Hemam Sofiane Mounine[1], Hidouci Khaled Walid[2]
[1]Computer and Mathematics Department
University of Kenchela
BP:1252 El Houria, Khenchela 40004
Algeria
[2]High School of Computer Science
BP 68M Oued Smar, Algiers 16309
Algeria
{s_hemam@esi.dz, w_hidouci}@esi.dz

JDIM
Journal of Digital Information Management

ABSTRACT: *The need for large-scale data sharing between autonomous and possibly heterogeneous decentralized systems on the Web led to the concept of P2P database systems. In this paper, we present D3-P2P a new architecture which allows to manage a distributed and replicated database, in Peer-To-Peer (P2P) system with high node dynamicity. This architecture is based on a quorum system to solve problems of concurrent update and nodes failure. The proposed architecture allows assigning a unique timestamp to each distributed transaction in order to build a local precedence order graph in each peer, to select the server replicas and to coordinate the distributed execution of the transaction. To avoid deadlock between nodes, we propose a fully distributed algorithm.*

Categories and Subject Descriptors
C.2.4 [Distributed Systems]; Servers: C.2.2 [Network Protocols]

General Terms:
P2P Systems, Data Transfer, Data Transfer Architecture, Distributed Algorithm

Keywords: Partial Replication, Database, P2P, Failure, Node Dynamicity, Deadlock, Quorum System

Received: 11 June 2012, Revised 17 August 2012, Accepted 31 August 2012

1. Introduction

Several definitions about P2P systems exist in the literature [1, 2, 3]. Even if there is no standard definition of these systems, most researchers characterize them by: (1) scalability in terms of node number and resource number; (2) node autonomy; (3) dynamicity, (4) resource heterogeneity, (5) decentralized control and (6) self-configuration. In such systems, each node can act as: (1) a server when it offers its resources to be used by other nodes, (2) client when it uses the resources of other nodes, (3) router when it propa-gates coming queries and messages to other nodes and (4) data source when it shares its own data with the system nodes. The researches on P2P systems have become in-creasingly frequent as well as the contexts in which they are used. In this paper, we focus our study on the P2P database context.

Several application areas need to share their data in P2P systems due to their advantageous characteristics, we can cite: (1) the genetic data management which requires a large storage capacity, and the discovery of a new protein requires a complex analysis to determine its functions and classifications to store it, (2) many researchers around the world, want to share their data about a drug for Hepatitis-C disease, during the duration of an experiment, and many others areas.

In order to more illustrate the P2P database utility, we take as an example the management of patient data by doctors. Each specialist has a group of patients and he manages in his personnel computer the patient data (e.g., name, address, X-rays, prescription, allergy to drugs, history, etc). For most of these patients, the specialist accepts to share their data, but there are always some cases that he doesn't accept to share for different reasons (e.g., part of his research program on a new drug, etc). To more understand the importance of the sharable patient data, we present two cases. In the first case, by making the sharable patient data available to other specialists, it allows them to look for other patients who may have similar symptoms as their own patients, and hence can help them in making better decisions on the treatment (e.g., drugs to prescribe, reactions to look out for, etc). In the second case, assuming that a patient got sick during a trip abroad, the treating doctor needs to access to data located in the origin country of this patient to know his medical history. The treating doctor, as a user must have the ability to link

its database with the databases of doctors who have treated the patient. Here, we think that the deployment of a P2P distributed management system for sharing data allows the doctor to access the desired data at anytime and anywhere, since: (1) any doctor can join/leave the network, (2) nodes have to search for content as in P2P systems, (3) the schema defined by each specialist may be different, (4) there is a need for data management, and (5) each doctor has something to share and is also interested in others data.

Since, each doctor has in his personnel computer a fragment of database, the database is partially replicated onto a P2P system allowing data availability and consistency in order to deal with fast updates 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 update.

The use of quorum system, allows to solve the problem of consistency. Informally, a quorum system is a collection of subsets of server replicas, every pair 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, which are used to compare various quorum constructions with one another, as described in [4].

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 describes the failure model. In section 5, we deal with node dynamicity and in section 6 we propose an algorithm to avoid deadlock. In section 7 we validate our prototype (D3-P2P) through simulation. The related work is presented in section 8. Section 9 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.

2.1 Global Architecture

In [5], we have presented the global architecture of our system. We therefore distinguish five layers in this architecture (figure 1): 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 in order to send them to the concerned server nodes.

Coordinate Manager: This service allows its 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 server nodes.

DBMS: Each node has 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.

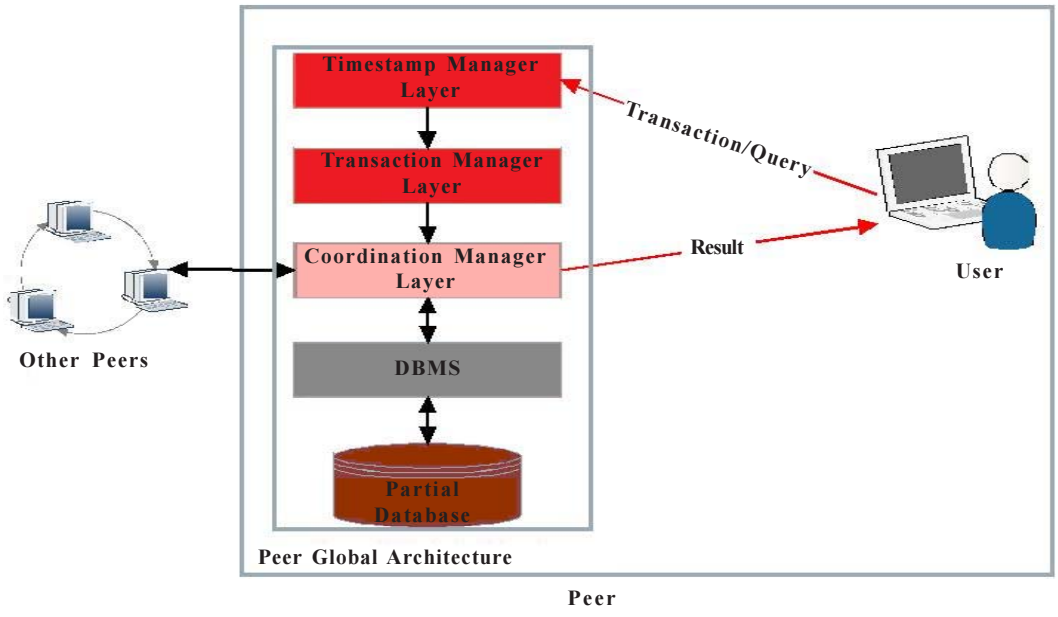

Figure 1. Global Architecture

2.2 Replication and Transaction Model

We assume a single database composed of tables $R1, R2 \ldots Rn$ that is partially replicated over a set S of m nodes $\{N1, N2 \ldots Nm\}$. Each table Ri is duplicated in a replica-subset Si ($Si \subset S$). Moreover, the subsets replicas $S1, S2, \ldots Sn$ consist of a partition of S (i.e. $S1 \cap S2 \ldots Sn = \varnothing$ and $S1 \cup S2 \cup \ldots Sn = S$) (figure 2). The local copy of Ri at node Nj is denoted by Rij and is managed by the local DBMS.

