Replicated Database Transactions Processing in Peer-To-Peer Environments

Sofiane Mounine HEMAM[1],Khaled Walid HIDOUCI[2]
[1]LCSI Laboratory
University Centre of Khenchela, CUK
Khenchela, Algeria
s_hemam@esi.dz

[2]LCSI Laboratory
High School of Computer Science, ESI
Algiers, Algeria
w_hidouci@esi.dz

ABSTRACT: *We propose a decentralized approach to processing transactions in a replicated database, under partial replication in Peer-To-Peer (P2P) environments. In order to solve problems of concurrent updates, we propose an architecture based on quorums, this architecture allows assigning a unique timestamp to each distributed transaction in order to built local precedence order graph is from at each peer , to select the servers replicas and to coordinate the distributed execution of the transaction.*

Keywords: Partial replication, Replicated database, P2P, Distributed transaction

Received: 3 January 2011, Revised 9 February 2011, Accepted 15 February 2011

1. Introduction

Nowadays, Peer-to-Peer (hereafter P2P) systems become very popular. This popularity can be seen as a result of the features of these systems such as: scalability, node autonomy, self-configuration and decentralized control. P2P systems offer a good opportunity to overcome the limitations of the Client/Server based systems. By avoiding bottlenecks and being fault tolerant, P2P systems are suitable for large-scale distributed environments in which nodes (interchangeably called peers) can share their resources (e.g. computing power, storage capacity, network bandwidth) in an autonomously and decentralized manner. The more the resources are available in a P2P system, the more the computing power and the storage capacity have important values. This advantage enables P2P systems to perform complex tasks with relatively low cost without any need to powerful servers.

In P2P systems [14], peers are working to achieve specific needs. The needs can be classified into several areas as: file sharing (music or other), distributed computing, distributed storage, communication, and many other areas. As an example of application, the sharing of databases by the doctors for epidemiological purposes, or operating results of teams spread across the globe for researchers in physics. The challenge for these systems is to ensure data availability and consistency in order to deal with fast updates. To solve this problem, systems use expensive parallel servers. Moreover, data is usually located on a single site, which limits scalability and availability. Implementing Database partially replicated onto a p2p system allows overcoming these limitations at a rather low cost. In order to improve data availability, data is partially replicated and transactions are routed to the replicas. However, the mutual consistency can be compromised, because of two problems: concurrent updates and node failures. Another point is that some queries can be executed at a node which misses the latest updates.

Quorum systems are well-known tools that address the first concern. Informally, a quorum system is a collection of subsets of server replicas, every peer of which intersect. Thanks to the intersection property, each subset namely a quorum can act on behalf of the whole replicas group, which reduces server replicas load and decreases the number of messages needed. Similarly, overall availability is enhanced, since a single quorum is sufficient for the server to operate. These advantages were early recognized and formalized into quality metrics quorum size, availability and load which are used to compare various quorum constructions with one another, as described in [1].

The rest of this paper is organized as follows. We first present in Section 2 the global system architecture together with the replication and transaction model. Section 3 describes our transaction processing algorithm. Section 4 presents the performances evaluation of through simulation. Section 5 presents related work. Section 6 concludes.

2. System Architecture

In this section we describe how our system architecture and model are defined. We first present in the section 2.1 the global architecture for better understanding our solution. Then, we describe the replication and transaction model in section 2.2.

2.1Global Architecture

The global architecture of our system is depicted on Figure1. We therefore distinguish five layers in this architecture: Timestamp Manager, Transaction Manager, Coordinator Manager, DBMS, and Database.

- Timestamp Manager. This service assigns a unique timestamp to each transaction.
- Transaction Manager. The function of this service is to analyze a transaction from a client node, to divide it into sub transactions and queries and send them to the concerned servers nodes. It also has to balance the load between servers nodes.
- Coordinate Manager. This service allows it's node to act as coordinator in order to monitor the success of the transaction that has been divided into sub transactions and queries and sent to several different servers nodes.
- DBMS. Each node have its own Data Base Manager System
- Database. Our architecture is based on the database partially replicated, i.e. each node has got a fragment of the database.

