TransPeer: Adaptive Distributed Transaction Monitoring for Web2.0 applications

Idrissa Sarr
UPMC Paris Universitas
LIP6 Lab, France
idrissa.sarr@lip6.fr

Hubert Naacke
UPMC Paris Universitas
LIP6 Lab, France
hubert.naacke@lip6.fr

Stéphane Gançarski
UPMC Paris Universitas
LIP6 Lab, France
stephane.gancarski@lip6.fr

ABSTRACT

In emerging Web2.0 applications such as virtual worlds or social networking websites, the number of users is very important (tens of thousands), hence the amount of data to manage is huge and dependability is a crucial issue. The large scale prevents from using centralized approaches or locking/two-phase-commit approach. Moreover, Web2.0 applications are mostly interactive, which means that the response time must always be less than few seconds. To face these problems, we present a novel solution, TransPeer, that allows applications to scale-up without the need to buy expensive resources at a data center. To this end, databases are replicated over a P2P system in order to achieve high availability and fast transaction processing thanks to parallelism. A distributed shared dictionary, implemented on top of a DHT, contains metadata used for routing transactions efficiently. Both metadata and data are accessed in an optimistic way: there is no locking on metadata and transactions are executed on nodes in a tentative way. We demonstrate the feasibility of our approaches through experimentation.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Database; H.2.4 [**Systems**]: Transaction processing—*Distributed databases*

General Terms

Performance, Design

Keywords

Replication, Transaction processing, middleware

1. INTRODUCTION

Web2.0 applications such as social networks, wikis and blogs allow users to add and manage contents. They are characterized by (*1*) a read-only intensive workload, (*2*) a number of users very important (tens of thousands), hence a huge amount of data to manage (for instance, Facebook has over five billion photos, and its users upload over 11 million new photos every day), (*3*) various kind of interactive applications such as virtual game, chatting and so on, which need fast processing to reach an acceptable response time (at most few seconds).

Even though smaller than the read workload, the write workload leads in thousands of updates requests per second, thus requiring huge computing resources. Centralized approaches on top of a single parallel (e.g. 128 cores) server are too expensive. Using a data center costs too much as well, in terms of equipment and manpower [6]. Consequently, we aim to design a cheaper solution that runs on a peer to peer (P2P) network of computers. 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. Availability of both data and transaction manager must be ensured even when the underlying nodes leave the network or fail. Dependability is also related with scalability and response time. In Web2.0 applications, most of the transactions are sent on-line and users are not willing to accept long response time which, in some cases, may be considered as a denial of service. On the other hand, many Web2.0 applications are not critical, and users may be likely to trade quality (for instance data freshness) for better response time.

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. Moreover, some replication solutions [13, 11, 12] are not suited to our P2P context since they rely on certification protocols that block in case of too many unavailable replicas. Furthermore, many of these solutions ([13, 10, 5, 8]) are targeting medium-scale systems (*e.g.* PC clusters) with intrinsic reliability, and are no longer suitable for large-scale P2P systems. In consequence, lazy replication appears to be better adapted to our context, since it gives better flexibility to update propagation and thus increases scalability. The

SAC'10 March 22-26, 2010, Sierre, Switzerland.

counterpart is that some nodes can be temporarily stale. If freshness relaxing is accepted by applications, data can be read from any node if its staleness is less than what the application tolerates, improving load balancing [8, 1, 16].

We propose TransPeer, an adaptive middleware-based solution for addressing the problem of transaction management in a large-scale area. To this end, we use some P2P principles such as decentralization and communication between nodes for performing global tasks. In [16], we proposed an approach for managing transactions at large-scale. It ensured global serializability in a pessimistic way. Each transaction is associated with its conflict classes, which contains the data that the transaction may potentially read (resp. write). Conflict classes can be seen as potential read and write sets, usually at the table granularity. They are obtained by parsing the transaction code and are supersets of the read and write sets obtained at execution. Based on conflict classes, transactions are ordered based on their arrival date in the system. Precedence constraints are expressed in a *Global Ordering Graph* (GOG) and the solution ensured that all transactions are executed on different nodes following compatible orders. This approach provides global serializability, but as it is based on potential read and write sets, it may unnecessarily reduce parallelism. For instance, if transactions T and T' potentially write the same table R, they will be executed in the same order on every node. If, actually, T and T' do not access the same tuples in R, they could have been executed in any order on data nodes (causal consistency).