Given a set of sites N, a set system universe $Q = \{Q1, Q2, ..Qn\}$ is a collection of subsets $Qi \subseteq N$ over N. A quorum system defined over N is a set system Q that has the following intersection property: $\forall\, i, j \in \{1 .. n\}, Qi \cap Qj \neq \varnothing$ (figure 3).

We use a lazy multimaster (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 update 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.

3.1 Timestamp Manager algorithm

Before explaining this algorithm, we will give an example to understand its utility. We assume that the node ***N1*** contains the ***patient-not-treated*** table (city, disease, number), and the node ***N2*** is replicated from N1. Also, we assume two transactions ***T1***: 150 new cases of patient having hepatitis-C in Lyon city, and ***T2***: treating 20% of patients of Lyon city having hepatitis-C with X treatment. Initially, we suppose that the tables (in $N1$ and $N2$) contain the following tuple: «Lyon, hepatitis C, 6000».

Node: N1	Node: N2
Initially: « Lyon, hepatitis-C, 6000 »	**Initially**: « Lyon, hepatitis-C, 6000 »
T1: « Lyon, hepatitis-C, 6150 »	**T2**: « Lyon, hepatitis-C, 4800 »
T2: « Lyon, hepatitis-C, 4920 »	**T1**: « Lyon, hepatitis-C, 4950 »

Table 1. Order of transactions execution on nodes

As indicated in table 1, the mutual consistency is compromise, because the order of execution of $T1$ and $T2$ on both nodes is not the same.

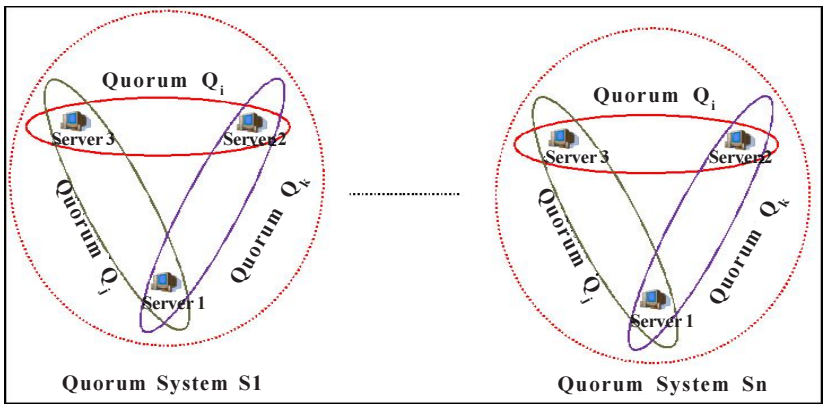

Figure 2. Database partially replicated based on quorum system

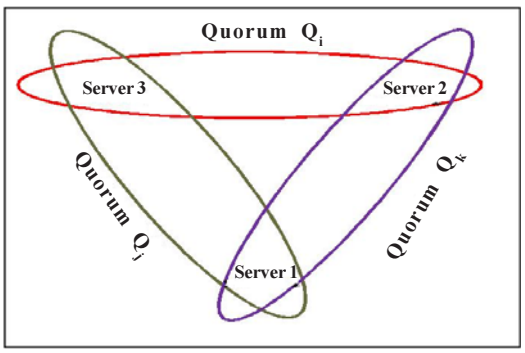

Figure 3. Quorum System

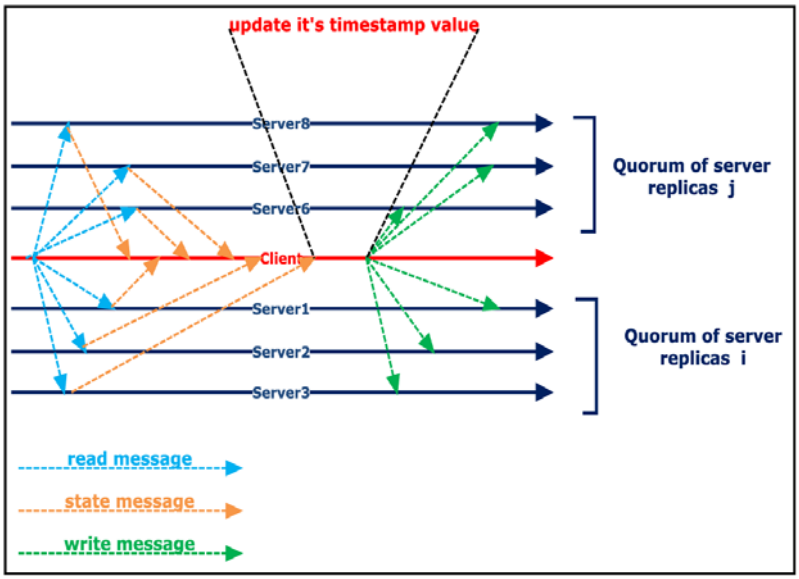

Figure 4. Client interacting with a different quorum of replica servers to update its' timestamp value

To solve the mutual consistency and the update problems, we have described in [5] 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 initialized to zero (0).

We take into account the concurrency problem when accessing to the stamp variable, due to the presence of several Timestamp Manager simultaneously. To solve this problem, we use traditional locking mechanism (locks on stamp variable are kept until the end of the update). The locks are released quickly, because the update of stamp variable is very fast.

First, the client selects a quorum of each group, and then it sends a READ message to each server replica in each selected quorum (figure 4). 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. Eventually, when a server replica grants access to the client, it sends back a STATE message, and considers that its stamp variable is locked for this client. The STATE message contains current server replica state, along with its stamp value. When the client collects, from selected quorums, all STATE messages, it selects the highest value of the stamp variable among the values received from the server replicas. Then, it updates its variable stamp value which, is the selected highest value incremented by one, and it will be the highest one. Next, it sends back a WRITE message to each server replica of each selected quorum, which contains its stamp variable value along with a release request. When a server replica receives a WRITE message, it replaces the value of its variable stamp by the new received value and unlocks its stamp variable. Finally, the server replica proceeds to the next pending READ re-quest, if any.

Algorithm1: The goal of this algorithm is to receive messages from client nodes or from server replica nodes, because the node is a peer and can be both client and server. When the node is a server replica, it can receive two kinds of message: READ message and Write message, and when the node is a client, it can receive the STATE message.