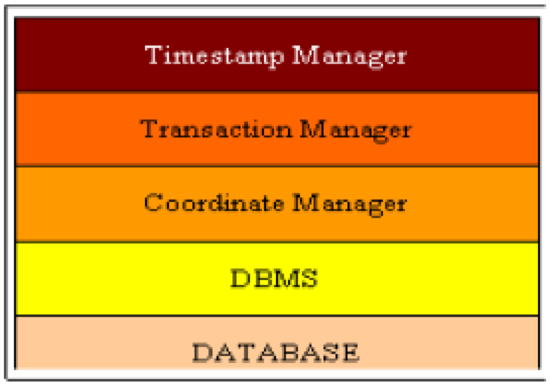

Figure1. Global Architecture

2.2 Replication and Transaction Model

We assume a single database composed of relations R_1,R_2...R_n that is partially replicated over a set S of m nodes {N_1,N_2...N_m}. Each relation R_i is duplicated in a replica-subset S_i ($S_i \subset S$). Moreover, the subsets replicas S_1, S_2, … S_n consist of a partition of S (i.e. $S_1 \cap S_2 \cap \ldots S_n = \varnothing$ and $S_1 \cup S_2 \cup \ldots S_n = S$) (figure 2). The local copy of R_i at node N_j is denoted by R_{ij} and is managed by the local DBMS.

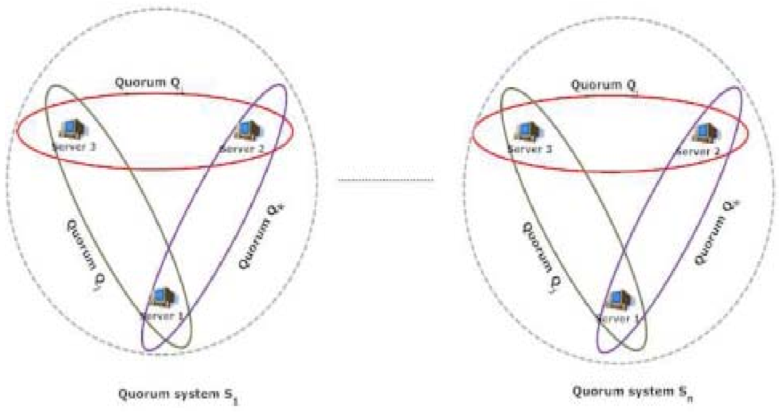

Figure 2. Database partially replicated based on quorum system

Given a set of sites M, a set system universe $Q = \{Q_1, Q_2, ..Q_n\}$ is a collection of subsets $Q_i \subseteq N$ over N. A quorum system defined over N is a set system Q that has the following intersection property: $\forall$ i, j $\in \{1..n\}$, $Q_i \cap Q_j \neq \varnothing$ (figure 3)

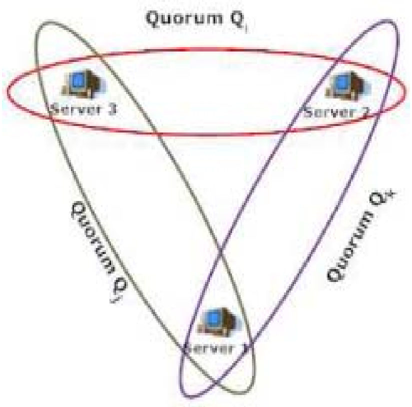

Figure 3. quorum system

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 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.

3. Transaction Processing

In this section, we describe how the transactions are routed in order to improve performance, and we present the timestamping algorithm.

3.1 Routing Process Specification

In this paper, we deal with systems which have two kinds of client node: a novel client node, which wants to join the system for submitting a transaction and the old client which is in the quorum and want to submit a transaction. The processing can be split in three steps.

• Setup phase. During this phase, a client node contacts a quorum of nodes in order to submit a transaction. We assume that any client node knows at least one quorum of nodes. To submit a transaction, the client node must send a transaction, its' identity address, and its category (new or old).

• Routing phase. This phase is performed by the Transaction Manager Layer. When a transaction arrives, the transaction manager verifies the client category. If it is a new one, then, the client is invited to join a quorum selected by the transaction manager, else the transaction will be sent to DBMS layer. This phase is also activated by the node itself, when it wants to send a transaction, the later will be divided into sub transactions and each of the sub transaction will be sent to the respective server node.

• Execution phase. This last phase starts after a server node has received a new sub-transaction and lasts until it sends results to the coordinator layer of the client node. The coordinator of the client node is still awaiting the results (the successful or failure end of execution) of all sub transactions from the servers nodes.

