Concurrent execution of transactions in a peer-to-peer database network

Mehedi Masud* and Sultan Aljahdali

Department of Computer Science,
College of Computers and Information Technology,
Taif University,
P.O. Box 888, Zip Code 21974, Taif, Saudi Arabia
E-mail: mmasud@scientist.com
E-mail: aljahdali@tu.edu.sa
*Corresponding author

Abstract: Transaction execution in a peer-to-peer database network specifies an update made to a peer's instance is applied to the peer's local database and propagated to related peers. Maintaining a successful execution of a transaction in such a network is challenging due to the dynamic behaviour of peers and unstructured topologies of networks. In this paper, we present a decentralised transaction execution process that guarantees the correct execution of a transaction without relying on any global coordinator. In the network, a peer executes a transaction and provides the local execution information to the initiator of the transaction. The initiator of a transaction plays important roles for the successful execution and termination of a transaction. Transactions originated from different peers may involve in a conflict during their execution in the network. In this paper, we also show a process to resolve conflicts using a universal leader election algorithm, called Mega-Merger.

Keywords: database; transaction processing; peer-to-peer networks; intelligent information; concurrency.

Reference to this paper should be made as follows: Masud, M. and Aljahdali, S. (2011) 'Concurrent execution of transactions in a peer-to-peer database network', *Int. J. Intelligent Information and Database Systems*, Vol. 5, No. 5, pp.510–531.

Biographical notes: Mehedi Masud received his PhD in Computer Science from the University of Ottawa, Canada. He is an Assistant Professor at the Department of Computer Science, Taif University, KSA. His research interests include issues related to P2P and networked data management, query processing and optimisation, and information security. He has published several research papers at international journals and conferences.

Sultan Aljahdali received his BS from Winona State University, Winona, Minnesota in 1992, his MS with honour from Minnesota State University, Mankato, Minnesota in 1996, and his PhD in Information Technology from the Volgenau School of Information Technology and Engineering at George Mason University, Virginia, USA. He is the Dean of the College of Computers and Information Systems at Taif University. His research interest includes software testing, developing software reliability models, and soft computing for software engineering.

1 Introduction

In the last few years, steady progress has been made in research on various issues related to peer data management systems, such as data integration models (Halevy et al., 2003), mediation methods (Halevy et al., 2004), coordination mechanisms (Serafini et al., 2003; Rodriguez-Gianolli et al., 2005), and data-level mappings (Kementsietsidis et al., 2003) among the peer databases. These systems combine both P2P and database management system functionalities. The local databases on peers are called *peer databases*. Each peer chooses its own database schema and maintains data independently. Contrary to the traditional data integration systems where a global mediated schema is required for data exchange, in peer data management systems semantic relationships exist between two peers, or among a small set of peers for sharing data. The data is accessed globally from any peer by traversing the network of peers.

There is an increasing interest in the creation of peer data management systems, which includes establishing and maintaining mappings between peers and processing queries using appropriate propagation techniques. While there is a rich body of research concerning frameworks and mapping issues among peers, dynamic aspects of data in such systems have received much less attention. For example, in many data sharing efforts, particularly in biological and health sciences, data in sources are continuously corrected and cleaned by the users of the local sources. In this case, the exchange of updates among sources is equally important in order to keep the peers updated with the cleaned data. In such an update exchange, a question of significant interest is how to define consistency during the exchange and processing of updates, while still allowing autonomy among the peers. Surprisingly, little work has addressed update exchange mechanisms for peer data management systems.

Peers in a peer-to-peer database network are autonomous and there is no global control of the execution of transactions. Therefore, during propagation of transactions, different conflicting situations with respect to transactions may occur which lead to data inconsistency in the network. Hence, a conflict resolution protocol is required to select the candidate transaction from the conflicting transactions.

In this paper, we consider this problem of consistent execution of transactions and propose a decentralised mechanism for resolving conflicts. In this approach, conflicts are resolved in a decentralised collaborative fashion by exchanging some status information of the transactions between the initiator and participants. In the process, a peer that executes a given transaction is called a participating peer or simply a *participant*. The status information provided by a participant to initiators includes the local execution status of the transaction, the local conflict information, and the transactions spawned by the participant. Essentially, each participant exchanges information with the transaction's initiator during the execution of a transaction. The initiator plays an important role for the correct execution, conflict resolution, and termination of transactions. Initiators of the conflicting transactions select a candidate transaction and the candidate transaction is finally executed in the network. A candidate transaction is selected using a universal leader election protocol, called Mega-Merger (Santoro, 2006). The Mega-Merger protocol is selected since it runs in every network, requires no a priori knowledge of the topology of the network nor its properties.

The paper is organised as follows: Section 2 presents the system model of a peer-to-peer database network and describes the properties of a global transaction. Section 3 describes the execution protocol of a global transaction and Section 4 presents the process of selecting a candidate transaction from the conflicting transactions. Section 5 presents results we achieved from experiments and Section 6 reviews related work. Finally, Section 7 concludes.

2 System model

We assume a peer-to-peer database network with a set of peers $P = \{P_1,P_2,\ldots,P_n\}$ where each peer P_i has a pre-existing database DB_i. Each peer has full control over its local database (e.g., modify schema, update data in the database). Each peer also establishes mappings with other peers in the network in order to share data. Mappings specify data sharing constraints between peers.