Replication can be implemented either by multicasting every update statement, as in [16]. It can also be implemented by capturing transaction writesets and propagating them after certification [7]. In TransPeer, we use writeset replication (also called *asymmetric replication*) not only to speed update propagation, but also to develop a more optimistic version of our concurrency and freshness control. Potential conflicting transactions are first executed on different nodes before a fine-grain mechanism is used to check real conflicts: in case of a real conflict, one transaction at least is aborted. In any case, the conflict classes are substituted by the read and write sets, so that the global ordering graph gets more accurate. In a replicated environment, we can distinguish two cases of update intensive workloads: a huge volume of updated data or updating few items quite often (hotspot). In a context like Web2.0 applications, the first one happens and users often modify their own data, conflicts are not frequent and the abort rate remains acceptable.

With respect to [16], TransPeer also improves availability of the system through a collaborative detection of failure. For instance, there are several nodes in the system serving as transaction managers. Whenever one of them fails, available transaction managers complete any transaction processing previously managed by the failed node.

The main contributions of this paper are the following :
• 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 both read performances (availability and response time) and load balancing. While [16] was using a pure pessimistic approach, TransPeer uses an hybrid one. Transaction are executed once in an optimistic way, then checked for conflicts (conflict classes are used to reduce the number of conflicts to check, since two transactions may actually conflict only if they potentially conflict). Update propagation is then made in a pessimistic way, according to the ordering graph based on actual conflicts.
• A large-scale distributed directory for metadata. We use a DHT to implement the directory, distributing and replicating merely metadata over several nodes to scale out and to reach availability. 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 is needed to keep metadata consistent.
• An efficient mechanism for rebuilding global ordering graph by exchanging messages with only one node: the latest writer on metadata.
• An implementation/evaluation with thousands of nodes demonstrates the feasibility of our solution and quantifies its performance benefits.

The article is organized as follows. Section 2 describes the basic concepts of TransPeer. Section 3 presents the main algorithm for transaction routing and validation. Section 4 presents a solution to guarantee consistent routing. Implementation issues and experimental validation are presented in Section 5. Section 7 concludes.

2. ARCHITECTURE AND BASIC CONCEPTS

This section describes the global architecture of our middleware solution for transaction routing.

2.1 Architecture overview

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

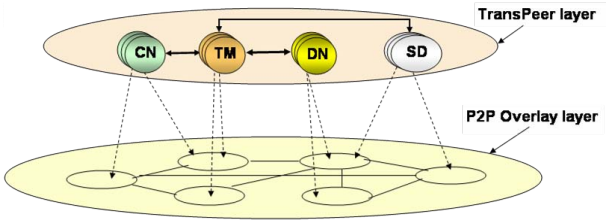

Figure 1: Architecture Overview

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 TransPeer layer, provides advanced data management services: database node with transaction support (DN), shared directory (SD), interfacing with client application (CN) and middleware transaction routing (TM). Theses 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 the transaction 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 [9] in order to facilitate the collaborative detection of failures. This detection allows available TMs to complete any transaction processing managed by a failed TM. Futhermore, in case of DN failures, TMs rely on avalaible DNs for executing current transactions. Consequently, our solution is fault tolerant since it always succeed to run a transaction whenever a DN or a TM failure occurs. Failures are managed in a straightforward way using timeouts, retransmissions, and updates to lists of failed sites (the details of the failure detection and management and its low overhead are subject to another work).

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. We suppose that any DBMS implements a protocol ensuring an ACID execution of local transactions.

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.

2.2 Replication and Transaction Model

We assume a single relational database that is partially replicated at m nodes $N_1, ..., N_m$. Data are partitioned into relations $R^1, ..., R^n$. The local copy of R^i at node N_j is denoted R^i_j and is managed by the local DBMS. We suppose that data are partitioned and partially replicated such that each transaction can be entirely executed at a single node. 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. Lazyness is controlled in such that inconsistencies are hidden to the client according to the transactions requirement We now define, what data a transaction is supposed to access and what data it is really accessing.

***Definition** 1.* We define by $Rel(T)$, the relations names that a transaction T plans to access. We have $Rel(T) = \{Rel_R(T), Rel_W(T)\}$, with $Rel_R(T)$ (resp. $Rel_W(T)$) being relation names that T plans to read (resp. write).

***Definition** 2.* We define by $DataSet(T)$, the set of tuples that a transaction T has accessed. We have $Rel(T) = \{ReadSet(T), WriteSet(T)\}$, with $ReadSet(T)$ (resp. $WriteSet(T)$) being the set of tuples that a transaction T has read (resp. written).

Based on those definitions, we distinguish three kinds of transactions:

• An update transaction is a sequence of SQL statements: at least one of them updates data.