3.2 Timestamp Manager algorithm

In[15], we have described the algorithm handled by the Timestamp Manager layer. This algorithm assigns a unique timestamp to each transaction. Thus, each node has a variable called stamp initialised to zero (0).

We have take into account the concurrency problem due to the presence of several Timestamp Manager, thus to simultaneous access the stamp variable. We have decide to solve this problem using traditional two phase locking (locks on stamp variable are kept until the end of the update this variable). The locks are released rather quickly, because the update of stamp variable is very fast.

The client first selects a quorum of each group, and then it sends a READ message to each server replica in each quorum selected. The READ message contains the request to read stamp value and as well as a lock request. When server replicas receive the READ message, they process the embedded lock request according to the mutual exclusion algorithm. When a server replica eventually grants access to the client, it sends back a STATE message, and considers its stamp variable locked for this client. The STATE message contains the current state of the replica, along with its stamp value. When the client has collected all STATE messages from all quorum selected, it selects the highest value of the stamp variable among the values received from the server replica.

The client updated then the value of its variable stamp, its new value will be the highest value received incremented by one, and then it sends back a WRITE message to each server replica of each quorum selected, which contains the value of its stamp variable, along with a release request. When a server replica receives a WRITE message, it replaces the value of its variable stamp by the new value received and unlocks its stamp variable. Finally, the server replica proceeds to the next pending READ request, if any.

3.3 Transaction Manager Algorithm

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.

To achieve global consistency, we maintain a decentralized graph in each node, called local precedence order graph. This local graph is constructed from the timestamp of each transaction attributed by the Timestamp Manager Layer. 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.

Each node is tagged with a version value denoting its freshness. In order to read a consistent (though) state, the client first selects a quorum (Figure 4.) of each group for each sub queries, and then it sends a READ message to each server replica in each quorum selected. The READ message contains the request to read version value and as well as a request for locking the database for only writing. When server replicas receive the READ message, they send back a VERSION message and locks its' database for writing. The VERSION message contains the current version value of its server replica.

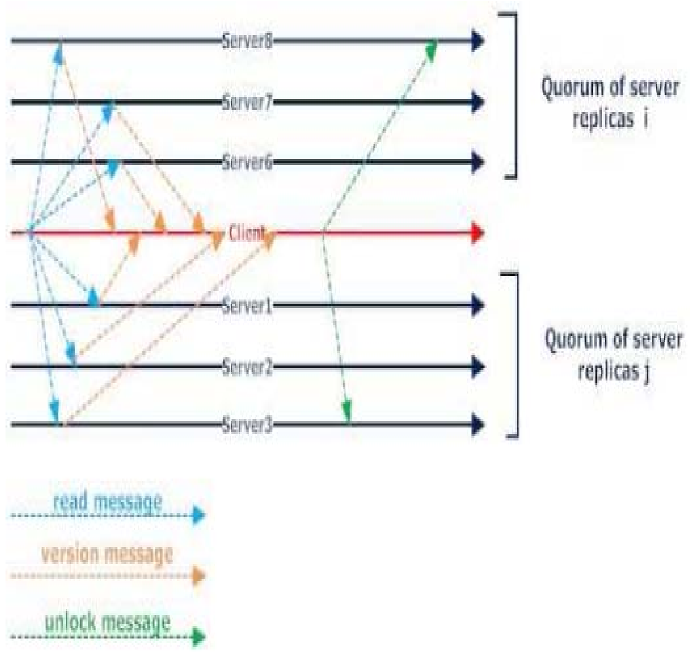

Figure 4. Client interacting with a different quorum of server replicas for selecting the set of the servers replicas

At each time when the client receives VERSION messages from a server replica, it inserts the version value of this server replica in a specific set for the quorum, i.e. the client creates for each selected quorum, a set which contains version values of server replicas of that quorum. When the client has receiving all VERSION message, it retains as the selected value of the data the one with the highest version number of the intersection of its sets. Finally, the client node selects the servers replicas which the value of their version was selected and sent an unlock message to servers replicas not selected.

• Algorithm1. This algorithm runs on both client and server nodes in a different way, because the node is a peer and it can be both client and server. When the node is server replica it can receive two kind of message: READ message and UNLOCK message, and when the node is a client, it can receive the VERSION message.