Algorithm1: Timestamp manager algorithm (runs on server and client node)

```
00   S_Q = {S_1, S_2, .S_n}; //set of servers replicas in each quorum selected//
01   S_R = nil; // List of clients that have sent a READ message to this client //
02   Novel-stamp: integer;
03 begin
04  while true do
05  wait for next message from m;
06   switch message type of m do
07   case READ
08    if stamp is lock then
09     insert C_i in the end of the SR list;   //Client C_i has sending READ message//
10    else
11    lock (stamp);
12        send-message STATE (locked, value of its' stamp);
13    end-if;
14   end-case;
15   case STATE
16    stamp := Max (stamp, STATE.stamp);
17   S_Q := S_Q – {S_i}; // S_i: is a server replica has sending STATE message//
18    if SQ = Ø then
19    Novel-stamp := stamp +1;
20    send-message WRITE(Novel-stamp, unlock);
21    end-if;
22   end-case;
23   case WRITE
```

```
24    stamp := novel-stamp;
25    unlock (stamp);
26    if S_R not empty then
27    Extract the first client message from the list S_R;
28    goto 08;
29    end-if;
30   end-case;
31  end-while;
32 end.
```

Algorithm2: This algorithm treats only the case where the client sends a READ message to all server replicas, and then it waits for the STATE message from these server replicas according to Algorithm 1.

Algorithm2: Timestamp manager algorithm (runs on client node)

```
00   S_Q = {S_1, S_2, .S_n}; // set of servers replicas in
each quorum selected//
01  S: replica server;
02 begin
03   lock (stamp);
04   foreach S ∈ S_Q do
05    send-message READ(read stamp value, lock);
06   end-foreach;
08  wait for replay from all replica servers in S_Q;
09 end.
```

3.2 Transaction Manager Algorithm

In a lazy multimaster replicated database, the database mutual consistency can be compromised by execution conflict transactions at different nodes. To solve this problem, update transactions are executed at database nodes in a compatible order. Thus, we will have 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 to which it is sent.

Each node is tagged with a version value denoting its freshness. In order to read a consistent (though) state, we have proposed in [6] an algorithm that allows a client to select the replica servers with the highest version value.

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. 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 to at least one table in common, and at least one of the transactions performs a write on that table.

In order to read a consistent state, a client first selects a quorum (Figure 5.) of each group for each sub queries, and then it sends a READ message to each server replica in each selected quorum. 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 lock their databases for writing. The VERSION message contains the current version value of its server replica. 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 of 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 receives all VERSION message, it retains the one with the highest version number from the intersec-tion of its sets. Finally, the client node selects the server replicas which have as version value the one which has been selected above, and sent an unlock message to the other server replicas.

Algorithm3. 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 kinds of messages: READ message and UNLOCK message, and when the node is a client, it can receive the VERSION message.

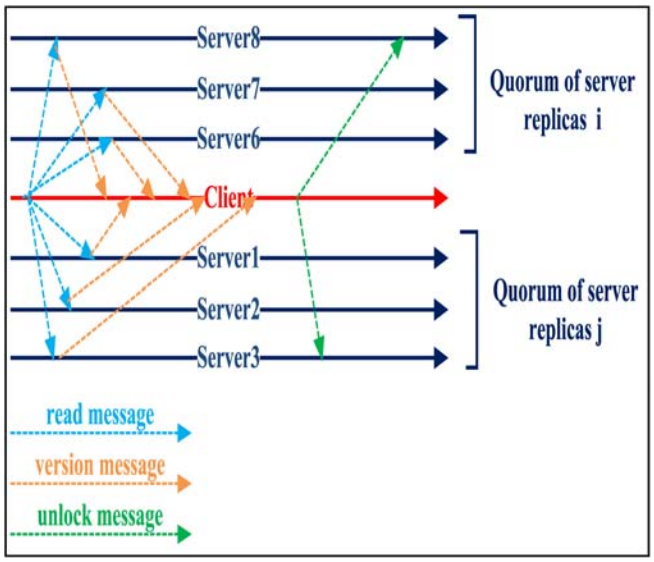

Figure 5. Client interacting with a different quorum of replica servers for selecting the set of the replicas servers

Algorithm 3:Transaction manager algorithm (runs on server and client node)

```
00   S_S = {S_1, S_2,, ….,S_n}; // set of replicas servers in
each quorum selected//
01   S_V = {V_1, V_2,, ….,V_m}initialized to{∅, ∅,…, ∅}; //V_i: is
a set of number version of each replica server belonging to
the quorum i //
02   S_Q = {Q_1, Q_2,, ….,Q_m}initialized to{∅, ∅,…, ∅}; //Q_i : is
a set of replica server 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 ∅;
05  S, Sunlock: sets of replica server initialized to ∅;
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 Si 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 replica server has sending
VERSION message//
23     if S_S = ∅ then
24     V : = V_1 ∩ V_2 ∩… ∩ V_m;
25      H-Version := highest number version in V;
26     for k := 1 to m do
27        S := S U { a replica server belonging to Q_k and having
H-version};
28     end-for;
29      S_unlok := S_s – S;
30     foreach s C S_unlok 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.
```

Algorithm 4. In this algorithm, the client sends a VERSION message to all server replicas, and then it waits for the VERSION message from these server replicas according to Algorithm 3.

Algorithm 4: Transaction manager algorithm (runs on client node)

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

3.3 Coordinator Manager Algorithm

In [7], we have proposed 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. Nodes of this graph are the transactions and arcs that connect its transactions are the precedence order. In this graph we define three kinds of nodes.

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

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 commits, its ghost nodes in other graphs will be considered as committed 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 global serialization graph, the global coordinator ensures easily the serializable schedules. In this case, the global coordinator will easily be able to reject or delay the commit of a transaction, if this transaction dependents on another active transaction. Consider the illustrated graphs in Figure 6 –Node i-, and in Figure 6 –Node j-, there is a committed transaction ($T0$), and some other transaction ($T1$,…,$T7$) that still active. In Figure6 –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 transac-tions are in another peer (Figure6 –Node i-). We have the same case in Figure 6 –Node i-, the ghost nodes are $T3$, $T4$, $T6$ and $T7$. When the active transaction $T0$ in Figure 6 – Node i- commits, then its equivalent in Figure 6 – Node j-, which is the ghost transaction ($T0$), will be considered as committed transaction.

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 execu-tion will be delayed if this active transaction conflict with the ghost transactions that precede it. e.g. in the Figure 6 – Node j-, when the

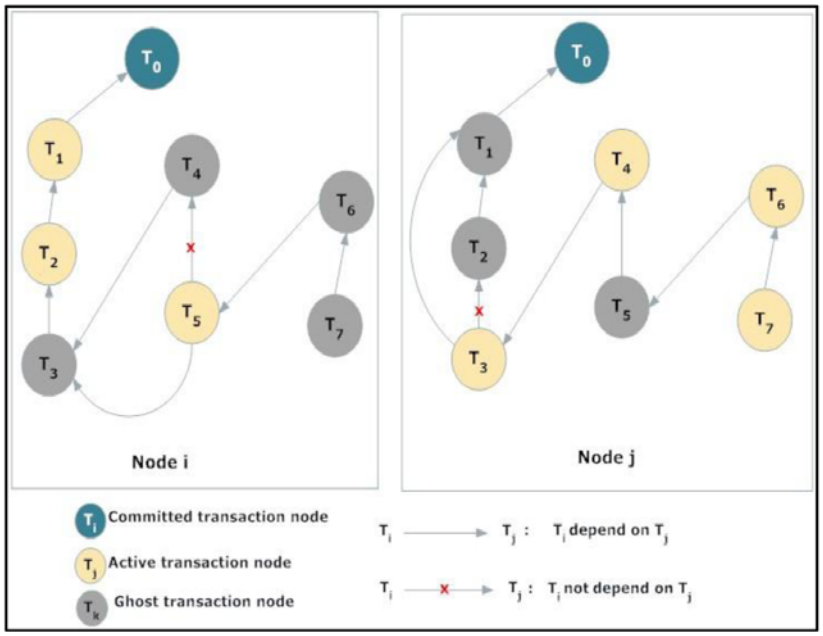

Figure 6. Local precedence order graph at node (i) and at node (j)

transaction $T0$ commits, 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 T, 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 as committed.