• A refresh transaction is used to propagate update transactions to the other nodes for refreshment. In other words, it applies its WriteSet on another node than the initial one. There is no possible confusion between update and refresh transactions since each DN insures that a transaction is processed only once.

• A query is a read-only transaction. Thus, it does not need to be propagated.

We suppose that $Rel(T)$ is known at the time that T is submitted for routing. *Rel* is obtained by analysing the transactions code. Whereas we assume that any DN can deliver $WriteSet(T)$ or $ReadSet(T)$ at the end of the execution of the update transaction T. Even though we use relational data structure for describing our solution, we claim that our approach is also suitable for others kind of data structure on what update and query operations can be performed through a program.

2.3 Data Consistency Model and GOG Definition

In a lazy multi-master replicated database, the mutual consistency of the database can be compromised by conflicting transactions executing at different nodes. To prevent this problem, update transactions are executed at database nodes in compatible orders, thus producing mutually consistent states on all database replicas (eventual consistency).

To assess compatible orders, we maintain a directed acyclic graph called Global Ordering Graph (GOG) where a vertex is a transaction and an edge is a precedence constraint between two transactions. We assess the precedence constraints from the conflict classes of incoming transactions. Due to precedence transitivity, maintaining the transitive reduction of the GOG is sufficient (*i.e.* an edge from T1 to T2 is not added if T1 already precedes T2 indirectly). The GOG keeps track of the conflict dependencies among active transactions, *i.e.* the transactions currently sent for execution at a DN 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. Later on, at commit time, we check potential conflicts to keep only real ones: if two transactions appear to be actually commutative, we remove the corresponding potential precedence constraint from the GOG. Furthermore, due to database replication, the GOG also contains the edges linking committed transactions that have not been propagated at every replica. This is required for ordering refresh transactions. The GOG is partitioned by relation name to be stored in the shared directory. One partition called $GOG(R)$ includes all transactions accessing R. This facilitates further GOG search by relation name, providing independant access to each $GOG(R)$. This pre-ordering strategy is comparable to the one presented in [2]. The main difference is that the *GOG* is also used for computing nodes freshness.

2.4 Freshness Model

Queries may access stale data, provided it is controlled by applications. To this end, applications can associate a *tolerated staleness* with queries. Staleness can be defined through various measures. Here, 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 transactions 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 refresh transactions do not tolerate staleness. 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. Queries are sent to any node that is fresh enough with respect to the query requirement ; we allow for propagating a sequence refresh transactions (called a *refresh sequence*) on any stale node such that it will become fresh enough for processing a query. 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.

3. TRANSACTION ROUTING PROTOCOL

We first describes the whole transaction routing process starting when a client application generates a transaction call, until it receives the transaction answer. Then, we study the case of concurrent transaction routing that occurs in a decentralized (multi TM) architecture. We state which condition has to be guaranteed in order to keep transaction routing consistent.

3.1 Step by step routing protocol

At the beginning, a client node (CN) sends a transaction T to a transaction manager node (TM). If T is an update transaction, then the following steps occur:

• The TM asks the shared distributed directory (SD) for two kinds of information: the existing precedence constraints related with the transaction T, and the location of every site which content includes all the data that T needs to access. The TM receives from the SD a description of all the candidate sites along with their state mentioning which preceding transactions is already executed (or pushed in the run queue) on which replica.

• Then, the TM chooses the data node replica (DN) that minimizes the estimated time to process T. The cost excludes the time to propagate preceding (wrt. the GOG) transactions that are missing at the replica, since the tentative execution often ends to be enough (no need to process preceding transactions) The estimation relies on the standalone response time of each transaction (collected in advance using calibration). The cost is underestimated when a real conflict occurs, since all the preceding transactions would have to be processed before T. However, this case is rare, thus has minor impact on the overall performances. Then, the TM asks the chosen DN for processing an execution plan (say P) composed of the refresh sequence of every transaction (not yet commited) preceding T, followed by T.

• The DN manages to schedule the execution plan, it tries to run T first, then checks if the data is still consistent. If not, the DN runs the entire execution plan P. Once T has committed, the DN acknowledge both the client and the TM. Upon receiving the DN acknowledgement, the TM notifies the SD about transaction commit.

If T is a read-only query, then Step 1 is the same as above. On Step 2, the plan P includes all the preceding transactions to propagate in order to make the corresponding DN fresh enough wrt. the query requirements. The TM chooses the DN that minimizes the cost of P. On Step 3, the DN always runs the entire plan P.