Algorithm1: runs on server and client node

```
00  $S_S = \{S_1, S_2,, ….,S_n\}$; // set of server replica in each quorum selected//
01  $S_V = \{V_1, V_2,…,Vm\}$ initialized to $\{Ø,Ø,…,Ø\}$; //$V_i$ : is a set of number version of each server replica belonging to the quorum i //
02  $S_Q = \{Q_1, Q_2,…,Qm\}$ initialized to $\{Ø,Ø,…,Ø\}$; //$Q_i$ : is a set of server replica belonging to the quorum i //
03  $S_R$ = nil; // List of clients that have sent a READ message to this client //
04  V: set of number version initialized to Ø;
05  S, $S_{unlock}$: sets of server replica initialized to Ø;
06  H-version: integer;
07  **begin**
08   **while** true **do**
09  **wait** for next message from m;
10   **switch** message type of m **do**
11   **case** READ
```

```
12    if DATABASE is lock then
13       insert $C_i$ in the end of the $S_R$ list; // $C_i$ is a client has
sending READ message//
14     else
15    lock (DATABASE);
16        send-message VERSION (locked, value of its
version);
17      end-if
18     end-case
19    case VERSION
20       inset the version of $S_i$ in the corresponding $V_j$;
21       inset $S_i$ in its' corresponding $Q_j$;
22          $S_S := S_S - \{S_i\}$; // $S_i$ : is a server replica has
sending VERSION message//
23      if $S_S = Ø$ then
24     $V := V_1 \cap V_2 \cap \ldots \cap V_m$;
25      H-Version := highest number version in V;
26     for k := 1 to m do
27             S := S U { a server replica belonging to $Q_k$ and
having H-version};
28     end-for;
29        $S_{unlok}$ := Ss – S;
30     foreach s € Sunlock do
31     send-message UNLOCK (unlock DATABASE);
32     end-foreach;
33   end-if;
34  end-case;
35  case UNLOCK
36   unlock (DATABASE);
37   if $S_R$ not empty then
38   Extract the first client message from the list $S_R$;
39   goto 12;
40   end-if;
41  end-case;
42 end-while;
   43 end.
```

• Algorithm2. In this algorithm, the client sends a VERSION message to all the server replica, and then it waits for the VERSION message from this server replica according to Algorithm 1.

Algorithm2: runs on client node

```
00   $S_Q = \{S_1, S_2, \ldots S_n\}$; // set of server replica in each
quorum selected//
01   S: server replica;
02  begin
04    foreach S € $S_Q$ do
05      send-message READ(read number version, lock);
06    end-foreach;
08    wait for replay from all server replica in $S_Q$;
09  end.
```

3.4 Coordinator Manager Algorithm

In the following, we present an algorithm that is able to enforce globally serializable schedules in a completely distributed way without relying on complete global knowledge. For this, the local precedence order graph is built from the timestamps assigned to transactions by the timestamp manager layer. The nodes of this graph are the transactions and the arcs that connect its transactions are the precedence order. In this graph we define tree kinds of nodes.

• The active transaction node. This node represents the transaction running on the peer and not yet running the commit.

• The ghost transaction node. It indicates that, this transaction is running on another peer, and in this peer, the transaction has no effect, i.e. when the active transaction on a peer has committed its ghost nodes in other graphs will be considered as commit transaction.

• The committed transaction node. This kind of node in the graph represents the committed transaction.

Our system model assumes that there is no centralized component with complete global knowledge, e.g., in form of a global serialization graph, which would allow a global coordinator to easily ensure globally serializable schedules. If this was the case, the global coordinator would easily be able to reject or delay the commit of a transaction if this transaction is dependent on another active transaction. Consider the graph illustrated in Figure 5 –Node i-, and in Figure 5 –Node j-. There is one transaction (T0) which have already committed and some other transaction (T1,…,T7) that are still active. In Figure5 –Node j-, from the set of active transactions, T1, T2 and T5 are the ghost transactions, because they have no effect in this peer and their corresponding active transactions are in another peer (Figure5 –Node i-), and the same principle in the Figure 5 –Node i- but this time the ghost nodes are T3, T4, T6 and T7. When the active transaction T0 in Figure 5 –Node i- is committed, its equivalent in Figure 5 –Node j- which is the ghost transaction T0 will be considered as committed.

