

Failure-Tolerant Transaction Routing at Large Scale

Idrissa Sarr

UPMC Paris Universitas

LIP6 Laboratory Paris, France

idrissa.sarr@lip6.fr

Hubert Naacke

UPMC Paris Universitas

LIP6 Laboratory Paris, France

hubert.naacke@lip6.fr

Stéphane Gancarski

UPMC Paris Universitas

LIP6 Laboratory Paris, France

stephane.gancarski@lip6.fr

Abstract—Emerging Web2.0 applications such as virtual worlds or social networking websites strongly differ from usual OLTP applications. First, the transactions are encapsulated in an API such that it is possible to know which data a transaction will access, before processing it. Second, the simultaneous transactions are very often commutative since they access distinct data. Anticipating that the workload of such applications will quickly reach thousands of transactions per seconds, we envision a novel solution that would allow these applications to scale-up without the need to buy expensive resources at a data center. To this end, databases are replicated over a P2P infrastructure for achieving 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. Moreover dynamic behavior of nodes, which can join and leave the system at anytime and frequently, can compromise mutual consistency. In this article, we propose a failure-tolerant solution for the distributed control of transaction routing in a large scale network. We leverage a fully distributed approach relying on a DHT to handle routing metadata, with a suitable failure management mechanism that handles nodes dynamicity and nodes failures. Moreover, we demonstrate the feasibility of our transaction routing implementation through experimentation and the effectiveness of our failure management approach through simulation.

Keywords-Database replication; middleware; failure management.

I. 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 emerging Web 2.0 applications such as virtual worlds, social networks, wikis and blogs. These applications are characterized by a huge amount of data and therefore a heavy workloads to manage. In [16], eBay is reporting to manage 100 millions of items classified into 50000 categories, all together sizing 2000 terabytes of data. Moreover, each day the underlying database is processing 130 millions of user procedure calls which corresponds to an average load of 1500 transactions per second. The challenge, facing such heavy workloads, is to ensure data availability and consistency in order to deal with fast updates.

To solve this problem, today's applications are using expensive parallel servers. Moreover, data is usually located

on a couple of datacenters, which limits scalability and availability. Nevertheless, using a grid or a P2P approach to implement Web2.0 applications appears to be cost effective. With a P2P approach, data is stored over the nodes and can be shared or accessed without any centralized mechanism. Data is distributed and replicated so that parallel execution can be performed to reduce response time through load balancing. This approach copes with Web2.0 features as described above. However, because P2P systems are highly dynamic, dependability does not come for free. More precisely, mutual consistency can be compromised, because of concurrent updates. Since node failures/disconnections occur frequently, the system must be adaptive to cope with changes in the network topology while maintaining global consistency. Scalability and response time are also crucial issues because of the huge amount of users and data.

Many solutions have been proposed in distributed systems for managing replicas, such as [14][12][13]. Some solutions tackle the node failures to ensure fault tolerance, such as [2][8][9]. As far as we know, most of existing solutions for distributed transactions management fail to get good performances if participating databases are not available, because concurrency control relies on blocking protocols, and the commit protocol is waiting for a majority of participants to answer.

Recently, we proposed two approaches for managing transactions at large-scale. Both of them ensures global data consistency during transaction processing, by controlling concurrent access to the shared metadata in a pessimistic [17] or optimistic [18] way. We leverage the two approaches with a suitable failure management mechanism which is general enough to be applied as is on both approaches. Dealing with nodes failures at large scale is very challenging since it requires to design efficient decentralized algorithms to prevent any single point of failure and to achieve scalability. Thus, we propose a collaborative solution to detect and resolve failures.

Our goal is to guarantee that every correct transaction requested by a client will eventually terminate. More precisely, whenever a node fails during transaction processing, another similar available node will automatically continue to process the transaction. Failures are managed in a straightforward way using timeouts, retransmissions, and maintaining lists

of failed sites. However, the novelty of our approach is to take advantage of the knowledge of each node role involved in transaction processing, to design a transaction oriented failure management solution. Our solution aims to perform better than a generic solution by taking into account the specific requirements of each node in terms of failure management.