3.2 Concurrent routing

The routing scenario described above, occurs at any TM, each time a TM receives a transaction call from a client. Thus, two or more TMs may face a concurrent access to the same SD node if their transactions share the same conflict class (*i.e.* are potentially accessing the same data), or to the same DN if the lowest cost node is equally estimated at both TMs.

This requires managing concurrent access carefully. For instance, SD access has to guarantee that any couple of transaction potentially conflicting is detected, that the corresponding precedence constraint is added into the GOG, and that any concerned TM is kept informed. The details on how to efficiently keep the global ordering graph consistent is described in the next section. Furthermore, because of concurrent access to the same DN, two TMs may ask for propagating the same preceding transactions. This situation implies to serialize transaction at each DN and check if it has already been processed.

Because two transactions connected by a precedence constraint may be processed on two distinct nodes, a DN checks at commit time that database consistency is not compromised. The details on how to compute this checking efficiently is detailed in the next section.

4. DEPENDABILITY OF TRANSACTION ROUTING

In this section we focus on the integrity facet of the routing dependability. The main objective is to guarantee that routing transactions can not compromise the overall system consistency. More precisely, consistency must be ensured at two levels:

• At the metadata level, during transaction routing. Preserving metadata consistency requires dedicated struture (Section 4.1) along with customized update methods (Section 4.2), and TM to TM coordination to access up-to-date metadata (Section 4.3).

• At the data level, during transaction execution and commit. Preserving data consistency implies appropriate DN to DN coordination as detailed in Section 4.4.

4.1 Metadata Structure

As pointed out previously, our database is partitioned into several relations. All information related to one relation R is gathered into a structure called $Meta(R)$. It contains necessary information for TM to retrieve the GOG(R) related with a given transaction and to be aware of the current state of replicas.

• The history of all transactions that already updated R but not are yet propagated at every replica. This is the sub-

set of the $GOG(R)$ partition reduced to committed transactions. It also mention at which initial node a transaction has been routed for the first time. We call it $History(R)$. It is a graph where a vertex is a couple (T_i, DN_j) and an edge is precedence constraint.

• The last TM that asked for this $Meta(R)$. This allows for ordering concurrent TMs. It is needed for serializing TMs updating the same GOG, and it keeps the GOG consistent. We call it $Last(R)$.

• The local state of every replica: it is the list of transactions accessing R and already executed on the replica. This allows for assessing the refresh sequence specific for each replica. We call it $State(R_i)$.

The choice to keep the last TM into $Meta(R)$ structure is guided by our design principle to never use any locking mechanism during metadata access. We argue that locks mechanism does not fit to large-scale system, since a node holding a lock can leave the system, and then block all remaining nodes trying to access the same data. However, we still need to control many TMs simultaneously accessing $Meta(R)$ in order to avoid the GOG to be inconsistent, *i.e.* cyclic. In short, the $Last(R)$ allows TMs for rebuilding the complete $GOG(R)$ consistently. See section 4.3 for more details on how the rebuilding is done. When transaction accesses many relations, the correposndig GOG is built by merging the different $GOG(R_i)$. Moreover, relations are ordered based on their identifier (name). Thus when a transaction needs to access several relations, TM retrieves corresponding $GOG(R_i)$ with respect to that order. This ordered access ensures that the GOG remains acyclic, and therefore guarantees metadata consistency. Because of lack of space we do not give more details related to this case in this paper.

4.2 Shared Directory over a DHT

Storing the shared directory on a single node may create a bottleneck and a single point of failure. In our P2P environment, existing solutions for indexing data, such as distributed hash table (DHT) appears to provide interesting functionalities such as resilience to node disconnections. Thus, we choose to rely on a DHT [15] deployed over several SD nodes. The DHT distributes and replicates the metadata to achieve better availability and parallelism for transaction routing.

In the following, we detail the mapping of metadata into a DHT. On one hand we identify the metadata access methods that are natively supported by the DHT. On the other hand, we describe the DHT customization required to achieve metadata update.