At the end of the committed transaction, the coordinator manager must choose the next transaction from its local graph for running, here, we distinguish two cases. The first case is that the next transaction in the local graph is the active transaction, in this case the coordinator manager selects this transaction and it will be executed. The second case is when the next transaction is the ghost transaction, in this case the coordinator manager search in its local graph the first active transaction, this last is selected but its execution will be delayed if this active transaction conflict with the ghost transactions that precede it. e.g. in the Figure 4 –Node j-, when the transaction T0 is committed, the coordinator manager selects T3 because it is the first active transaction in the graph, then it checks if it conflicts with the ghost transactions T1 and T2 which precede it. In this example T3 conflicts with T1, but it does not conflict with T2, in this case the execution of T3 will be delayed until the ghost transaction T1 will be considered commit.

Another important role of the coordinator manager is to contact other peers to ensure the coherence of the replicated database. Indeed, the coordinator manager once it receives the sub transactions of the global transaction and having the same stamp, it analyzes the updates of this sub transaction, if the updates concerns the local database, then the coordinator manager of this peer will act as coordinator, i.e. it will send the sub queries to the selected servers replica, then it waits for their responses. When the coordinator manager receives all the responses, it sends the responses of the sub requests to the DBMS layer for performing the update and then it sends a refresh transaction to the peers of its' quorum. This refresh transaction must have the same stamp as the update transaction.

The protocol relies on the observation that by cooperation, transactions and peers are able to enforce that a transaction does not commit if it depends on another active transaction. At commit time of a transaction, it must be ensured that the transaction knows about all conflicts it is involved in. If the transaction depends on any other transaction it must delay its commit until all these transactions have committed. A transaction can get the information about these transactions from the peers on which it has invoked services. At service invocation time, the corresponding peer can determine the local conflicts using its local log. If a conflict occurs, the peer sends the information about the conflict to the transaction together with the result of the service invocation. In this way, each transaction knows exactly about the transactions it depends on.

4. Experimental Evaluation

In this, section we validate our approach through simulation by using Peersim [16]. Peersim is a P2P system simulator developed with Java. We have extended Peersim classes in order to implement our experiments.

The results obtained in Figure 6 shows that the quorums system (red line) is a well system for a partial replicated database compared to the decentralized system (line blue). We run a set of experiments, varying the number of replicas server from 10 to 60, and we compare the response time in the system. The figure 6 shows that when the number of replica server is 60, the average of the response time is 2,5 in the quorums system and 35,5 in the decentralized system.

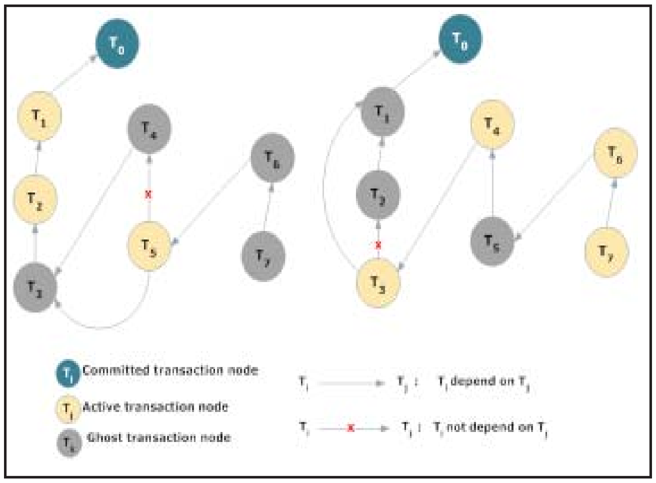
T_0
T_1
T_2
T_3
T_4
T_5
T_6
T_7
T_0
T_1
T_2
T_3
T_4
T_5
T_6
T_7
T_i Committed transaction node
T_j Active transaction node
T_k Ghost transaction node
$T_i \rightarrow T_j$: T_i depend on T_j
$T_i \xrightarrow{\times} T_j$: T_i not depend on T_j

Figure 5. Local precedence order graph at node (i) and at node (j)