In P2P, there are two types of mappings, schema-level (Halevy et al., 2004) and data-level (Kementsietsidis et al., 2003). A schema-level mapping is a logical assertion of the form:

$$\forall\overline{x},\overline{y}(\phi(\overline{x},\overline{y}) \rightarrow \exists\overline{z}\Psi(\overline{x},\overline{z}))$$

where the left hand side (LHS) of the implication, ϕ, is a conjunction of atoms over variables $\overline{x}$ and $\overline{y}$, and the right hand side (RHS) of the implication, Ψ, is a conjunction of atoms over variables $\overline{x}$ and $\overline{z}$. The mapping expresses a constraint about the existence of a tuple in the instance on the RHS, given a particular combination of tuples satisfying the constraint of the LHS. Data-level mappings can be established using mapping tables (Kementsietsidis et al., 2003). A mapping table is a relation over the attributes X,Y, where $X \subseteq U_i$ and $Y \subseteq U_j$ are non-empty sets of attributes from two peers P_i and P_j. A tuple (a, b) in a mapping table indicates that the value $a \in dom(X)$ is associated with the value $b \in dom(Y)$. Mapping tables are generally used when there is data level heterogeneity between peers. Mappings in mapping tables also store data sharing constraints between two peers corresponding to the associations in mapping tables. Without loss of generality, we assume that mappings are in placed by the administrator of each peer using common agreements when they want to share data. The construction of mappings m_{ij} forms an acquaintance (i, j) between P_i and P_j. Here, P_j and P_i are *acquaintees* of each other.

2.1 Transaction model

A transaction consists of a sequence of read-and-write (update) operations on data items. A transaction is classified as a *read-only transaction* or an *update transaction*. A read-only transaction consists of only read operations that executes in the network without involving in the proposed conflict resolution protocol. This allows a read-only transaction to terminate its execution without being blocked. On the other hand, an update transaction consists of a sequence of write operations that is executed in the network may involve in the proposed conflict resolution protocol.

In a peer-to-peer database network, when a user submits a transaction T_i to a peer P_i, the transaction is executed at P_i and appropriate actions are performed in its local database DB_i. Peer P_i is called the *initiator* of T_i. For maintaining data consistency between peers, whenever changes occurs in data at P_i by T_i, the data in each acquaintee P_j of P_i need to be changed. However, this is subject to the satisfaction of the mapping Σ_{ij} between P_i and P_j. If the data accessed by T_i satisfies the mapping Σ_{ij} then P_i forwards T_i

to its acquaintees. Before forwarding T_i, P_i transforms T_i wrt the schema of its acquaintees. The transformation of T_i for an acquaintee P_j is denoted by T_i^j. When P_j receives T_i in transformed form T_i^j, P_j also executes T_i and forwards T_i to its acquaintees. This is a recursive process. Base cases of the recursion are peers those have no acquaintees to forward the transaction, i.e., the peers have no mappings with any other peer. We call these peers *terminate peers*. Therefore, a transaction is propagated from the initiator to all related peers until the transaction propagation ends at terminate peers. Hence, from an initial transaction, a set of transactions is generated dynamically in the network. The initial transaction is called a *global* transaction since the transaction is executed in the network. The set of transactions generated from the global transaction are called *remote* transactions. The semantics of global and local transactions is discussed in (Masud and Kiringa, 2007).

We now describe the logical structure of a global transaction generated from a transaction T_i originated at P_i. When P_i produces a set of remote transactions from T_i for the execution in its immediate acquaintees, T_i can be viewed as a two-level global transaction. In this case, T_i becomes the root. T_i becomes a multi-level global transaction when the acquaintees of P_i also generate remote transactions for their respective acquaintees. Consequently, a global transaction may have multiple layers depending on the number of hops it propagates. Intuitively, as remote transactions are generated in the system acquaintance-by-acquaintance, a *transaction dependency graph* is induced. The nodes in this graph represent remote transactions and there is an edge from a transaction T_i^j to a transaction T_i^k, if T_i^k has resulted from the propagation of T_i^j by P_j to P_k. When a peer receives a transaction, the transaction is either executed (if the transaction does not involve in a conflict with any other transaction originated by another peer) or is blocked or halted (if conflict occurs). If a transaction is blocked then it participates in the election process to become a candidate. When the transaction becomes a candidate, the execution of the transaction continues. If the transaction fails to become a candidate, it is compensated and no further execution of the transaction occurs. The more details is provided in Section 4. Note that cycles can exist in the network topology. Therefore, a peer can receive the same transaction from multiple paths from a peer that originated the transaction. We assume that when a peer receives the same transaction it just discards the transaction that is later received.

The execution of a transaction in a peer-to-peer database network is different from other extended transaction models, such as nested transactions (Moss, 1985), sagas (Garcia-Molina and Salem, 1987), etc. The difference is that the set of component transactions to be invoked in a peer-to-peer database is not known in advance. The component transactions are generated dynamically based on mappings between peers. In this respect, transactions in a peer-to-peer database network are closest to the transactional model for long running activities proposed in Dayal et al. (1991). Moreover, each of the transaction generated from the initial transaction is an atomic transaction resulted from the direct or indirect propagation in the network. Each transaction accesses data items only at the local peer. Unlike a transaction in a multi-database system (Breitbart and Silberschatz, 1988; Breitbart et al., 1992), a transaction is not decomposed into sub-transactions to access data at different peers.

There is also a difference between a distributed transaction model and P2P transaction model. In a distributed transaction model global level transactions are issued to the global transaction manager (GTM), and are decomposed into a set of sub-transactions to be individually submitted to the corresponding LDBSs. However, in our P2P transaction model, a global transaction is not decomposed but rather is propagated as an entire transaction. A peer, after executing a transaction locally, forwards the entire transaction (not the individual read-and-write operations that constitute the transaction) to its acquaintances. The remote peer that receives the transaction considers the transaction as submitted by local users. In a distributed transaction model, transactions are executed under the control of the GTM. In contrast, a P2P transaction model is built on a network of peers without a GTM or controller. However, we assume that each local database management system preserves the atomicity, consistency, isolation, and durability (ACID) properties (Bernstein et al., 1987) of transactions and ensures serialisability of the local schedule using the local concurrency protocol since the LDBSs are pre-existing. In a traditional distributed database system, serialisability is ensured using the distributed two-phase (2PL) protocol (Bernstein et al., 1987) and atomicity of transactions is ensured using the two-phase commit (2PC) protocol (Bernstein et al., 1987). However, in a P2P transaction model, application of these protocols is not feasible or applicable.