Another important role of the coordinator manager is to contact other peers to ensure the replicated database coherence. Indeed, the coordinator manager once it receives the sub transactions and queries of a global transaction, it analyzes the updated sub transaction, if it concerns the local database, then the coordinator manager of this peer will act as coordinator, i.e. it sends the sub queries to the selected replica servers, then waits for their responses. When the coordinator manager receives all responses, it sends them to the DBMS layer to perform the update, and then it sends a refresh transaction to 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. Failure Model

In [8], we have dealt with systems which have only two kinds of components: nodes (CN: Client Nodes and SN: Server Nodes), 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.

4.1 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.

4.2 Communication Failure

A communication failure occurs when a node Ni is unable to contact a 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 which has been 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.

4.3 Failure Detection

Usually, the failures are detected either periodically by heartbeat messages [9], or on demand by ping-pong messages [10]. [11] Presents the principles of collaborative detection targeted to large scale systems. We use ping-pong messages because the number of SN is high, periodic detection would potentially use a lot of network bandwidth, compromising the overall performance. Thus, we decide to detect SN failure without additional cost by integrating failure detection into the routing protocol. A failed SN is detected only when a CN attempts to send a transaction to this SN.

5. Dealing with Node Dynamicity

We describe the approach which is used to deal only with a node when it leaves the system during the execution of a transaction. For that, we assume that any node which joins the system is able to locate one available SN. The contacted SN is then responsible for including it (the new node) in the adequate quorum in order to achieve load balancing between quorums.

We distinguish two situations: predictable and unpredictable disconnection.

5.1 Predictable Disconnection

When $SN_i^{Q_n}$(a server Node i belonging to the Quorum Qn) deliberately decides to leave the system, it sends a Discon-nection request message called **D_Request** with its version number to each $SN_l^{Q_n}(SN_l^{Q_n} \in S.SN_s^{Q_n} / S.SN_s^{Q_n} = S.SN_s^{Q_n} - \{S.SN_i^{Q_n}\}$ and $S.SN^{Q_n}$ is a set of servers nodes at the quorum Qn) and stills waiting for responses. When each $SN_l^{Q_n}$ receives this message, it compares at first the version number of this $SN_i^{Q_n}$ with its own version. Here we can distinguish two cases.

The version of a $SN_l^{Q_n}$ is higher or Equal to the version of a $SN_i^{Q_n}$. In this case (figure 7) the $SN_l^{Q_n}$ removes this from its server available list called **SAList**. This allows to inform any other CN that this $SN_i^{Q_n}$ is disconnected, and sends back to this $SN_i^{Q_n}$ a Disconnection accepted message called **D_Accepted**.

The version of a is lower then the version of a $SN_i^{Q_n}$: Before removing the $SN_i^{Q_n}$ from its server available list, the SN_l sends at first a freshness request (figure 8) called **F_Request** to this $SN_l^{Q_n}$. The $SN_i^{Q_n}$ when receives the **F_Request** messages, sends back to each $SN_l^{Q_n}$ the adequate freshness transaction (**F_Transaction)**. When $SN_l^{Q_n}$ receives the **F_Transaction** message, it runs its freshness transaction, removes $SN_i^{Q_n}$ from its available list and gen-erates the disconnection accepted message (**D_Accepted**) to this $SN_i^{Q_n}$.

In both cases, the $SN_i^{Q_n}$ will be disconnected after receiving all the **D_Accepted** messages from these $SN_l^{Q_n}$, and is, then, considered as disconnected until further notification to join the system.

5.2 Unpredictable Disconnection

The unpredictable disconnection can appear during one of the two phases defined below:

Fault-Management during routing Phase: When the CN_{id} have chosen a quorum Q_n, it sends a transtraction to the selected $SN_i^{Q_n}$ which has the highest version (figure 9). This CN_{id} uses two kinds of timeouts: λ_i^1 for ensuring that the transaction reaches its destination at $SN_i^{Q_n}$ and

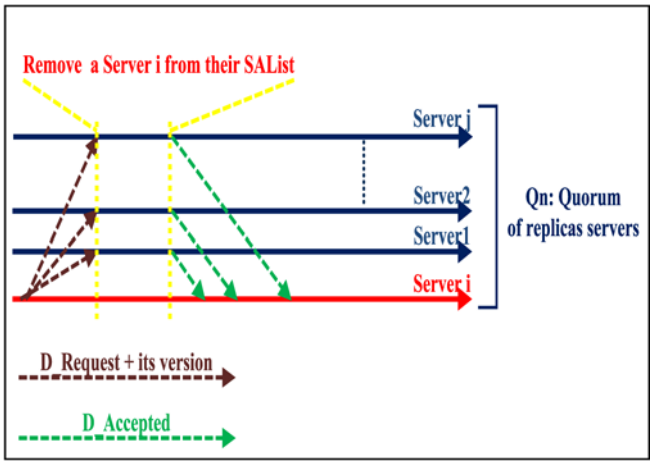

Figure 7. Disconnection of the replica server having a low version

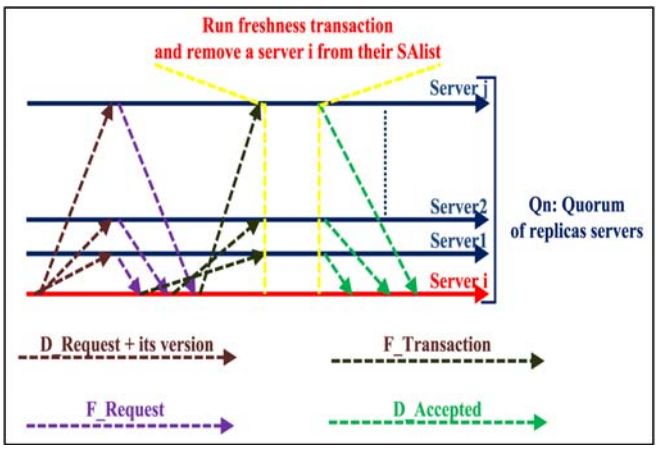

Figure 8. Disconnection of the replica server having a high version

λ_i^2 for waiting the result of transaction execution from $SN_i^{Q_n}$. When λ_i^1 elapses, if there is no acknowledgement from the $SN_i^{Q_n}$ previously contacted, then the CN_{id} deduces that a communication or a failure occurred. Two cases can be identified at this point: (i) The $SN_i^{Q_n}$ was unreachable due to either a communication failure or its own failure. (ii) The $SN_i^{Q_n}$ has received the transaction and has sent an acknowledgement to the CN_{id}. Unfortunately, a communication failure occurred before the reception of the acknowledgement by the CN_{id}.