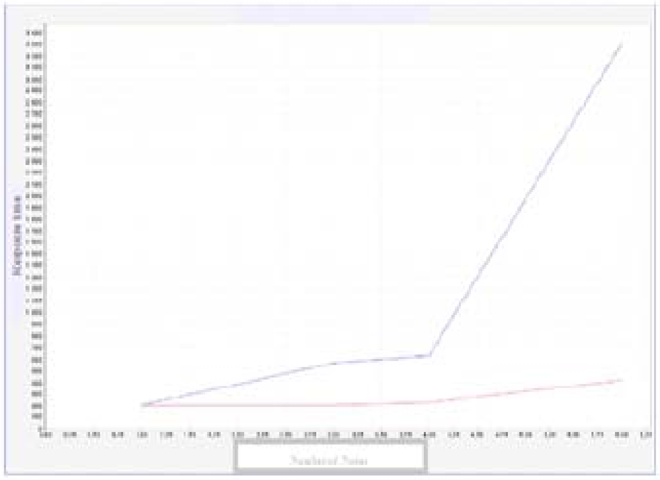

Figure 6. Quorums system vs. Decentralized system

The figure 7 shows the average response time under increasing number of quorums. In this simulation we have fixed the number of nodes to 200 in each sub-set and at each simulation we increase the number of quorums from 3 to 9 but the number of nodes is the same i.e. is 200. The results of this experimentation shows that when the number of quorums increase, the average of response time decreased i.e. when the number of quorums is 1 the average response time is 34P2P and when the number of quorums became 9, the average time response became 17

5. 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 multimaster replication. Second, replica update strategy is either synchronous or not.

In the Database State Machine approach [2] applies update values only but in a fully replicated environment. Its extensions [3, 4] to partial replication require a total order over transactions.

Committing transactions using a distributed serialization graph is a well-known technique [5]. Recently Haller et al. have proposed to apply it [6] to large-scale systems, but their solution does not handle replication, no faults. [7] Present an algorithm for replicating database systems in a large-scale system. The solution is live and safe in presence of non-byzantine faults. Their key idea is to order conflicting transaction per data item, then to break cycles between transactions. This solution either achieves lower latency and message cost, or does not unnecessarily abort concurrent conflicting transactions.

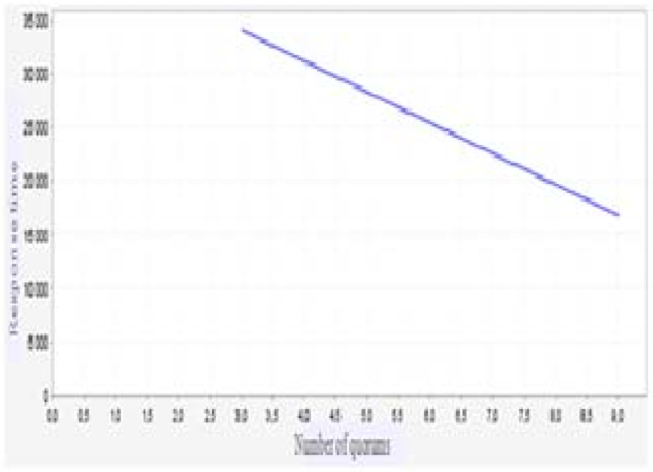

Figure 7. Average response time under increasing number of quorums in a subset

Committing transactions using a distributed serialization graph is a well-known technique [5]. Recently Haller et al. have proposed to apply it [6] to large-scale systems, but their solution does not handle replication, no faults. [7] Present an algorithm for replicating database systems in a large-scale system. The solution is live and safe in presence of non-byzantine faults. Their key idea is to order conflicting transaction per data item, then to break cycles between transactions. This solution either achieves lower latency and message cost, or does not unnecessarily abort concurrent conflicting transactions.

A fully decentralized approach to transaction management in peer-to-peer environments [8] is well-suited especially for long running transactional processes. The protocol is based on distributed serialization graph testing. Each transaction owns a local serialization graph that reflects the conflicts it is involved in. Based on its not necessarily up-to-date serialization graph, each transaction is able to decide on its own whether it is allowed to commit without violating the global correctness criterion of conflict-serializability

DTR [9] is a solution which takes into account the problem due to concurrent updates and controls the freshness in order to improve performances. DRT2 [10] is a new system relying on the Leg@net [11] approach to deal with transaction routing at a large-scale. It also extends [9] for dealing with node failures (or dynamicity) which are very frequent in large scale systems. Pastis [12] is a novel peer-to-peer multi-user read-write file system. Unlike existing systems, Pastis is completely decentralized and scalable in terms of the number of users.

Pastis relies on the Past distributed hash table and benefits from its self-organization, fault-tolerance, high availability and scalability properties.

P2P-LTR[13], provides P2P logging and timestamping services for P2P reconciliation over a distributed hash table (DHT).