2.2 Transaction execution life cycle

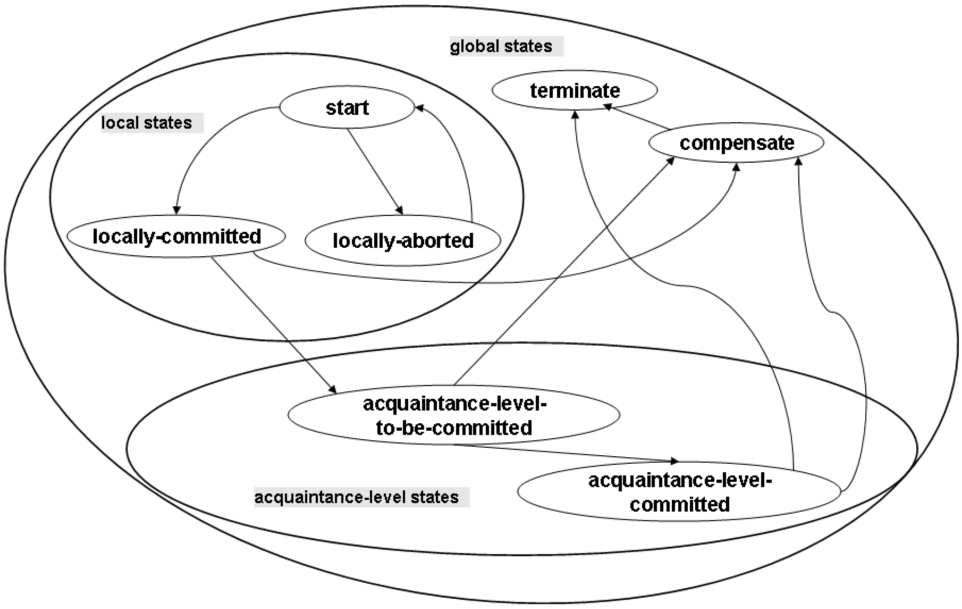
A transaction may have different execution status during its execution in a peer-to-peer network depending on the execution level of the transaction. The levels are execution of a transaction in a peer, in acquaintees, and in a network. We categorise the execution status into three transaction state groups, namely, *local*, *acquaintance-level*, and *global*. The local states show the execution status of a transaction in a peer, the acquaintance-level states show the execution status of a transaction in the immediate acquaintees of a peer, and the global states show the status of a transaction in the network. In the following we describe the groups and the states. In Figure 1, we depict the states of a transaction during its execution in a peer to peer network.

2.2.1 Local

Local states symbolise the different sates of a transaction during its local execution in a peer. There are three different local states, namely, *start*, *locally-aborted* (*LA*), and *locally-committed* (*LC*).

The start state symbolises the beginning of execution of a transaction in a peer. A transaction can be LA or LC in a peer. If a transaction is successfully executed in a peer, it is committed by the local transaction manager of the peer and the state of the transaction changes from start to LC state. A change of state is denoted by an arrow in the Figure 1. However, if the transaction is aborted due to the failure of execution, the state becomes LA. Examples of a transaction abort are a transaction abort to timeout, or a failure to pass the validation test by the transaction manager of a peer. The transaction manager starts execution of a LA transaction after the recovery steps that are managed by the recovery manager of the database management system in the peer. The details of the recovery process can be found in Gray and Reuter (1993). If a peer finds the state of a transaction in LC state, the peer forwards the transaction to its acquaintees and the state of the transaction is changed from LC to *acquaintance-level-to-be-committed* state. Now, the peer waits for the successful execution of the transaction in its acquaintees. The state of a transaction can be changed from LC to *compensate* state as shown in the Figure 1 if the transaction is involved in a conflict with another transaction before forwarding the transaction to its acquaintees. In this case, the transaction is selected as a *victim* for compensation and the state is changed from LC to compensate. In *global* state, we talk about compensate state and in Section 4, we describe the situation when a transaction is selected as a victim transaction.

Figure 1 States of a transaction



2.2.2 Acquaintance-level

There are two states in this group, namely, *acquaintance-level-to-be-committed* (*ALC*) and *acquaintance-level-committed* (*AC*). These two states symbolise the execution status of the forwarded transaction in the immediate acquaintees of a peer. The ALC state symbolises that the forwarded transactions are to be committed at acquaintees and the AC state symbolises that the forwarded transactions are successfully committed at the acquaintees. If the acquaintees committed the transaction, the state of the transaction changes to acquaintance-level-committed for that level of acquaintance from which the transactions are forwarded. The state of a transaction can be changed from ALC to *compensate* state if a forwarded transaction in an acquaintee involves in a conflict with another transaction that the acquaintee received from another peer. The decision is made by the conflict resolution protocol described in Section 4.

2.2.3 Global states

The global states symbolise the execution status of a transaction in a peer-to-peer network. There are two states in this group, namely, *terminate* and *compensate*. The terminate state of a transaction symbolises that the transaction is successfully committed by the participating peers in the network. If a transaction is terminated, all the information related to the execution of the transaction in the network is deleted from the participating peers. The compensate state of a transaction symbolises that the transaction has involved in a conflict with another transaction and the conflict resolution protocol in Section 4 has decided to compensate the effect of the transaction in participating peers. This compensation is done by invoking a compensate transaction in reverse order (Schuldt et al., 2002). The compensate transaction semantically undoes the effect of the execution of the transaction.