Our main contributions are:

- A failure management mechanism well suited to a large scale system. To this end, we propose a selective approach that allows for adapting the subset of nodes responsible for failure detection and recovery. On the opposite of most of existing approaches, it only involves nodes which participate to the execution of a transaction.
- A formal analysis of our solution showing that, in presence of numerous failures, the time to process a transaction is upper bounded.
- An implementation of the proposed failure management algorithm on top of our DTR [17] prototype for transaction routing. An experimental validation in order to demonstrate the feasibility of our solution and measure its performance benefits.

The rest of this paper is organized as follows. We first present in Section II the global system architecture together with the replication model. Section III describes our transaction routing algorithm with freshness control. Section IV deals with node dynamicity. Section V deals with performance evaluation of our failure management mechanism. Section VI presents related work. Section VII concludes.

II. SYSTEM ARCHITECTURE AND MODEL

In this section we describe our system architecture and model.

A. Global Architecture

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

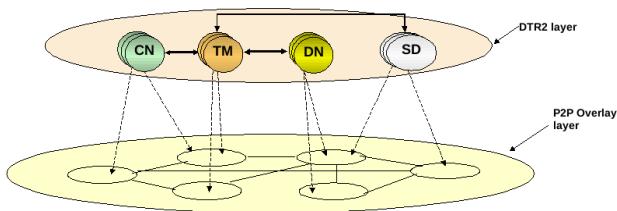


Figure 1. Global Architecture

Our solution is designed to be distributed at large scale over the Internet. It leverages existing P2P services (lower layer) with a set of new functionalities for transaction routing (upper layer). The lower layer is a P2P overlay built on top of the physical Internet network. It gathers nodes into a structured P2P network (node leave/join primitives) and provides

basic services such as node location, node unique identifier support, and inter-node asynchronous communication.

The upper layer, named DTR2, provides advanced data management services: database nodes with transaction support (DN), shared directory (SD), interfacing with client application (CN) and middleware transaction routing (TM). These services are instantiated down to the P2P layer such that each peer node may provide one or more services. The dark edges, in the upper layer, illustrate inter-service communications between nodes. For instance, a TM node can communicate with several reachable DNs (directly or not) through the P2P overlay. A client application (CN) may know several TMs in order to benefit from more routing resources. In the following, we briefly explain each node role:

Client Nodes (CN) send transactions to any TM. A CN assigns to each transaction a global unique identifier GId which is the local transaction sequence number prefixed by the client name.

The CN serves as an interface with the data manipulation procedures of the application such that each transaction call from the application can be intercepted by a CN.

Transaction Manager Nodes (TM) route transactions for execution on data nodes while maintaining global consistency. TMs use metadata stored in the shared directory for routing incoming transactions to data nodes. TMs are gathered into a logical ring [10] in order to facilitate the collaborative detection of failures. This detection allows available TMs to complete any transaction processing managed by a failed TM.

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 (SD) are the building blocks of the shared directory implemented as a distributed index. The shared directory contains detailed information about the current state of each DN, *i.e.* the relational schema of data stored at a DN and the history of recent modifications. We use a DHT to implement the shared directory. Therefore, metadata can be merely distributed and replicated over several nodes in such that scalability and availability are reached. DHT services allow for a fast retrieval of metadata from the database allocation schema. Even though metadata can be accessed concurrently, only a few communication messages between routers (TMs) is needed to keep metadata consistent. The implementation details of the shared directory through a DHT and the metadata consistency control are detailed in [18].

B. Replication Model and Global Consistency

We assume a single database with n relations R^1, \dots, R^n that is fully replicated at m nodes N_1, \dots, N_m . The local copy of R^i at node N_j is denoted by R_j^i 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 updates.

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 (eventual consistency). To achieve global consistency, we maintain a graph in the shared directory, called global precedence order graph. This global precedence graph is distributed on several SDs. 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 [7], is comparable to the one of used in [3].

C. 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.

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) *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.

3) *Failure Detection*: Usually, failures are detected either periodically by heartbeat messages [1], or on demand by ping-pong messages [10]. [6] 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 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. The failure of any SD node is under the control of the DHT which manages it transparently.

III. TRANSACTION ROUTING WITH GLOBAL CONSISTENCY 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 [17] and [7]. Then, we discuss the specific issues raised by the use of a shared directory.