Both the two cases are treated by the same manner (figure.9), the CN_{id} retransmits the transaction with the same global identifier. In order to give to the retransmission a successful outcome, the CN_{id} transmits this transaction to each $SN_k^{Q_n}$; $(SN_k^{Q_n} \in S.SN_v^{Q_n} - \{SN_i^{Q_n}\} / S.SN_v^{Q_n}$ is a set of servers nodes at the quorum $\boldsymbol{Q_n}$ and they have the same version v as the first selected). In parallel, it invokes the Failure Detection Module which checks if is $SN_i^{Q_n}$ available or not. To this end, it sends a checked failure server message called ***checked_server_fail*** to each $SN_l^{Q_n}$; $(SN_l^{Q_n} \in S.SN_s^{Q_n} / S.SN_s^{Q_n} = S.SN^{Q_n} - \{SN_i^{Q_n}\}$ and $S.SN^{Q_n}$ is a set of servers nodes at the quorum $\boldsymbol{Q_n}$) and stills awaiting the

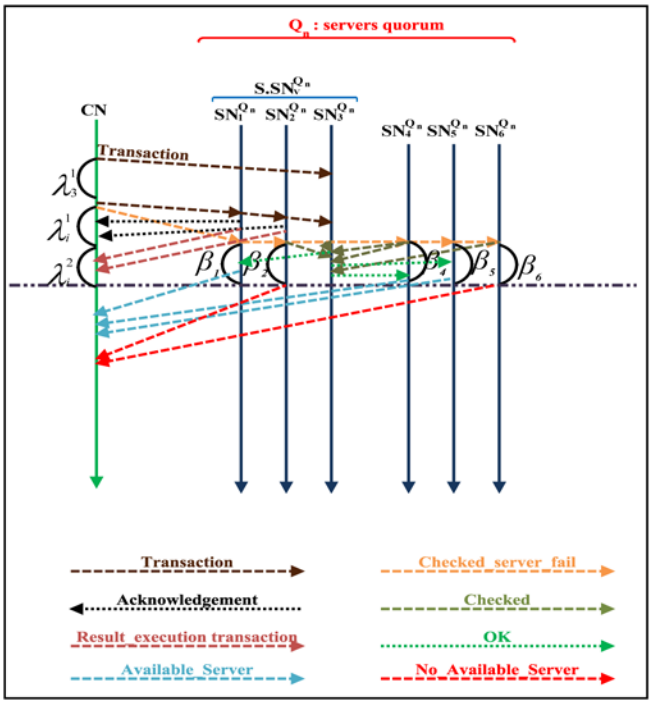

Figure 9. Client checks the availability of a server

responses from each $SN_l^{Q_n}$. Each $SN_l^{Q_n}$(of $S.SN_s$) tries to contact the suspicious node $SN_i^{Q_n}$ by sending it a ***checked***message and sets its timeouts β_i for ensuring that the ***checked*** message reaches its destination at the suspicious node. If the $SN_i^{Q_n}$ not fails, it returns back ***OK*** message to this $SN_l^{Q_n}$, after that, each $SN_l^{Q_n}$ sends ***no_available_server*** *response or* ***available _server*** response to a CN_{id}. In the case when the CN_{id} receives at least one ***available _server*** response, it consider that there is a temporary communication failure between itself and $SN_i^{Q_n}$, otherwise it sends a **failure_server** message to each $SN_i^{Q_n}$in order to allow them to append a $SN_i^{Q_n}$ to their Server Failure list, called **SFList**.

Notice that consistency cannot be compromised since the transaction is delivered to more than $SN_k^{Q_n}$ for execution, because the chosen $SN_k^{Q_n}$ have the same and the highest version.

When the second timeout λ_i^2 elapses, if the CN_{id} has not receive the transaction results, it sends once again the query to $SN_i^{Q_n}$. Two cases can be identified with respect to the retransmission issue: (i) The transaction is in progress, the CN_{id} is invited to wait (figure 10) and to reinitialize its second timeout λ_i^2. (ii) The transaction is executed, but the CN_{id} failed after the transaction has been committed, thus could not be reached. In this case, the SN_i sends again the results to the CN_{id} (figure 11). In order to reduce useless and frequent retransmissions, timeouts values are based on the network latency and average time to process transactions. Then $\lambda_i^1 \geq 2 * \delta$ and $\lambda_i^2 > \lambda_i^1 + Avg(T)$ where δ is the network latency and $Avg(T)$, the average time to process transaction T.

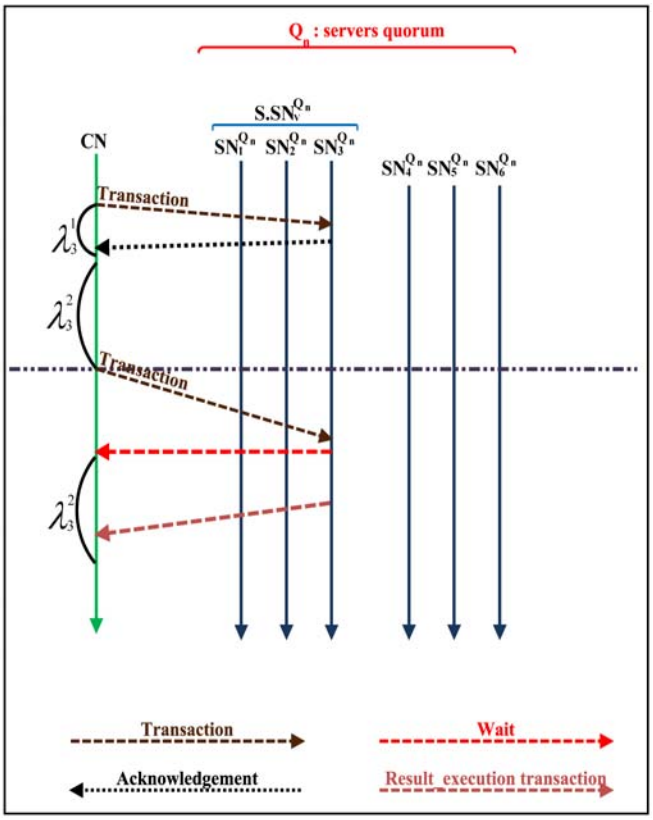

Figure 10. Client is invited to wait the end of transaction execution

Fault-Management in Execution Phase: A $SN_i^{Q_n}$ performs transactions sent by CN_{id}, and notifies the end of their executions. It sends back results or failure execution to the original CN_{id}. After this, the $SN_i^{Q_n}$ initializes a timeout λ_{id} and waits an acknowledgement from the CN_{id}. If $SN_i^{Q_n}$ does not receive this acknowledgement when the λ_{id} expires, it appends this message to a buffer for sending it later. When a CN_{id} receives a failure execution message from a $SN_i^{Q_n}$, it chooses another $SN_k^{Q_n}$ from $S.SN_v^{Q_n}$.