3 Transaction execution

In this section, we present a transaction execution protocol. The protocol relies on the following observations:

- *Conflict graph (CG):* Each peer maintains conflict relationships among the active transactions in the form of a *CG* that the peer executes. The transactions that are not terminated in the network are called active transactions. A conflict relationship, i.e., an edge between two transactions is created in the graph based on the notion of potential conflict (Ganarski et al., 2007). According to the definition in Ganarski et al. (2007), a potential conflict occurs between two transactions if they access at least one data item in common and at least one of the transactions performs a write operation on that data item. This potential conflict does not allow a read transaction to continue its execution in a P2P network. In a P2P network, a read transaction should continue its execution without being halted. This eliminates the abort of a read transaction. Since queries are more frequent than updates in P2P networks, allowing a read transaction to execute without involving in a conflict resolution protocol is logical, though sometimes users will not get the consistent result. We say a transaction T_i which is active in a peer P_i potentially conflicts with another transaction T_j that is also active in the same peer, if both the transactions access at least one data item in common and perform a write operation on that common data item. This definition allows a read transaction to execute in the network without being halted. Formally, we define a potential conflict as follows:
  - *Potential conflict:* Let T_i and T_j be two transactions that are active in a peer. Let $WS(T_i)$ and $WS(T_j)$ denote the set of data items on which T_i and T_j perform write operations respectively. A potential conflict occurs between T_i and T_j if $WS(T_i) \cap WS(T_k) \neq \emptyset$.
- *Transaction dependency tree (TDT):* Each global transaction initiator maintains a dynamic data structure, called *TDT*, for each global transaction it originates until the transaction is terminated in the network. TDT is used to keep the dependency relationships among the remote transactions generated from a global transaction in the network. The construction of a TDT for a global transaction is discussed below.
  1. When a peer P_i initiates a transaction T_i and successfully executes T_i, the peer creates a node for T_i in the TDT of T_i.

2. Peer P_i generates remote transactions for its acquaintees and forwards the remote transactions. When remote transactions are forwarded, a list of new remote transactions at the node T_i of $TDT(T_i)$ is added plus edges are inserted between T_i and the newly generated remote transactions.
3. When a peer receives a remote transaction, it executes the transaction locally and generates remote transactions for its acquaintees.
4. After successfully executing the received transaction, a peer sends a *vote* message to the initiator and waits for a *forward* message from the initiator in order to forward the newly generated remote transactions. When a peer sends a vote message, a peer also attaches a list of the newly generated remote transactions from the received transaction plus the ids of new acquaintees where the peer is ready to forward the new transactions.
5. When the initiator receives the vote message, it creates nodes for each of the new transaction that are in the vote message and inserts edges between the newly generated transactions and the remote transaction from which the transactions are generated. Initiator now sends a forward message to the sender of the vote message.
6. When a peer receives a forward message it forwards the remote transactions to its acquaintees.

Note that when a peer forwards a transaction, it also forwards the id of the initiator and the global id of the transaction. In this way, every peer knows which peer is the initiator of the global transaction. Here, we do not show any conflicting scenario during the construction of a TDT. In Section 4.2, we shall show how to deal with conflicts between transactions generated from different peers. We now give an example.

Figure 2 TDT construction, (a) peer to peer network (b) TDT for the transaction T_1 initiated at P_1

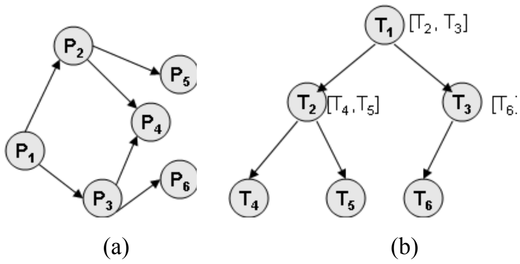


Figure 2 shows the construction of a TDT corresponding to a transaction T_1 that is originated at P_1. Figure 2a shows a peer-to-peer database network. Figure 2b depicts the construction of the tree from T_1. After P_1 successfully executed T_1, it creates a node for T_1 and it becomes the root of TDT for T_1. After that P_1 generates two remote transactions T_2 and T_3 from T_1 for acquaintees P_2 and P_3 and forwards the transactions. Assume that a remote transaction for a peer P_j is denoted by T_j. P_1 now inserts an edge from T_1 to each of the remote transaction T_2 and T_3 and add a list [T_2, T_3] at T_1 node. After receiving T_2 from P_1, P_2 executed T_2 successfully. P_2 also generated two new remote transactions T_4 and T_5 from T_2. P_2 now sends a vote message to the initiator of T_2 together with the list of transactions [T_4, T_5] that are generated from T_2 at P_2. When P_1 receives the vote message, it creates two new nodes for T_4 and T_5 and inserts edges from T_2 to T_4 and from T_2 to T_5. P_1 sends a forward message to P_2. Note that P_2 is waiting for the decision from the initiator of T_2 in order to forward the transactions T_4 and T_5. Only after receiving the decision from the initiator, P_2 forwards the transactions T_4 and T_5 to P_4 and P_5, respectively. P_1 now waits for the execution decision of T_4 and T_5 from P_4 and P_5. Similarly, P_3 does the same task. Note that according to the links in Figure 2a, P_4 receives the same global transaction from P_2 and P_3. We assume that P_4 receives the transaction from P_2 earlier than P_3. Hence, no edge is created from T_3 to T_4 since P_4 rejects T_4 from P_3.

3.1 Transaction execution protocol

A transaction execution protocol starts when a peer receives a transaction from its clients. As we mentioned earlier that an initiator maintains a dynamically generated TDT for a global transaction it originates. Besides maintaining a tree, each initiator also maintains a *transaction status tree* (TST) for monitoring the execution status of the component transactions of a global transaction. Each node in a TST is labelled with a state that represents the status of a remote transaction in a peer. When a remote transaction, e.g., T_i is executed locally in a peer P_i, the corresponding node status is changed to LC_i. When all the remote transactions generated from T_i are executed successfully by all the relevant acquaintees, then the status of T_i is changed to AC_i. When the status of all the nodes is acquaintance-level committed then the initiator sends a terminate message to all the peers. After receiving the terminate message all the peers delete the stored information of the transaction.