4.2.1 DHT Native Support for Metadata Access

DHT provides two useful primitive operations: $put(k, v)$ for inserting a value v associated with a key k, and $get(k)$ for retrieving v associated with k. In our case, k is a relation name R, and v is the $Meta(R)$ metadata. Moreover, in order to tolerate node disconnections, the DHT replicates every (k, v) couple at many nodes. A $put(k, v)$ operation creates n replicas $(k, v_i) 0 < i < n$. The value of n is set at put time according to desired level of availability. Then, every replica is possibly used by the DHT, without any distinction. The DHT only insures that any get operation will find one of the replicas. Thus, two concurrent get operations may find different replicas managed by two distinct peer nodes. The DHT does not care of detecting concurrent get access, because of the read-only nature of DHT data. In consequence, there are only two cases where native put and get operations can be used by the TM. (i) When a TM routes a read-only transaction, it uses the $get(R)$ primitive to obtain $Meta(R)$ including the GOG. The TM does not modify the GOG but only traverses it to compute the refresh sequence. (ii) When a $Meta(R)$ is created and inserted first in the SD. It uses the *put* primitive to insert in the DHT the content of $Meta(R)$. The other cases of metadata access require DHT customization.

4.2.2 DHT Customization for Metadata Update

The two other cases of metadata access are: (i) When a TM routes an update transaction, it reads the GOG with update intent, thus it needs to be informed of concurrent readers, otherwise the GOG may diverge. We call that a get_for_update operation. (ii) When a TM notifies the commit of an update transaction, it must complete the metadata but not fully overwrite it. The $Last(R)$ is not modified because remaining TMs may have accessed the same metadata between TM routing and TM commit notification. The GOG is overwritten. The $State$ is updated: a transaction is appended in list of replica that committed the transaction. We call that a $metadata\_update$ operation.

The implementation of get_for_update and $metadata\_update$ methods is merely based on the DHT and overlay access primitives. Precisely, we slightly modify the replication mechanism provided by existing DHT in such that all nodes keeping a replica of the same data do not play the same roles. Thus, the closest node to a key k, stores the first couple (k, v) and is called the master of v and is used for updates (*i.e.* get_for_update operations). The remaining successors act as slaves and are used for DHT native primitives. Thus, the master node is used for updating metadata and is responsible for synchronizing remaining nodes. The update operation $metadata\_update(v)$ is the sequence of:
$v = get(k), v' = f(v), [remove(k)], put(k, v'), [propagate(k, v')]$.

4.3 Conflict Management at Routing Time

Metadata consistency at routing time is twofold: the former is metadata consistency among the SD at the DHT level; the latter is consistent metadata access from TM. Since the DHT guaranties data consistency among SD nodes, we focus on the latter point. We detail the solution which guaranties that any TM reads the latest version of the GOG and update it consistently wrt. other concurrent TMs. The main concern is that the SD is not able to maintain the latest GOG version (this comes from our choice to prevent any locking). However, the SD maintains sufficient information to make a TM rebuilding the latest GOG easily. This implies the TMs to exchange some parts of the GOG. Thanks to the $Last(R)$ attribute, a TM handling an update transaction, is aware of which TM has previously written on the same GOG.

For example, Figure 2 depicts the scenario for retrieving a complete GOG. Three client applications submit transactions $T1$, $T2$, and $T3$ to routers TM_1, TM_2, and TM_3 respectively. The 3 transactions are updating the same relation R. TM_1 routes $T1$, it asks SD for $meta(R)$ through the operation $getMeta(R)$: GOG and $Last$ are initially empty. Then, TM_2 routes T_2, it asks SD for $meta(R)$, and receives $Last$=TM_1. Thus, TM_2 asks TM_1 for additional GOG information ($getLastEdge(R)$); TM_1 sends back the vertex $T1$. So, TM_2 rebuilds the complete GOG: $T1 \rightarrow T2$. Fi-

nally, TM_3 routes T_3, it asks SD for $meta(R)$, and receives $Last{=}TM_2$. Thus, TM_3 asks TM_2 for additional GOG edge $T1 \rightarrow T2$. Therefore, TM_3 rebuilds the complete GOG: $T1 \rightarrow T2 \rightarrow T3$.

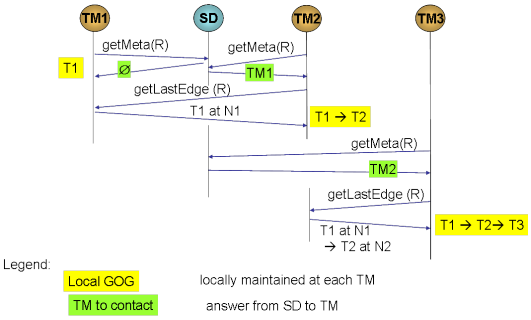

Figure 2: Concurrent Routing Access

In conclusion, despite that the SD delivers a stale version of the GOG, TMs are able to rebuild a consistent GOG, with the limited overhead of one round trip communication. Moreover, we remind that GOG size remains small since any transaction committed at all replicas is removed from the GOG.