6. Deadlock managing process

In this section, we describe the solution in order to avoid the deadlock in quorum system (Figure 13). The proposed solution is fully decentralized in P2P environment based on quorum system, because the client decides to write access if it collects the majority of the servers vote. Thus, each node handles a clients queue (Figure 12), which contains clients with an associated number of servers vote.

First, the client selects a quorum, and sends a WRITE message to each server replicas of a selected quorum. The Write message contains the request to write or update value in the partial database. When server replicas receive the WRITE message, they process to send back one kind of message. When a server replica receives more

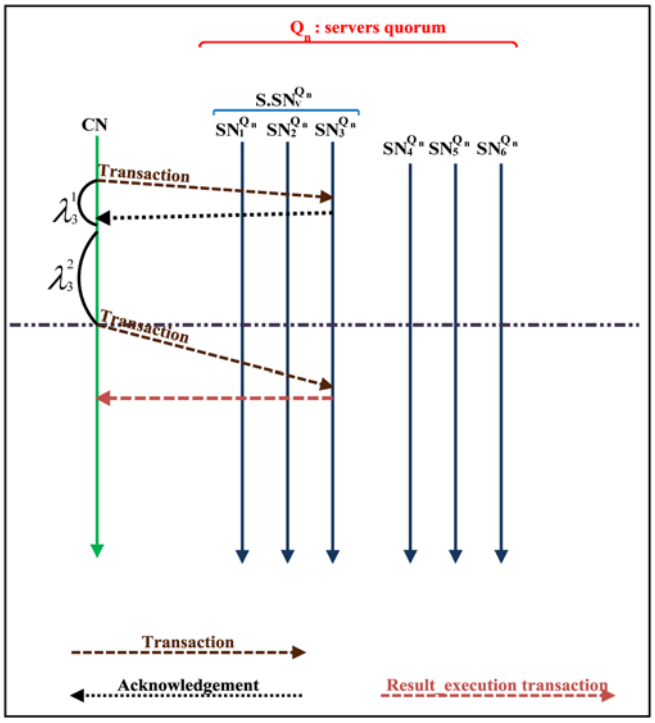

Figure 11. Retransmission the result of the execution

Figure 12. Clients queue

than one WRITE message, it send back an ACCEPT message to the first client and REFUSE messages to the other clients (figure 13). The REFUSE message contains the ID of the client which has received the ACCEPT message from this server replica. When the client receives the ACCEPT message from server replica, it increments its NBR-Of-ACCEPT value, and when it receives the REFUSE message from a server replica, it inserts the ID of the client (which has received an ACCEPT message from this server replica) in the clients queue, if it is not already in and initializes the NB-server to one (01), otherwise, it increments the NB-server value of the founded ID-client.

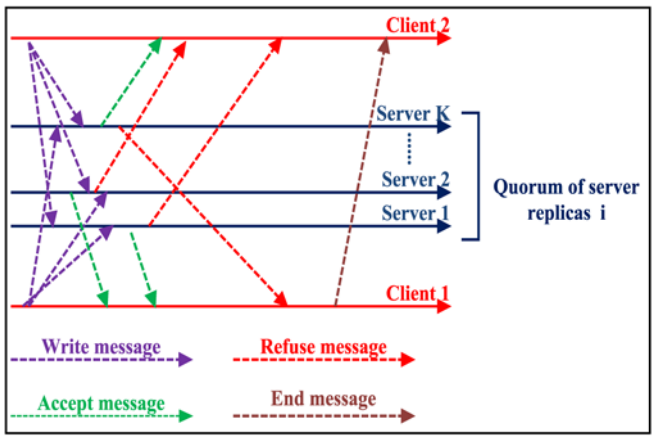

Figure 13. Clients interacting with replica servers of a quorum to prevent deadlock

When the client collects all messages (ACCEPT and refuse message) from server replicas, it compares its NB-server value with those of the clients which are saved in its clients queue. We can distinguish two cases.

In the first case, when the NB-server value of this client is the highest, this client decides to write access the table (if it is the unique client having the highest NB-server value else, if there are several clients having the same highest value, the client which have the lower ID-client decides to write access). In the second case, when the NB-server value of this client is not the highest one, this client must wait the END message from the client (having the highest NB-server value), to remove it from its clients queue and repeats the comparison until it became the selected client to write access.

Algorithm 5: Algorithm to avoid deadlock in quorum system

```
00  S_S = {S_1, S_2, ….,S_n}; // set of replica servers in each quorum selected//
01  NB-server: Number of server having send an accept message initialized to 0;
02  L-C: list clients having received an accept message from server replicas;
03 begin
04  while true do
05      Sends WRITE message to all servers replicas of the selected quorum
06   Repeat
07    wait for receiving message from replica server;
08    switch message type do
09     case ACCEPT message
10       NB-server := NB-server +1
11     end-case
12     case REFUSE message
13          ID-client := extract ID client from REFUSE message
14       if ID-client not exist in L-C then
15          inset L-C.ID-client;
16          L-C.NB-server: = 1;
17       else
18          L-C.NB-server := L-C.NB-server + 1
19       end-if;
20     end-case;
21  Until receiving message from all servers replicas
22  while true do
23      compare the NB-server value of this client with the others in L-C list
24      If this client is the unique having the highest NB-server Then
25        break
26     end-if
27      If this client is not the unique having the highest NB-server Then
28          if the ID of this client is the lower compared to the others having the same
29           NB_server then
30            break
```

```
31            else
32                waiting END message from client
33            end-if
34          end-if
35      end-while
36        write access
37            sends END message when finishing write
 access
42    end-while;
43  end.
```

7. Validation

In this section, we validate our approach through simulation by using Peersim [24]. Peersim is a P2P system simulator developed with Java. We extend Peersim classes in order to implement our experiments.

The experiments follow three goals. First, we need to prove that the distributed database over P2P system based on quorum is better than the distributed database over pure P2P system. Second, we want to show if the increasing number of quorum improves transaction response time. Third, we want to see the nodes disconnection effect on the response time.

7.1 Experimental setup

The experiments run on an Intel Core 2 Duo, 2.4GHz, and Windows machine equipped with 4GB main memory. Every result is the average of about 10 runs.

In this experiment, we propose the following database (cf. Table 2):

Since each doctor has in his personal computer only data about his patients and his research, we fragment this database (table 2) over nodes (i.e., doctor personal computer) and each one of these nodes is replicated to ensure data availability. Finally, we run some transactions and queries, to obtain and analyze experience results.