A. Routing Protocol Specification

We describe here the failure free routing case (see Section IV for failure management). By simplifying the approach described in [17], the transaction processing can be split into three phases.

Setup phase: During this phase, a CN sends a transaction to a TM. 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.

Routing phase: TM performs the routing algorithm (see Section III-B) for sending the incoming transaction to a DN.

Execution phase: During this phase, a DN receives a transaction T , runs it, and returns the result back to the corresponding CN. DN also notifies the corresponding TM that T has been processed.

Figure 2 represents the specification of the transaction execution process.

B. Routing Algorithm and Global Consistency

Our routing strategy is cost based and uses late synchronization. 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. When a TM receives a transaction, it asks the shared distributed directory (SD) for the existing precedence constraints related with the transaction T . Then, the TM chooses the data node replica (DN) that minimizes the estimated time to process T .

The routing scenario described above occurs at any TM, each time a TM receives a transaction call from a client.

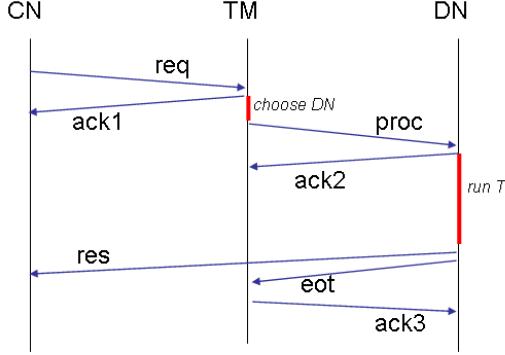


Figure 2. The phases for executing transactions

Thus, two or more TMs may face a concurrent access to the same SD node if their transactions share the same conflict class, if they potentially access the same data. To ensure metadata consistency despite concurrent access, we use a First-In First-Served approach to order TM writes on the SD and to build a correct precedence graph. The details of such a mechanism for ensuring metadata consistency is detailed in [18].

IV. 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.

A. Predictable Disconnection

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

1) *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.

2) *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 prevents 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.

B. Unpredictable Disconnection

In order to deal with unpredictable disconnection, we detail each of the three phases defined in Section III. 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. In the following, we name messages as defined in Figure 2.

1) *Fault-Management from the CN side:* CN emits *req* (it requests a TM to process T) then sets a timer δ_a (see Figure 3). When δ_a elapses, CN concludes that either *req* or *ack1* have been lost due to a communication or a TM failure. Then, CN retransmits *req* with the same global identifier. In order to give the retransmission a successful outcome, the CN increments the number of targeted TMs. More precisely, the CN appends one TM to its destination list, each time it retransmits *req*. 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 DN, thanks to the use of a global identifier and the consistent access to the shared directory.

When a CN receives *ack1* from a TM, it stops any further retransmission of *req* and sets a timer δ_s . When δ_s elapses, CN has not received any transaction result. It concludes that the transaction T is still running or its result has been lost or T has failed. Then, CN transmits *req'* to the previously contacted TMs (*req'* looks like *req*, but also means that CN has already received an *ack1*).

In order to reduce useless and frequent retransmissions, timeout values are based on the network latency and average time to process transactions. Then, $\delta_a \geq 2 * \lambda_N$ and $\delta_s \geq 2 * \delta_a + \lambda_D + Avg(T)$ where λ_N is the network latency, λ_D is the time to read/write the shared directory, and $Avg(T)$ is the average time to process transaction T.

2) *Fault-Management from the TM side:* Upon receiving *req*, the TM replies *ack1* to the CN. Then, TM checks if the transaction T have already ended or is currently running. If T is not mentioned in the shared directory, then the TM routes T to a DN. This avoids performing the transaction twice.

Upon receiving *req'* from CN, three cases are identified depending on the current state of transaction T:

1. If T has already been processed at a known DN_i , then the TM retransmits T to DN_i . This case arises when the transaction result did not reach the CN due to a communication failure.