4.4 Conflict Management at Commit Time

For better performance, a DN chooses to process transactions in an optimistic way. Suppose two data nodes DN_1 and DN_2. DN_2 receives an execution plan for T_2 composed of the sequence of all the preceding transactions, followed by T_2. The execution plan also informs about the DN (say DN_1) that initially processed the last preceding transaction (say T_1 that directly precedes T_2). DN_2 run T_2. Then DN_2 asks DN_1 for the dataset of T_1 *i.e.* $DataSet(R)$ for each R in $Rel(T_1)$. Hence, DN_2 compares the T_1 dataset with the T_2 dataset. Two cases may occur:

- If there is no conflict, then T_2 commits and DN_2 acknowledges the TM and inform it to remove the edge $T1 \rightarrow T2$ from the GOG.
- Else, once one of the received dataset conflicts with T_2 dataset, T_2 aborts. Then DN_2 runs the entire sequence: T_1 followed by T_2.

The major advantage of this late checking algorithm is that it allows for more parallelism and raises new opportunities to postpone propagation. This is well suited in our application context where most of potential conflicts are not real conflicts. Dataset comparison takes into account all the cases of select/insert/update/delete operations of two transactions for deciding whether they commute or not. Notice that in terms of communication cost between DNs, dataset comparison yields few overhead since it requires only one message from the last preceding DN. In the general case, this messsage contains the datasets from all the preceding transactions.

5. EXPERIMENTAL VALIDATION

In this section, we aim at providing an evaluation of the performances of our solution through experimentation and simulation. To this end, we first check if our middleware approach is not a bottleneck, *i.e.* , it routes every transaction fast enough. Then, we assess the scalability of our solution.

5.1 Experimental and Simulation Setup

Our prototype is written in Java 1.6, and can be run on any system that supports the Java Virtual Machine (JVM) 1.6. Depending on the type of node described in Section 2 (DN, TM, SDN and CN), all components are not executed in the same JVM. As P2P overlay, we use Pastry [14] and we rely on PAST[15] DHT to store metadata.

Our TMs act as a middleware: it provides a transaction processing interface for the applications. We run part of experiments on a real environment with 20 physical computers connected by a network supporting 100 other computers. To study the scalability of our algorithms far beyond 20 peers, we use the FreePastry simulator. As a DBMS, we use postgreSQL and rely on triggers and JDBC for extracting writesets and readsets. For simplicity, data are partionned into relations and each relation is replicated over half of the total DN nodes.

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 TransPeer single router can process up to 50 transactions per seconds. This threshold is satisfying considering that more routers would be able to handle higher workloads.

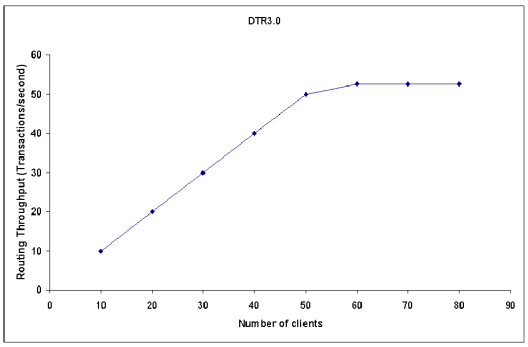

Figure 3: TransPeer Single Router Throughput

Furthermore, we study the impact of multiple routers concurrently accessing the distributed shared directory. The workload is made of the same applications as the former experiment, but the transactions are sent to two routers (such that half of the workload goes to each router). We measure the performance of TransPeer when the conflict rate (proportion of transactions that conflict at least with one other) goes from 0% to 100%. In Figure 4, we observe that the throughput declines steadily as conflict rate grows. When conflict rate is equal to 100%, the overall throughput decreases by 40% from the initial throughput obtained with a single router . This slowdown can be explained by the fact that when TMs access concurrently to metadata, they needs to exchanges some information for building the correct graph, and thereby introduces a delay before routing transactions. However, for the applications we target, the conflict rate is always rather low (less than 10%), and thus

the throughput should not be affected by the concurrency between routers.