Table-Name	Attribute
Doctor	Doctor-Name, address, phone, specialty
Patient	name, address, X-rays, prescription, allergy to drugs, history
Patient-not-treated	city, health, number
Doctor-research	health , treatment, dosage, result

Table 2. Database

Since each doctor has in his personal computer only data about his patients and his research, we fragment this database (table 2) over nodes (i.e., doctor personal computer) and each one of these nodes is replicated to ensure data availability. Finally, we run some transactions and queries, to obtain and analyze experience results.

7.2 Comparison between P2P system based on quorum and pure P2P system

In the first test, we run a set of experiments, varying the nodes number from 100 to 600, and we compare the response time in both P2P system based on quorum and pure P2P system.

Response time in pure P2P system: In this experience, we vary the nodes number from 100 to 600, and we obtain response time of each experience.

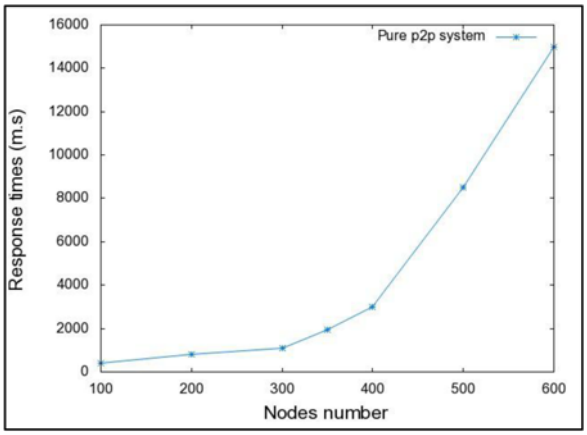

Figure 14. Response time in an unstructured p2p system

The figure 14 shows the simulation result of an unstructured (pure) P2P system. We observe that the response time increases relatively with the increasing of the nodes number. Effectively, the response time average has quickly in-creased from 500 millisecond to 15000 millisecond, when we vary the nodes number from 100 to 600

Response time in P2P system based on quorum: In this experience, we need to observe the response time in P2P system based on quorums. For this, we construct, at first, with each node and its replicas a set of quorums.

In this test, we fix the quorums number of each set to 3, and we vary the nodes number from 100 to 600.

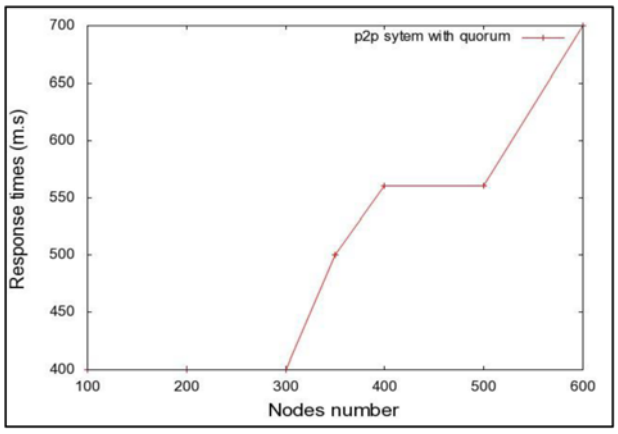

Figure 15. Response time in p2p system with quorum

The simulation result of P2P system based on quorums obtained in figure 15, shows that the response time is the same (400 millisecond), when we vary the nodes number

from 100 to 300. And when the nodes number is varied from 350 to 600, the average response time has slowly increased from 500 millisecond to 700 millisecond.

Discussion: The results obtained from both simulations, shows that the P2P system based on quorums is better than pure P2P system. Indeed, we can see that with our approach (i.e. P2P system based on quorums) we gain 53.44% of the response time compared to the first technique (Pure P2P system).

7.3 Response time with increasing quorums number
In this simulation we fix the number of nodes to 600, and we vary the quorums number from 3 to 9.

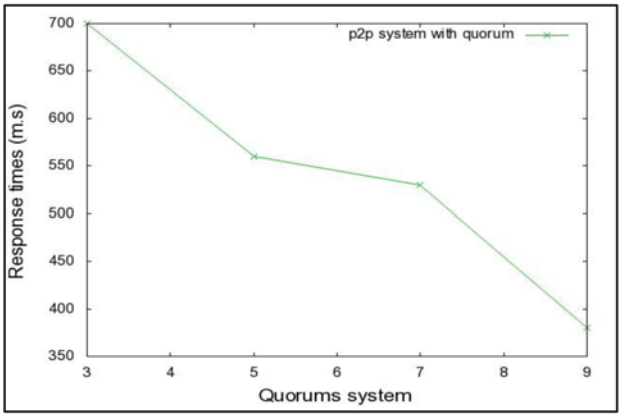

Figure 16. Response time relative to quorum's number

The results of this experimentation (figure 16) shows that when the number of quorums increase, the average response time decreased i.e. when the number of quorums is 3 the average response time is 700 millisecond, and when the quorums number increased to 9, the average time response became 380 millisecond.

Discussion: Effectively, when we fix the nodes number and we increase the quorums number, automatically the quorum size decrease, and therefore, the messages number sent to nodes of this quorum decrease also, resulting a gain in response time.

7.4 Response time with nodes failure
In this experimentation, we fix the nodes number to 600 and quorums number to 3, and we vary the nodes failure rate from 0% to 40%.
Discussion: In this last simulation (figure 17), we can see that the response time increase from 700 millisecond to 1220 millisecond, when we increase the nodes failure rate. This is due to the Failure Detection Module invocation after expiration of or or the both according to the node failure case, which uses additional messages.

8. 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

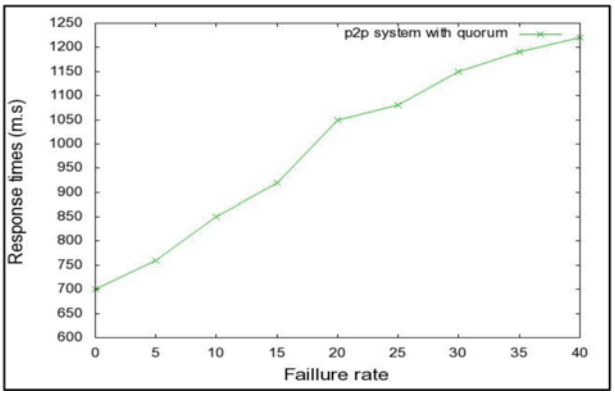

Figure 17. Response time relative to nodes failure rate

upon which these solutions rely. First, replication type depends on the number of updatable copies i.e. either mono or multi-master replication. Second, replica update strategy is either asynchronous or not. Most of existing solutions have functionalities similar to our solution. We summarize related solutions before comparing them.

Middle-R [12] is a synchronous replication middleware that ensures a data consistency. It focuses on dynamic adaptation to failures and workload variation. It improves previous work on active replication such as [13] and [14]. However, to synchronize replicas, Middle-R uses group communications that are costly, especially for large-scale networks.

