2 the main properties that clasify each related solution

Existing Solutions Characteristics	Middle-R	C-JDBC	FAS	Data Currency [3]	RepDB	Sprint	Leganet	DTR2
Replication Framwork	Multi+ Sync	Multi+ Async	Multi+ Sync	Multi+ Async	Multi+ Async	Multi+ Async	Multi+ Async	Multi+ Async
Autonomy	Y	Y	N	Y	N	N	Y	Y
Consistency enforcement	Strong	Weak	Outdated read	Weak	Outdated read	Strong	Outdated read	Outdated read
Freshness	N	N	Y	N	N	N	Y	Y
Load balancing	Y	Y	Query Only	N	N	N	Y	Y
Availability (or Fault-Management)	Y	Y	N	Partially	N	Y	N	Y
Scale	Medium	Medium	Medium	Large	Medium	Medium	Medium	Large

Table 2: Characteristics of related work vs. ours

in comparison with our work. Although the list of functionalities is not exhaustive, Table 2 shows that (to the best of our knowledge) only one of the surveyed solutions supports more than 3 out of 6 functionalities.

8 Conclusion

This paper presents DTR2, the design and implementation of a large scale data management system. DTR2 is designed for large scale and data intensive applications such as global distribution system.

We extend a previous work, DTR, with fault management capabilities. We use 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. We propose a protocol to face every situation when a node is leaving the system during transaction processing. We carefully adapt existing detection protocols to meet the different requirements of each node type (TMs and DNs).

The experimental evaluations of the system show that *(i)* the overhead due to the distributed directory access is low. *(ii)* By using the distributed directory, we can implement several instances of the router. For heavy workloads, this significantly increases the global throughput of the system with respect to the centralized version of the router. *(iii)* We achieve good performances in terms of response time and load balancing with respect to the tolerated staleness of queries.

Moreover, the simulation evaluation shows the effectiveness of our fault-management protocol which improves performance in terms of number of executed transactions and communication cost comparing to existing approaches.

Ongoing experimentations are conducted

to find out the optimal number of router instances with respect to the workload pattern. We will also take into account the grid heterogeneity (intra-cluster links faster than inter-cluster links) in order to improve node choice and failures detection and thus to yield better performances. Next, we plan to enhance transaction managers with self-adaptive capabilities and to investigate how the system behaves in case of strong nodes dynamicity.

Acknowledgments We thank G. Antoniu and L. Breuil for their help in setting up the JuxMem software.

References

[1] M. Aguilera, W. Chen, and S. Toueg. Using the Heartbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks. *Theoritical Computer Science (TCS)*, 220(1), 1999.

[2] 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 DataBases (VLDB)*, 2005.

[3] R. Akbarinia, E. Pacitti, and P. Valduriez. Data Currency in Replicated DHTs. In *Int. Conf. on Management of Data (SIGMOD)*, 2007.

[4] Amadeus. www.amadeus.com.

[5] 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), 2005.

[6] G. Antoniu, J. Deverge, and S. Monnet. How to Bring Together Fault Tolerance and Data Consistency to Enable Grid Data Sharing. *Concurrency and Computation: Practice and Experience*, 18(13), 2006.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[8] L. Camargos, F. Pedone, and M. Wieloch. Sprint : A Middleware for High-Performance Transaction Processing. In *ACM European Conf. on Computer Systems (EuroSys)*, 2007.

[9] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. Technical report, ObjectWeb, Open Source Middleware, 2005.

[10] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2), 1996.

[11] D. Chen, H. Dwarkadas, S. Parthasarathy, E. Pinheiro, and M. L. Scott. Interweave: A middleware system for distributed shared state. In *Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR)*, 2000.

[12] A. Das, I. Gupta, and A. Motivala. SWIM: Scalable Weaklyconsistent Infection-style Process Group Membership Protocol. In *Int. Conf. on Dependable Systems and Networks (DSN)*, 2002.

[13] Galileo. www.galileo.com.

[14] 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), 2006.

[15] R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(40), 1997.

[16] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, 13(1), 1987.

[17] M. Larrea, S. Arévalo, and A. Fernandez. Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous Systems. In *Int. Symposium on Distributed Computing (DISC)*. Springer, 1999.

[18] C. Le Pape and S. Gançarski. Replica Refresh Strategies in a Database Cluster. In *High-Performance Data Management in Grid Environments (HPDGrid VECPAR Workshop), selected papers*, 2006.

[19] 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)*, 2004.

[20] S. Monnet. *Gestion des Données dans les Grilles de Calcul: Support pour la Tolérance aux Fautes et la Cohérence des Données*. Thèse de doctorat, Université de Rennes 1, 2006.

[21] P. Narasimhan, L. Moser, and P. Melliar-Smith. Strongly consistent replication and recovery of fault-tolerant corba applications. *Computer System Science and Engineering Journal*, 17(2), 2002.

[22] E. Pacitti, C. Coulon, P. Valduriez, and T. Ozsu. Preventive Replication in a Database Cluster. *Distributed and Parallel Databases*, 18(3), 2005.

[23] E. Pacitti, P. Minet, and E. Simon. Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases. In *Int. Conf. on Very Large DataBases (VLDB)*, 1999.

[24] 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 (TOCS)*, 28(4), 2005.

[25] PeerSim. peersim.sourceforge.net.

[26] Grid'5000 Project. www.grid5000.org.

[27] K. Ramamritham and C. Pu. A Formal Characterization of Epsilon Serializability. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(6), 1995.

[28] U. Rohm, K. Bohm, H. Sheck, and H. Schuldt. FAS - a Freshness-Sensitive Coordination Middleware for OLAP Components. In *Int. Conf. on Very Large DataBases (VLDB)*, 2002.

[29] Sabre. www.sabre.com.

[30] I. Sarr, H. Naacke, and S. Gançarski. DTR: Distributed Transaction Routing in a Large Scale Network. In *High-Performance Data Management in Grid Environments (HPDGrid VECPAR Workshop)*, 2008.

[31] F. B. Schneider. *Distributed Systems (2nd Ed.)*, chapter Replication Management Using the State-Machine Approach. ACM Press, 1993.