An example of a transaction execution protocol is depicted in Figure 3. In the figure, left side shows a peer-to-peer database network where a transaction T_1 is originated at peer P_1. In the following, the steps of the protocol are described.

- *Step 1:* T_1 is executed at P_1. Hence, a node LC_1 is created in $TST(T_1)$ for T_1 showing that T_1 is locally committed.
- *Step 2:* P_1 has forwarded T_2 and T_3, the remote transactions generated from T_1, for peers P_2 and P_3. P_1 marks T_1 in $TST(T_1)$ to ALC_1 and waits for the votes from P_2 and P_3.
- *Step 3:* P_1 receives votes from P_2 and P_3. The status of T_1 is changed to AC_1 since T_1 has been executed successfully in P_1's acquaintees.
- *Step 4:* After receiving the vote message from P_2, P_1 knows that P_2 has no transaction to forward, therefore, an edge from $AC_1 \rightarrow AC_2$ is inserted. It represents that the component transaction T_2 has been successfully committed at P_2 and P_2 has not generated any new remote transaction. On the other hand, when P_1 receives the vote message from P_3, P_1 knows that P_3 has generated new remote transactions T_4 and T_5 to be forwarded to P_4 and P_5. Therefore, an edge $AC_1 \rightarrow ALC_3$ is inserted. It represents that T_3 is acquaintance-level-to-be-committed, that means P_1 has to wait

for the execution decision from P_4 and P_5. P_1 also sends a forward message to P_3 allowing P_3 to forward the newly generated transactions.

- *Step 5:* After receiving the forward message from P_1, P_3 forwards T_4 and T_5 to P_4 and P_5 respectively. P_1 receives vote message from P_4 and P_5 about the successful execution of T_4 and T_5 generated from T_3. Therefore, the status of T_3 is changed from ALC_3 to AC_3. It denotes that component transactions of T_3 have been successfully executed at the acquaintees of P_3.
- *Step 6:* When P_1 receives vote messages from P_4 and P_5, P_1 knows that there is no more component transactions generated from T_4 and T_5. Therefore, edges $AC_3 \rightarrow AC_4$ and $AC_3 \rightarrow AC_5$ are inserted. The edges denote that no further propagation has happened and all the remote transactions have been successfully executed in the network.
- *Step 7:* When P_1 notices that each node has the status AC, P_1 sends a termination message to all the participants of T_1. All the peers then terminate (T) the execution of T_1 and do the garbage collection.

Figure 3 Transaction execution protocol

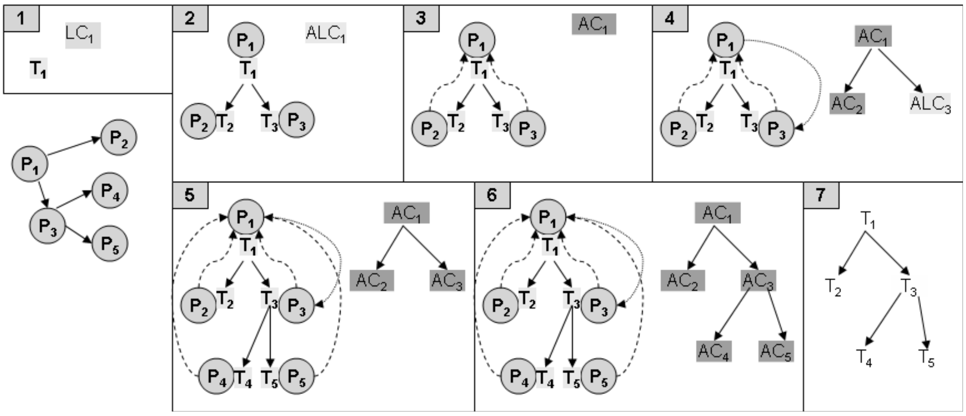


Figure 4 presents the protocol. From the protocol, we notice that the initiator maintains two data structures for a transaction T_i: a TDT ($TDT(T_i)$) and a TST ($TST(T_i)$). When a transaction T_i is originated at P_i, the transaction is first executed at P_i and P_i starts building $TDT(T_i)$ and $TST(T_i)$. After the local execution of T_i, P_i finds the list of acquaintees relevant to T_i using the function $ACQ(P_i(T_i))$. If there is no relevant peers for T_i, the execution of T_i is terminated at P_i. If there are relevant acquaintees for T_i then T_i is entered into the global execution phase. In the global execution phase, the initiator first updates $TDT(T_i)$ and $TST(T_i)$. Updating $TDT(T_i)$ has the following steps:

1. transforms T_i to T_j for all relevant acquaintees P_j in Π.
2. inserts an edge from T_i to each T_j
3. propagates each T_j to the respective acquaintee.