2. If T is in progress (already routed but not ended), then the TM replies *wait* to CN. This is the case when T lasts

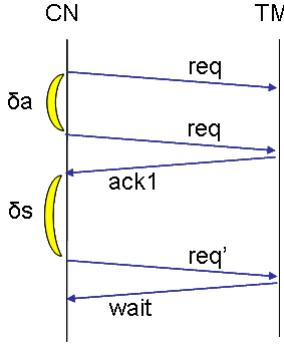


Figure 3. CN behavior depending on timeouts

longer than expected because of a DN failure.

3. If T , is not mentioned in the shared directory, then the TM routes T to a DN. This case arises when the TM fails before choosing a DN, thus before writing on the shared directory.

At each routing algorithm evaluation, the TM keeps the list of the candidate DNs sorted by increasing cost. Then, the TM sends *proc* to the first candidate DN_i , and sets a timer δ_a . Upon receiving *ack2*, TM updates the shared directory to mark T as running. Then it sets a timer δ_r .

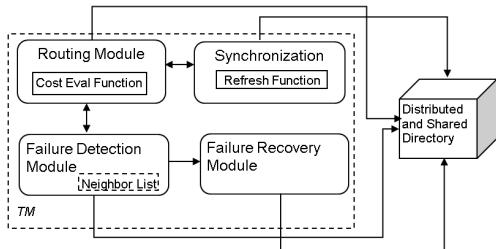


Figure 4. TM Architecture

When δ_a or δ_r elapses, the TM concludes that a failure occurred (communication or DN failure). Then, it sends *proc* to next candidate DN_j on the list (see Figure 5). In parallel, it invokes the *Failure Detection Module* (see figure 4) which checks if DN_i is available or not. To this end, it contacts its predecessors and successors: each of them try to contact the suspicious DN_i by sending it a message. The results of these handshakings are sent back to the initial TM. If all the results are negative it concludes that DN_i has failed and append it to the node failure list, called *NFLList*, stored in the shared directory. Conversely, if at least one of these results is positive, the TM assumes that there is a temporary communication failure between itself and DN_i . Thus, it considers DN_i like a potential candidate for later processing. The failure manager, called *Failure Recovery Module* (see Figure 4), is responsible for checking the recovery of any failed node, and to remove it from the *NFLList* as it is done

in [6].

Finally, when the TM receives *eot* from a DN_i , it updates the shared directory to mark that T has ended on DN_i , then it replies *ack3* to DN_i .

The value of δ_r is proportional to the network latency and the average processing time of T . We set $\delta_r \geq \delta_a + Avg(T)$. To avoid useless messages, we set $\delta_r < \delta_s$ such that a CN will not retransmit a request before the TM has detected a potential DN failure.

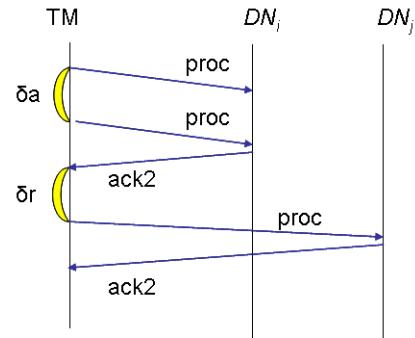


Figure 5. TM behavior depending on timeouts

3) *Fault-Management from the DN side:* Upon receiving *proc*, the DN replies *ack2* to the TM. Then, the DN checks in its log, if it has already run T . If not, the DN runs T . When T terminates, the DN answers *res* (containing the result of T) to the CN that initiated T . If T has already been run, the DN repeats *res* to the CN.

The DN also replies *eot* to the TM, then it sets a timer δ_a . When δ_a elapses, the DN concludes that a failure occurred (communication or TM failure). Then, it adds *eot* into a buffer for later use in a piggybacking fashion while it will send the next notification. This aims to reduce the number of messages sent to the TMs compared to periodic attempts.

Furthermore, we note that if the suspected node has already executed the transaction before falling down, then the execution of T on another DN does not compromise consistency, since the execution of all transactions is done similarly over all the nodes, *i.e.* within a global order of precedence.