C-JDBC [15] is a replication middleware for DBMS cluster and it is conceived as a JDBC connector. It allows to the user the transparent transactions processing. The routing strategy is simple and efficient: each query is sent to a different DBMS in round robin routing and each transaction is delivered to all DBMS. Consistency of replicas is not guaranteed because the first database replica that responds is designated for reference without taking care of the other databases replicas. Thus, this solution is restricted to a stable environment.

FAS [16] is an asynchronous and mono master middleware. It takes into account the nodes freshness in order to ensure transaction's requirements. It sends the transactions to a master node and queries to the least loaded node. It propagates updates to the slave nodes by deferred refresh transactions. FAS is mono master configuration, it does not supports an update-intensive applications. Moreover, if no node is fresh enough, the query will wait for a node to be refreshed, which can cause overloading a node when it becomes available to treat the waiting queries. In this case, immediately refreshing an idle node would have been beneficial.

In [17], authors deal with version of data in structured peer-to-peer system based on a distributed hash table (DHT). The mutual consistency is guaranteed via a timestamp service, fault-tolerant, which allows to find efficiently the current version of a replica. However the availability of nodes storing the data is not considered. An incon-

sistency may occur if a node that stores the latest version of a data leaves the system before propagating the newest data to the others nodes.

The RepDB [18] is a fully decentralized solution for managing replicated database. It supposes reliable communications to preserve messages order. The strong consistency of data is guaranteed by performing transactions according to their timestamp. A transaction can be committed after a delay which depends on network latency. Therefore, this solution is restricted to cluster with reliable networks where communication times are bounded.

Existing Solution	Autonomy	Consistency enforce	Freshness	Load Balancing	Availability	Scale
Middle-R	Y	Strong	N	Y	Y	M
C-JDBC	Y	Week	N	Y	Y	M
FAS	N	Outdated Read	Y	Query Only	N	M
Data Currency	Y	Week	N	N	P	M
RepDB	N	Outdated Read	N	N	N	M
Sprint	N	Strong	N	N	Y	M
Leg@net	Y	Outdated Read	Y	Y	N	M
DTR	Y	Outdated Read	Y	Y	Y	L
DTR2	Y	Outdated Read	Y	Y	Y	L
Pastis	Y	Outdated Read	Y	N	Y	L
D3-P2P	Y	Strong	Y	N	Y	L

P: Partially, M: Medium, L: Large

Table 3. Characteristics of existing solution vs. ours

Leg@net [20] is a solution to Multi-master replication for routing transactions in a database cluster. Leg@net relaxes as much as possible replica freshness, to reduce the over-head of replicas synchronization and thus to allow more resources to transactions processing. It targets transactional applications whose autonomy must be preserved. However, this solution is not suited to a larger scale because the middleware is centralized.

DTR [21] is a solution which takes into account the problem due to concurrent updates and controls the freshness in order to improve performances. DRT2 [22] is a new system relying on the Leg@net [20] approach to deal with transaction routing at a large-scale. It also extends [21] for dealing with node failures (or dynamicity) which are very frequent in large scale systems.

Pastis [23] 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. The peer-to-peer file systems are found to be effectively deployed in some other studies also. [25, 26, 27].

Finally, we summarize in Table 3 the main properties that characterize each related work in comparison to our approach.

9. Conclusion

In this paper, we have presented D3-P2P a novel architecture for fully decentralized approach to distributed transaction management in peer-to-peer environments. This architecture is based on timestamping, managing transactions and construction of a 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 it must selects for each sub queries a server replica from a quorum of each group. Finally, the coordinator manager layer coordinates with other coordinators in order to execute, correctly, the distributed transaction. This layer receives from transaction manager the selected set of server replicas, sub transactions and queries which have the same timestamp. Also, we have proposed a solution to deal with node dynamicity in both predicable and unpredict-able disconnection. Finally, we have proposed a protocol to avoid a deadlock in P2P system based on quorum, this protocol is fully distributed, because there is no centralized component to manage the deadlock.

In future work, we will integrate new functionalities in this architecture, such as how the transaction manager selects server replicas (a quorum) in order to load balancing between these servers and how we deal with byzantine nodes.

References

[1] Milojicic, DD. S., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S., Xu, Z. (Peer-to-Peer Computing. Tech. Report: HPL-2002-57, available on line at: http://www.hpl.hp.com/techreports/ 2002/HPL-2002-57.pdf

[2] Shirky, C. (2001). What is P2P and What Isn't. The O'Reilly Peer to Peer and Web Service Conf., Washington, D.C. November 5-8. Available on: http://conferences.oreillynet.com/p2p/.

[3] Theotokis, S. A., Spinellis, D. (2004). A survey of peer-to-peer content distribution technologies, *ACM Computing Surveys,* 36 (4) 335-371.

[4] Naor, M., Wool, A. (1998). The load, capacity, and availability of quorum systems, *SIAM Journal on Computing*, 27 (2) 423-447.

[5] IEEE *International Conference on Machine and Web Intelligence* ICMWI'10.

[6] *International Conference on Information Technology and e-Services* ICITeS'11.

[7] *Hemam, Sofiane Mounine., Hidouci, Khaled Walid* (2011). *Replicated Database Transactions Processing in Peer-To-Peer Environments, Journal of Networking Technology,* 2 (2) 63-72.

[8] To be published in procedia computer sciences journal-Elsevier

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

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

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

[12] Patino-Martinez, M., Jimenez-Peres, R., Kemme, B., Alonso, G. (2005). MIDDLE-R, Consistent Database Replica-tion at the Middleware Level. *ACM Transactions on Computer Systems* (TOCS), 28 (4).

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

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

[15] Cecchet, E., Marguerite, J., Zwaenepoel, W. (2005). C-JDBC: Flexible Database Clustering Middleware. Technical report, Object Web, Open Source Middleware.

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

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

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

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

[20] 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) 320–343, April.

[21] Sarr, I., Naacke, H., Gançarski, S. (2008). DTR: Distributed transaction routing in a large scale network. Proc. LNCS, High-Performance Data Management in Grid Environments (VECPAR'08), 5336, 521–531.

[22] 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.

[23] 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 (Eu-ro-Par 2005), 3648, 1173-1182, September.

[24] PeerSim. http://peersim.sourceforge.net/.

[25] Gotoh, Yusuke., Suzuki, Kentaro.,Yoshihisa, Tomoki., Taniguchi, Hideo., Kanazawa, Masanori (2011). Brossom: A P2P Streaming System for Webcast, *Journal of Networking Technology,* 2 (4) 161-189.

[26] Moujane, Ahmed., Chiadmi, Dalila., Benhlima, Laila Wadjinny, Faouzia (2012). Content Based Clustering for Semantic P2P Data Integration, *Journal of Networking Technology,* 3 (1) 13-27.

[27] Farrugia, Anthony., Al-Jumeily,Dhiya (2012). The Design, Implementation and Evaluation of a Web-based Student Teachers' ePortfolio (STeP), *International Journal of Web Applications*, 4 (2) 96-105.