Figure 4 Transaction execution protocol

```
status:{INITIATOR, PARTICIPANTS}
INITIATOR:
  users invoke a transaction T_i at peer P_i
    execute T_i; start construction of TDT(T_i)
    start building TST(T_i)
    update TDT(T_i)
    Π = ACQ(P_i(T_i))
    if |Π|==0 then
      terminate T_i
      remove information of terminating transaction T_i
    else
       update TDT(T_i); update TST(T_i)
      while true do
       wait for response R_j from all P_j
       for each response R_j do
         update TDT(T_i); update TST(T_i)
       endfor
       if checkTerminate(T_i)==true then
         send a terminate message to all peers
         remove information of terminating transaction T_i
         terminate while loop
       endif
      endwhile
    endif

PARTICIPANTS:
  while true do
   wait for next message m
   switch message type of m do
    case T_j invocation
      execute T_j
      send response to INITIATOR
   case forward control
      forward component transactions generated from T_j
   case T_i terminate control
      remove information of terminating transaction T_i
   endswitch
   endwhile
```

Meanwhile, updating *TST*(T_i) changes the status of a transaction based on the response received from the participants. In Section 3.1, we discussed how the status of a *TST* changes. When global execution phase starts, the initiator waits for responses from the participants. For each response, the initiator updates the *TDT* and *TST*. Updating *TDT*(T_i) also includes sending *forward* and *terminate* control messages. When a forward message is sent to a participant, the participant forwards the component transactions to its acquaintees generated from T_i. The initiator sends a terminate message when the status of all the nodes of $TDT(T_i)$ becomes acquaintance-level-committed. The termination condition is checked by the initiator through the *checkterminate*(T_i) function. Meanwhile, when a participant receives a transaction, it first executes the transaction locally then sends response message to the initiator. The response message includes:

1 id of the peer

2 id of the transaction

3 list of component transactions it generates

4 list of peers' ids to those the peer is waiting to forward the transactions.

When a participant receives a forward message, it then forwards the remote transactions. A peer terminates the execution of a transaction when it receives a terminate message.

Note that, a transaction can involve in a conflict during different states of the transaction. In the next section, we describe the mechanism of dealing with conflicts.

4 Candidate transaction selection protocol

In this section, we propose a protocol that selects a candidate transaction from the conflicting transactions that will eventually be executed in the network. Selecting a candidate transaction is required when more than one transactions conflict with each other during their execution in the network and the transactions are generated from multiple peers. Consider a situation where a peer receives two updates generated from two peers that modify a tuple in the database. Without the execution knowledge of other peers, the peer is unable to make a decision which one to accept or reject. Due to the arbitrary topology of a peer-to-peer database network, a conflict between the same pair of updates may occur at different peers during their propagation. In order to keep the databases consistent, each peer must reach the same decision to execute the updates.

We already mentioned that each peer maintains a CG for keeping the conflict relationship among transactions by implementing any existing conflict detection technique. According to the protocol, when a peer detects a conflict, the peer informs the conflict information to the initiators of the transactions and stops further execution and propagation of the transactions. For example, consider a situation where a peer P_k has executed a transaction T_1 before a transaction T_2 arrives. When T_2 arrives at P_k and T_2 conflicts with T_1, then P_k sends the conflict information to both the initiators of T_1 and T_2. Assume that T_1 and T_2 are originated at P_1 and P_2, respectively. Now, P_1 and P_2 detect a candidate transaction that will continue its execution. However, the victim transaction will be compensated. When a transaction is selected as a victim, the initiator of the victim transaction sends a compensate message to the participating peers of the victim transaction. After successful compensation, the peer which originated the victim transaction informs the originator of the candidate transaction. This decision enables the candidate transaction to continue its execution further in the network. In the proposed protocol, the initiators use a leader election algorithm to select the victim and the candidate transaction. Essentially, we adopt the concept of a universal leader election algorithm, called Mega-Merger (Santoro, 2006), to select the candidate transaction. Since, we consider the semantic conflict between transactions, therefore a single transaction must be executed in the network among the conflicting transactions. In the following, we discuss the concept of the Mega-Merger leader election protocol and simultaneously, we show how this concept fits our protocol for selecting the candidate transaction.

4.1 Concept of the Mega-Merger protocol

Mega-Merger is an efficient protocol for a leader election and the main feature of this protocol is that it is topology independent. In this protocol, nodes are treated as small villages, and edges are roads with different names and distances. A group of villages has a city. Initially, a village is also treated as a city of its own village. The goal is to have all villages merge into one large *megacity*. A city, even a village always tries to merge with the closest neighbouring city. When a merge of two cities takes place there are several important issues are considered:

1 the naming of the new city, the resolution of this depends on how far the involved cities have progressed in the merging process, i.e., on the level they have reached, and on whether the merge decision is shared by both cities

2 the decision of which roads of the new city will be serviced by public transports.

When a merge occurs, the roads of the new city serviced by public will be the roads of the two cities already serviced plus only the shortest road connecting them. In the following we describe the basic principles of the election algorithm and show how does it fit in our protocol.

- A *city* is a rooted tree; the nodes are called *districts*, the root is also known as *down-town*. Similarly, in our protocol, when a global transaction is executed in the network, a TDT is constructed. The transaction when it is originated becomes the root of the tree and all the remote transactions generated in the network progressively can be treated as districts.
- Each city has a level and a unique name; all districts eventually know the name and the level of their city. Similarly, in our framework, the initiator knows how many peers have executed the transactions successfully, since each participant sends a vote message to the initiator after the execution of a transaction. We can treat this count as a level of a global transaction. The level of a global transaction T_i is denoted by $level(T_i)$. In Mega-Merger, all districts know the name of their city. Similarly, all the participants of a transaction know the initiator of the transaction.
- Edges are roads, each with a distinct name and distance. In TDT, edges are acquaintance links through which a transaction has propagated. However, TDT does not need any name and distance concept for the edges.
- Initially, each node is a city with just one district, itself, and no roads. All cities are initially at the same level, i.e., zero. Similarly, when a transaction is originated at a peer and is executed locally, it can be treated as a city with one district, i.e., the transaction itself.
- A city merges with its closest neighbouring city to become a bigger city. To request the merging, a *Let-us-Merge* message is sent on the shortest road connecting it to the city. In the proposed protocol, there is no specific merge request from the originator of a transaction. The merging of two TDTs starts corresponding to two global

transactions when the transactions conflict in a peer during the construction of the TDTs. A merging situation occurs in an acquaintance link when a peer receives a transaction from an acquaintee through the acquaintance link and the received transaction conflicts with a transaction that is active in that peer. In this case, we can treat the edge as a merge link.