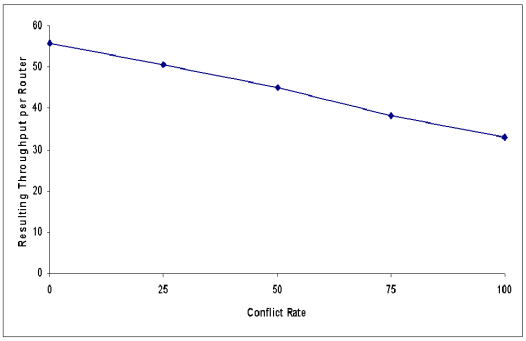

Figure 4: TransPeer Concurrent Router Throughput

5.3 Overall Routing Performance

Once we experiment the routing protocol mechanism, we evaluate the impact of using several routers when there is no concurrency between routers when accessing the directory. We measure the response time when the number of clients is varying from 100 to 1000 whilst the number of TMs move from 2 to 10. Our main goal is to show the benefit obtained by multiplying the number of routers when the workload increases. We use 100 data nodes for all the experiment in order to guarantee that DNs are always available and not overloaded (10 clients/DN), thus TM and SD nodes are the only possible bottlenecks. Figure 5 shows two results. First, it reveals that for heavy workload (1000 clients), response time decreases when the number of used routers grows. With 1000 clients, response time decreases by 23% when the number of TMs grows from 4 to 10. This improvement is reached because higher is the number of TMs, lower will be the time that a client must wait before its request is taken into account. Secondly, Figure 5 determines the minimal number of routers is needed when workload rockets up. For instance, with a workload around 400 to 600 clients, only 8 or 10 TMs are needed to ensure a response time lesser than 2.5 seconds.

Furthermore, we evaluate the impact of varying the number

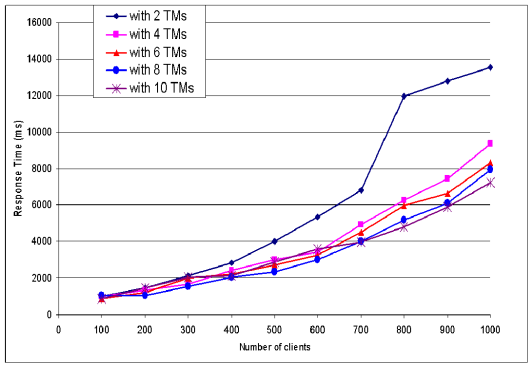

Figure 5: Response Time vs. Number of clients

of DNs (database replicas). To this end, we used 100 TMs to avoid any bottelneck on TM nodes and increased the number of DNs from 10 to 1200. We present the results obtained when workload varies from 1000 to 4000 clients. Figure 6 shows that response time shrinks as the number of replicas grows. With a workload of 1000 clients, the response time is acceptable. Moreover, it remains constant even though DNs grows. With a higher workload, adding new replicas improve significantly the response time until the replication degree reaches some hundreds. For heavy workload, adding more replicas does not increase the throughput because of two main reasons: (1) every replicas is saturated by propagating and applying updates and (2), the number of DNs is so important that TMs loose more time for choosing the optimal DN. In this case, it is thus necessary to increase the number of TMs.

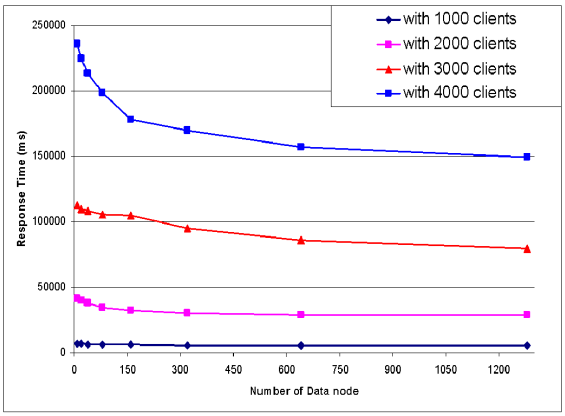

Figure 6: Response Time vs. Number of DN Replicas

6. RELATED WORK

Many solutions have been proposed for middleware-based replication and transaction management. Most of them rely on group communication [13, 17, 18] and are free of any centralized component. However, facing update intensive workloads, these solutions become saturated by resolving conflicts. Moreover, in large scale systems, group communication efficiency suffers from frequent disconnections. Our work differs from theses solutions on two points. First, we leverage conflict management with an optimistic routing strategy coupled with a late consistency check algorithm. Second, with our approach, a transaction is executed and validated completely on a single replica. In other words, we do not need to send a transaction or its writesets to all replicas for checking local conflicts and thus decide to commit or to abort.

Distributed versioning is a middleware based scheduler introduced in [3]. It provides a consistent and medium-scale solution for transaction processing in the context of e-commerce applications. It is based on table versioning combined with a conflict-aware scheduling strategy. Every update transaction is assigned an increasing version number for every table to be accessed. All operations are scheduled in version number order. This implies table level locking which limits efficiency when concurrent accesses are frequent. Moreover, the version manager is centralized which breaks the system availability and limits scalability. In contrast, our solution is free of any locking mechanism.