C. Analysis of the Disconnection Overhead

As described above, our routing protocol terminates despite nodes disconnections. However, the delay to complete a transaction increases wrt. the occurrence of disconnected nodes. The more attempts are needed to process a transaction, the longer is the time to complete it. Thus, we focus on estimating the number of attempts in order to show that it remains low in most cases. To this end, we note $avg(T)$ the average time to execute a transaction, λ_N the average of the network latency, λ_D the average shared directory access time, and \bar{S} the size of the refresh

sequence (see section III-B). Without any disconnection, the time required to execute a transaction T is: $time(T) = 3 * \lambda_N + (\bar{S} + 1) * avg(T) + \lambda_D$

If a node TM and/or DN involved in processing T disconnects itself, then, in the worst case, the transaction will succeed after k attempts initiated by the CN. Thus let k be the number of attempts, the time to process T is: $time_k(T) = k * time(T) + (k - 1) * \delta_s$

Let p be the probability that a transaction fails (*i.e.* CN did not receive any result) and X a random variable representing the number of attempts. $P(X = i) = (1 - p) * (p)^{i-1}$ is the probability that the $(i-1)$ first attempts fail and the i th succeeds. The average number of attempts is obtained by:

$$E(X) = \sum_{i=0}^n i * P(X = i) = (1 - p) * \left(\sum_{i=0}^n i * p^{i-1} \right)$$

An upper bound for $E(X)$ is $E(X) < (1 - p) * (\sum_{i=0}^{\infty} i * p^{i-1})$. By replacing $\sum_{i=0}^{\infty} i * p^{i-1}$ by its limit $\frac{1}{(1-p)^2}$, we have:

$$E(X) < \frac{1}{(1 - p)}$$

Then, we can approximate the number of attempts: $k \approx \lceil \frac{1}{1-p} \rceil$. For example, $k=2$ for a probability of failure less than 50%. Therefore, the routing protocol overhead is rather low in our context.

V. VALIDATION

This section presents the experimental validation of our failsafe routing solution.

A. Prototype implementation

We have implemented all the components (CN, TM, SD, DN) of our architecture, using the Java 1.6 language (5000 lines of code). Each component node is a standalone java application that can be replicated as many times as necessary, and run on any machine. The P2P communication layer between nodes relies on the FreePastry overlay. For better code reusability, we designed the failure-tolerant routing protocol as an enhanced subtype of a general routing protocol.

Following the FreePastry simulation approach, our prototype allows for two distinct communication modes. The nodes communicate each other, either using TCP sockets, or locally simulated through inter-thread messages invocation. First, we used the real communication mode to measure the overhead of failure management on transaction response time. Second, we used the local mode to measure the performance limits of the transaction routers at large scale.

B. Tuning the timeout

First, we aim to adjust the timeout to an optimal value. We vary the timeout from 10ms to 10s. We report the transaction response time on Figure 6, when 2CNs, 2TMs and 10 DNs are running. We observe a decreasing response time (around

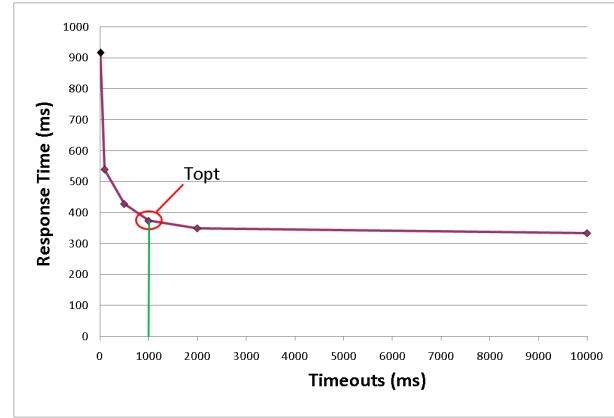


Figure 6. Response time vs. Timeouts

330ms) as expected. We look for T_{opt} that is the lowest timeout value such that no retransmission is performed. This corresponds to the breakpoint after which the timeout remains almost constant (*i.e.* less than 10% from the lower bound). We point out $T_{opt} = 1$ s. We argue that this value is a good tradeoff between, on one hand, a low overhead failure-tolerant routing solution, and on the other hand, a responsive solution that detects failures fast enough to reduce the average response time of transactions facing failures. We note that we could easily adjust T_{opt} dynamically to follow the failure rate evolution.