- When a merge occurs, the roads of the new city serviced by public transport will be the roads of the two cities already serviced plus only the shortest road connecting them. In our protocol, when a transaction becomes a candidate then the merge process starts. In the merge process, first the peers that executed the victim transaction are considered for the execution of the candidate transaction. This results the merge of TDT of the victim transaction with the TDT of the candidate transaction. For merging, the candidate transaction starts its execution along the edges of TDT of the victim transaction. The propagation of the candidate transaction starts from the merge link. Before, the propagation starts, the initiator of the victim transaction first sends a compensate message to all the participants of the victim transaction in order to revert the execution effect of the victim transaction.

4.2 Selection of a candidate transaction

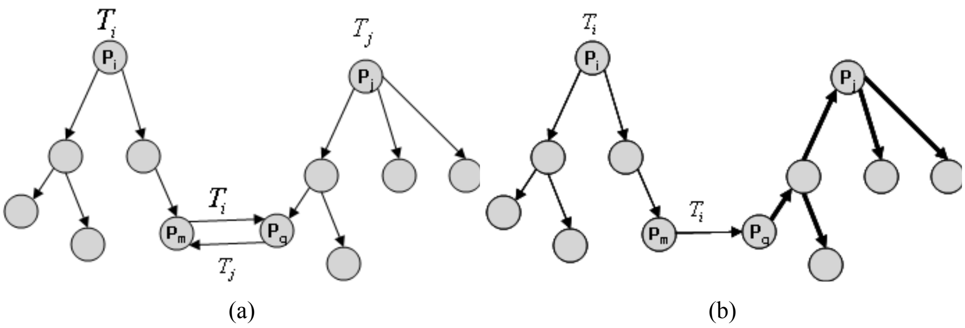
In this section we describe the process of selecting a candidate transaction from the conflicting transactions. A candidate is selected using two resolution protocols. The protocols are *friendly resolution* and *absorption resolution*. In the following, we discuss the protocols.

Consider two transactions T_i and T_j originated by P_i and P_j, respectively. Also, assume that T_i and T_j are conflicting transactions. The TDTs are denoted by $TDT(T_i)$ and $TDT(T_j)$, and the levels of the trees are denoted by $level(T_i)$ and $level(T_j)$, respectively.

- *Friendly resolution* ($level(T_i) = level(T_j)$)*:* There are two cases in friendly resolution.
  a *Case 1:* A participant P_m of T_i forwards T_i to a participant P_q of T_j and P_q also forwards T_j to P_m.
    - *Solution:* When P_m and P_q identify a conflict, they inform both the initiators of T_i and T_j. After receiving the conflict information, P_i and P_j choose one of the transactions as a candidate transaction and the other becomes a victim transaction. Consider that T_i is selected as a candidate transaction. When T_i is selected as a candidate transaction, the edge $P_m \rightarrow P_q$ becomes the merge link. After selecting the candidate, merging of $TDT(T_j)$ into $TDT(T_i)$ starts. There are two phases of merging:
      1 compensation
      2 merging.

  During the merge process, the status of T_j changes to compensate and the compensation phase begins. During the compensation phase, no new transaction is allowed to execute by the peers those are involved in constructing $TDT(T_j)$ and $TDT(T_i)$ and the further propagation of the transactions T_i and T_j is stopped. In order to begin the compensation phase, P_j sends a compensation message to all the participants of T_j. Each participant now generates a compensate transaction T_j^- and completes the compensation task. After compensation is done P_j informs the initiator of T_i that the compensation is completed. Now the merging process starts. The merging process starts from the merge link. In the merging process, execution of the candidate transaction starts to the participants of the victim transaction from the merge link. Figure 5a illustrates the conflict scenario. Consider that T_i is select as a candidate transaction. Therefore, the merge link is $P_m \rightarrow P_Q$. The merge process is depicted in Figure 5b. The bold edges show the merging of *TDT*(T_j) with *TDT*(T_i) and the propagation of T_i in *TDT*(T_j). After merging process is finished, all the participants of T_j become participants of T_i. Now, the execution of T_i starts. After the merge process, the *level*(T_i) is set to the summation of the *level*(T_i) and *level*(T_j).

Figure 5 Friendly resolution, (a) two TDTs have the same level considering that T_i and T_j are conflicting transactions (b) T_i is chosen as candidate and *TDT*(T_j) merged with *TDT*(T_i)



b *Case 2:* A participant P_m of T_i receives T_j from a participant P_q of T_j.

- *Solution:* When P_m identifies the conflict, it informs both the initiators of T_i and T_j. Now, the same solution is applied as described in Case 1.

- *Absorption resolution* (*level*(T_i) ≠ *level*(T_j))*:*

  a *Case 1:* A participant P_m of T_i forwards T_i to a participant P_q of T_j and P_q also forwards T_j to P_m.

  - *Solution:* When P_m and P_q identify a conflict, they inform both the initiators of T_i and T_j. If *level*(T_i) > *level*(T_j) then both P_i and P_j select T_i as a candidate transaction. Therefore, *TDT*(T_j) is absorbed by *TDT*(T_i) and merging from the link $P_m \rightarrow P_q$ starts. Otherwise, *TDT*(T_j) is absorbed by *TDT*(T_i) and merging from the link $P_q \rightarrow P_m$ starts. The merging process is the same as described in the friendly resolution.

  b *Case 2:* A participant P_m of T_i receives T_j from a participant P_q of T_j.