Some middleware solutions such as Leg@net [8], DTR [16], Ganymed [5], or C-JDBC[4], act as a load balancer or a router by distributing incoming transactions over all repli-

cas. A clear advantage of such approach is the efficient use of all available resources in parallel. Ganymed 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. Leg@net, 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 update intensive applications and keeps the databases autonomous. However, as Leg@net, 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.

C-JDBC[4] is a Java middleware designed as a JDBC driver. Routing strategy aims to be simple and efficient: round robin routing for queries, otherwise, send each SQL statement everywhere. In asynchronous mode, consistency is not guaranteed since the first database that responds is optimistically chosen as a reference, without taking care of the other databases. Thus, this solution is restricted to a stable environment that differs from ours.

7. CONCLUSION

We propose TransPeer a middleware for transaction management in a large-scale area. TransPeer targets Web2.0 applications, where data is shared by a huge number of clients but access conflicts are rare. Thus, it avoids using locks during the routing phase to handle consistency and rather uses an optimistic point of view.

We ensure data consistency through the use of a global precedence order graph expressing potential access conflicts, deduced from the applications, a priori. However, since potential conflicts almost never turn into real ones, we process transactions in an optimistic way, then, we check for real conflicts later at commit time (and update the precedence graph consequently). That allows for more parallelism and raises new opportunities for load balancing.

We describe the transaction processing protocol, with a focus on concurrent access to metadata by several transaction manager instances. With those extensions, performances are improved in terms of parallelism and therefore scalability.

Furthermore, no locking mechanism is needed to ensure consistency when several routers access simultaneously the distributed shared directory. This not only contributes to speed the routing phase, but is also better adapted than locking to P2P environments, where a peer may leave the system at any moment.

An implementation and extensive experimental evaluations of the system show that the (i) overhead due to the distributed directory access is low; (ii) adding more routers allows for supporting increasing workload; (iii) the overall throughput does not suffer from reasonable conflict rate.

Ongoing experimentations are conducted to figure out the limits of the solution in terms of scalability. We also plan to formally prove the correctness of the routing protocol. Finally, we plan to enhance TransPeer with transactions propagation in pull mode, so that data nodes are self-refreshed at the best time wrt. the dynamics of the workload.

8. REFERENCES

[1] F. Akal, C. Türker, H. Schek, Y. Breitbart, T. Grabs, and L. Veen. Fine-Grained Replication and Scheduling with Freshness and Correctness Guarantees. In *Int. Conf. on Very Large DataBase (VLDB)*, pages 565–576, 2005.

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

[3] A. L. C. C. Amza and W. Zwaenepoel. Distributed Versioning: Consistent Replication for Scaling Back-end Databases of Dynamic Content Web Sites. In *Middleware '03: ACM/IFIP/USENIX International middleware conference*, pages 16–20, 2003.

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

[5] 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.

[6] Data-Center-Knowledge. www.datacenterknowledge.com.

[7] C. Emmanuel, C. George, and A. Anastasia. Middleware-based database replication: the gaps between theory and practice. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 739–752, 2008.

[8] S. Gançarski, H. Naacke, E. Pacitti, and P. Valduriez. The leganet system: Freshness-aware transaction routing in a database cluster. *Journal of Information Systems*, 32(2):320–343, 2007.

[9] M. Larrea, S. Arï£¡alo, and A. Fernandez. Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous Systems. In *Int. Symp. on Distributed Computing (DISC)*. Springer, 1999.

[10] Y. Lin, B. Kemme, n.-M. M. Pati and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 419–430, 2005.

[11] E. Pacitti, C. Coulon, P. Valduriez, and T. Ozsu. Preventive replication in a database cluster. *Distributed and Parallel Databases*, 18(3):223–251, 2005.

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

[13] M. Patino-Martinez, R. Jimenez-Peres, B. Kemme, and G. Alonso. MIDDLE-R, Consistent Database Replication at the Middleware Level. *ACM Transactions on Computer Systems*, 28(4):375–423, 2005.

[14] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[15] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 188–201, 2001.

[16] 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.

[17] F. Schneider. *Replication Management Using the State-Machine Approach*, pages 169–197. Distributed Systems (2nd Ed.). ACM Press, 1993.

[18] V. Zuikevičiūtė and F. Pedone. Conflict-aware load-balancing techniques for database replication. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 2169–2173, 2008.