C. Failure management overhead

Since the optimal value of timeout is found, we measure the overhead yield by failure management. We compare our solution (DTR2) with a baseline routing protocol (DTR) that follows the same routing steps but without dealing with failures. When no failure occurs, we measure the variation in transaction response time between DTR2 and DTR. We deployed our prototype on 20 desktop PCs from our laboratory LAN, running one node per machine to avoid any slowdown due to physical resource sharing between nodes. The workload comes from two clients, each of them sending a sequence of transactions. Transactions are SQL update statements. Each DN node is connected to a backend Postgresql database that runs the transactions. Each experimental run is repeated 5 times for better confidence. Then, we measure the average transaction response time. We vary the number of DN nodes from 2 to 10. Figure 7 shows that DTR yields an average transaction response time around 150ms, almost constant (only 20ms more with 10 DNs than with 2 DNs) while DTR2 yields slower performance.

Figure 7 reveals that DTR2 performs 2 times slower than DTR. Indeed, DTR2 requires extra processing steps to handle failures. For instance, each node is coupled with a timer that signals every end of timeouts. Additional cost comes from indempotence check performed by TM, SD and

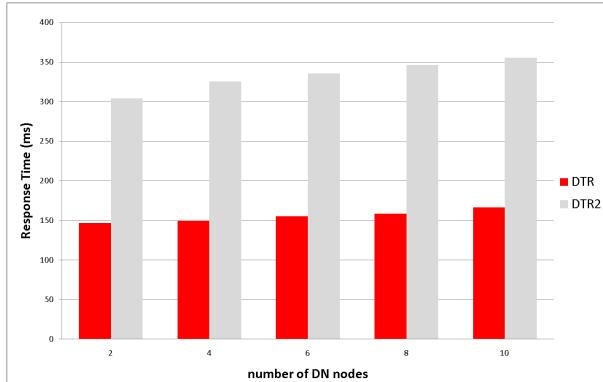


Figure 7. DTR2 vs. DTR overhead

DN nodes to ensure that a transaction is processed only once. This implies the TM to communicate with other TMs to check whether a transaction is already running or not. Moreover, during routing protocol, TM checks if any chosen DN node is available or not. In this experiment, DTR is better because there is no failure which does not reflect a real situation. However, the gain obtained with DTR is lost in presence of failure since transactions handled by failed node are not replayed. We aim to measure this breakdown in next experiment.

D. Fault tolerant routing performances

In order to measure the behavior of our solution at large scale, we instantiate many logical nodes per machine (500 CNs / 500 DNs / 50 TMs). We vary the failure rate of DN nodes from 0 to 100% node failed. The failures occurrence are uniformly distributed over the whole experiment period, i.e. a 100% failure rate means that the first node fails at the beginning of the experiment, the next one fails tf later and so on, with $tf = failurerate * totaltime/numberofnodes$. We report results on Figure 8.

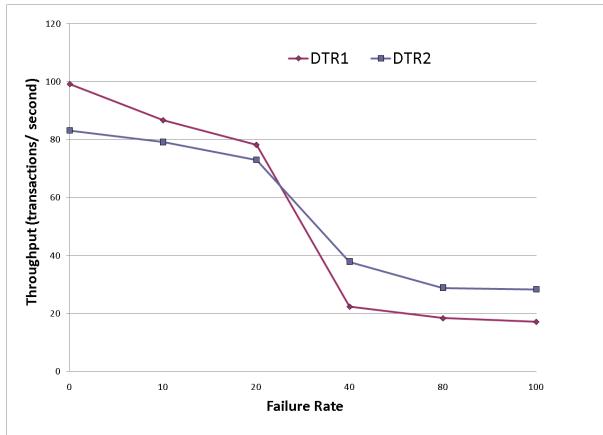


Figure 8. DTR2 vs. DTR Throughput

Results show that with low failure rate, DTR outperforms DTR2 up to 18%. This is due to detection and failure resolution overhead.

Fortunately, beyond 30% of failure rate, DTR2 outperforms DTR by a factor of 36% at least. This gain comes from the failover routing of DTR2 that is able to switch from a failed node to another available node. Moreover any failed DN node is registered at the TM state, thus, avoiding future misuses for new incoming transactions. The failure rate beyond which DTR2 outperforms DTR seems rather high (30%). However, one must keep in mind that it includes predictable and unpredictable disconnections of nodes, as well as network failures.