6. Conclusion

In this paper, we presented a novel architecture for fully decentralized approach to a distributed transaction management in peer-to-peer environments. The proposed architecture is based on timestamping, managing transactions and the construction of the local precedence order graph. The proposed algorithm in the timestamp manager is a distributed solution in a P2P environment which attaches a unique timestamp to each transaction, by which a total order over all transactions is established. The second proposed algorithm concerns the transaction manager, this last analyzes at first the transaction from a client node, to divide it into sub transactions and queries and in order to read a consistent state, and it must selects for each sub queries a server replica from a quorum of each group.

Finally, the coordinator manager layer, coordinate with others coordinators in order to execute the distributed transaction correctly. This layer receives from transaction manager the selected set of servers replicas and sub transactions and queries having the same timestamp.

In future work, we will integrate a new functionality in this architecture, such as how the coordinator layer detects the failed nodes and how the transaction manager selects serves replicas in order to balance the load between these servers.

References

[1] Naorm, M., Wool, A (1998). The load, capacity, and availability of quorum system, *SIAM Journal on Computing*, 27 (2) 423–447.
[2] Pedone, F., Guerraoui, R., .Schiper, A (2003). The database state machine approach, *Distributed Parallel Databases*, 14 (1) 71–98.
[3] Schiper, N., Schmidt, R., Pedone, F. (2006). Optimistic algorithms for partial database replication, *In*: Proc. of the 10th International Conference on Principles of Distributed Systems (OPODIS'2006), December 2006, p. 81–93
[4] Sousa, A ., Pedone, F., Oliveira, R. Moura, F. (2001).Partial replication in the database state machine". IEEE International Symposium on Network Computing and Applications (NCA'01), p.0298.
[5] Shih, C.S., Stankovic, J.A (1990).Survey of deadlock detection in distributed concurrent programming environments and its application to real-time systems. Technical report.
[6] Haller, K., Schuldt, H., T¨urker, C (2005). Decentralized coordination of transactional processes in peer-to-peer environments, In: Proc. ACM, 14th ACM international conference on Information and knowledge management, CIKM '05, p. 28–35.
[7] Sutra, P., Shapiro, M.(2008). Fault-tolerant partial replication in large-scale database systems, *In*: Proc. LNCS, the 14th international Euro-Par conference on Parallel Processing, 5168, 404–413
[8] Haller, K., Schuldt., Türker, C (2004). A Fully decentralized Approach to coordinating transactional processes in Peer-to-Peer environments. Technical Report, Institute of Information Systems, Swiss Federal Institute of Technology (ETH), Zürich, Switzerland, No. 463.
[9] Sarr, I., Naacke., Gançarski, S (2008). DTR: Distributed transaction routing in a large scale network, *In*: Proc. LNCS, High-Performance Data Management in Grid Environments (VECPAR'08), V. 5336, p. 521–531.
[10] Sarr, I., Naacke, H., Gançarski, S. DTR2: Routage décentralisé de transactions avec gestion des pannes dans un réseau à large echelle. RSTI-ISI in Press.
[11] Gançarski, S., Naacke, H., Pacitti, E., Valduriez, P(2007). The Leganet System: Freshness-aware Transaction Routing in a Database Cluster, *Journal of Information Systems*, 32 (2) April 2007, p. 320–343.
[12] Busca, J-M., Picconi, F., Sens, P.(2005). Pastis: a Highly-Scalable Multi- User Peer-to-Peer File System, *In*: Proc. LNCS, the 11th International Euro-Par Conference (Euro-Par 2005), V. 3648, September 2005. p. 1173-1182.
[13] Tlili, M., Dedzoe, W-K. Pacitti, P., Valduriez, P., Akbarinia, R., Molli, P., Canals, G., Laurière, S (2008). P2P logging and timestamping for reconciliation, *PVLDB Journal* 1(1) 1420-1423.
[14] Androutsellis-Theotokis, S., Spinellis, D. (2004). A survey of peer-to-peer content distribution technologies, *ACM Computer Survey* 36, 4 December 2004, p 335-371.
[15] Hemam, S-M., Hidouci, K-W. (2020). A Fully Decentralized Algorithm to Timestamping Transactions in Peer-To-Peer Environments, *In*: IEEE International Conference on Machine and Web Intelligence ICMWI'10. October 2010, p 164-168.
[16] PeerSim. http://peersim.sourceforge.net/.