Indeed, the failure rate beyond which DTR2 outperforms DTR is very important (30%), dealing with failure management is worth pursuing since predictable and unpredictable disconnections added to network failures lead to a higher number of situations interpreted by our approach as failures.

VI. RELATED WORK

Many solutions have been proposed for middleware-based replication and transaction management. Most of them fail to get good performances if participating databases are not available, because concurrency control relies on blocking protocols, and the commit protocol is waiting for a majority of participants to answer.

Ganymed [15] is a database replication middleware designed for transactional web applications. Its main component is a lightweight scheduler that routes transactions to a set of snapshot-isolation based replicas. Updates are always sent to the main replica whereas read-only queries are routed to any of the remaining replicas. However, as [7], Ganymed is not targeted to large scale systems since the middleware is centralized. Even though Ganymed takes care of failed replicas and configuration changes, the use of a single master for all updates makes the solution not suited to update intensive workloads. To overcome this drawback, we designed a multi-master solution.

Middle-R [14] is a middleware oriented solution that focuses on dynamic adaptation to failures and workload variation. Synchronous replication guarantees data consistency. It improves former work on active replication such that [8] and [19].

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[5] 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.

Sprint [4] 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.

While surveying related work, we pointed an interesting approach to recover from failures using replication, in the domain of object interoperability [11]. However this solution is not targeting data management nor transaction processing.

VII. CONCLUSION AND FUTURE WORK

This paper presents the design and the implementation of a distributed transaction routing model. Our solution is designed for large scale and data intensive systems such as web2.0 applications.

This paper extends previous works, DTR and TransPeer, with failure management capabilities. 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).

Currently, we implement DTR2 which is leveraging DTR with failure management. The experimental evaluation of DTR2 shows the effectiveness of our failure management protocol which yields better transaction throughput compared to DTR on a medium-scale. Ongoing tests is conduted to target large-scale configurations. Next, we plan to enhance transaction managers with self-adaptive capabilities and to investigate how our approach will apply to cloud computing. Furthermore, we plan to use a distributed mutex algorithm for reducing SD access time.

REFERENCES

- [1] M. Aguilera, W. Chen, and S. Toueg. Using the Heartbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks. *Theoretical Computer Science*, 1999.
- [2] 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.
- [3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
- [4] L. Camargos, F. Pedone, and M. Wieloch. Sprint : A Middleware for High-Performance Transaction Processing. In *ACM SIGOPS Operating Systems Review, Conference on EuroSys '07*, 2007.
- [5] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. Technical report, ObjectWeb, Open Source Middleware, 2005.
- [6] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2), 1996.
- [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), 2006.
- [8] R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(40), 1997.
- [9] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, 13(1), 1987.
- [10] M. Larrea, S. Arvalo, and A. Fernandez. Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous Systems. In *Int. Symp. on Distributed Computing (DISC)*. Springer, 1999.
- [11] 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.
- [12] E. Pacitti, C. Coulon, P. Valduriez, and T. Ozsu. Preventive Replication in a Database Cluster. *Distributed and Parallel Databases*, 18(3), 2005.
- [13] E. Pacitti, P. Minet, and E. Simon. Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases. *Int. Conf. on Very Large DataBases (VLDB)*, 1999.
- [14] 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.
- [15] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 155–174, 2004.
- [16] R. Shoup. eBay Marketplace Architecture: Architectural Strategies, Patterns and Forces. In *InfoQueue Conf. on Enterprise Software Development*, 2007.
- [17] I. Sarr, H. Naacke, and S. Gançarski. DTR: Distributed Transaction Routing in a Large Scale Network. *Int. Workshop on High-Performance Data Management in Grid Environments (HPDGrid)*, 2008.
- [18] I. Sarr, H. Naacke, and S. Gançarski. Transpeer: Adaptive Distributed Transaction Monitoring for Web2.0 applications. In *Dependable and Adaptive Distributed Systems Track of the ACM Symposium on Applied Computing (SAC DADS)*, Sierre, Switzerland, 2010.
- [19] F. B. Schneider. *Replication Management Using the State-machine Approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.