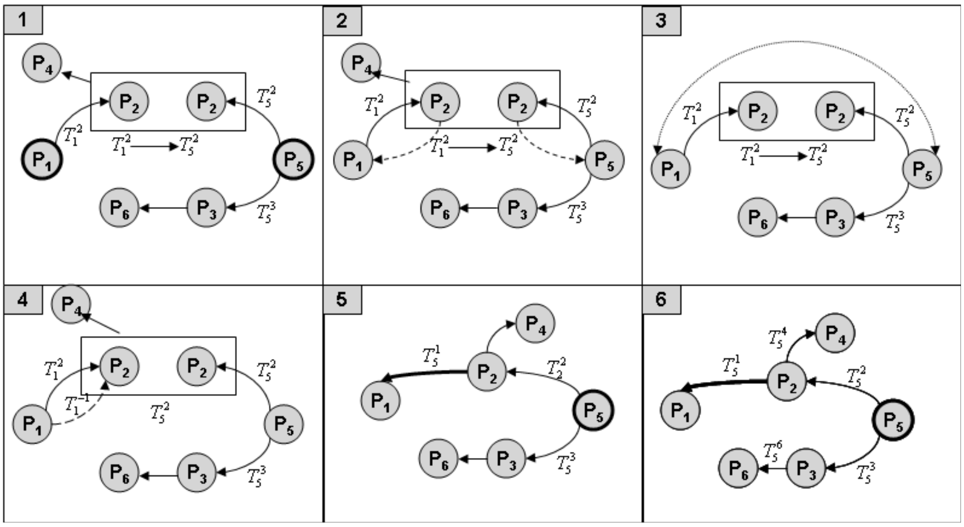
  - *Solution:* When P_m identifies the conflict, it informs both the initiators of T_i and T_j. Now, the same solution is applied as described in Case 1.

In the discussion above, we only consider the situation when two transactions are conflicting. However, there are several critical situations may occur. For example, a transaction may involve in conflict with multiple transactions during the construction of TDT or the transaction may involve in conflict during merge process with another transaction. In the later case, the execution of the new transaction is suspended until previous resolution decision is made. For example, a transaction T_k conflicts with a transaction T_j and T_j is in merge process with T_i. In this case the execution of T_k is suspended. After the merge process of T_i and T_j is finished, the conflict resolution between T_j and T_k is started. If T_j becomes the candidate then merge starts with T_j otherwise it will merge with T_i.

The first case is little bit complex. For example, at peer P_n, T_i conflicts with T_j and T_k. If the conflict between T_i and T_j happens before the occurrence of a conflict between T_i and T_k, then the conflict between T_i and T_j is resolved. If the conflict happens simultaneously, then P_n informs both the conflict information to the initiator of T_i. The initiator of T_i decides which one should be resolved first by considering the levels of T_j and T_k. The other transaction is suspended. The situation becomes more critical when conflicts occur in two different peers participating in the construction of $TDT(T_i)$. For example, at P_n, T_i conflicts with T_j and at P_m, T_i conflicts with T_k. Also in this case, the initiator of T_i decides which one should be resolved first by considering the levels of T_j and T_k.

In the following, we show an example of the candidate transaction selection.

Figure 6 Selection of a candidate transaction



Consider Figure 6 where two peers P_1 and P_2 originated two conflicting transactions T_1 and T_5 in the network.

- *Step 1:* P_1 has generated a component transactions T_1^2 from T_1 for peer P_2 and forwarded to P_2. Meanwhile, P_5 also generated two component transactions T_5^2 and T_5^3 from T_5 for peers P_2 and P_3 and forwarded to them. Assume that T_5^2 executed before T_1^2 at P_2. Therefore, the conflict relation between T_1 and T_5 at P_2 is $T_1^2 \rightarrow T_5^2$. P_3 also executed T_5^3 and waits for the forward message to forward T_5 to P_6.
- *Step 2:* After detecting the conflict, P_2 sends the conflict information to the initiators of T_1 and T_5, i.e., P_1 and P_5.

- *Step 3:* Initiators P_1 and P_5 decide the candidate transaction. In this case, $level(T_1) < level(T_2)$. Therefore, absorption protocol is applied, hence both P_1 and P_5 select T_5 as a candidate transaction.
- *Step 4:* P_1 sends a compensate transaction T_1^- to P_2. After the compensation phase, P_1 informs P_5 that the compensation of T_1 is complete. P_5 now sends a forward message to P_2 and P_3 to forward T_5.
- *Step 5:* After receiving the forward message P_2 forwards T_5 to P_1 and P_4. On the other hand, P_3 forwards T_5 to P_6.
- *Step 6:* Since, there is no new conflict and all the peers executed T_5, termination of T_5 starts and terminated successfully.

4.3 Discussion

In the proposed protocol, it does not require any global knowledge of a network topology for processing transactions. However, it seems that an initiator can become a bottleneck of the system since all participating peers need to connect to it before taking the next step to process a transaction. It is also possible that too many requests are sent to the initiator in a very short period of time. Also, the protocol may have a high complexity when several peers update the data at the same time. These seem to be the limitations of the approach.

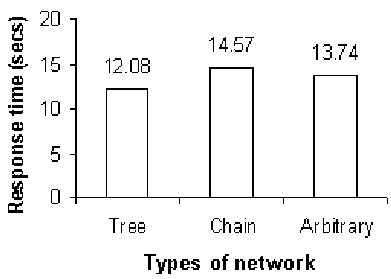
However, we assume that in a peer-to-peer database network, the global level transactions are not frequent and inconsistency of data in peers can be tolerated for the time being since transactions are not OLAP transactions. A transaction is only forwarded to its acquaintees only to resolve inconsistencies between peers. Moreover, one can claim that the design actually centralised since an initiator always need the global knowledge. However, this is not the case since we do not assume any dedicated controller who always monitor the global execution of transactions. Only the peer who initiates a transaction becomes the coordinator of the transaction during the execution period of the transaction in the system.

5 Evaluation

In this section, we show different experimental evaluations of the proposed transaction processing mechanism. In order to evaluate the performance over relatively large P2P settings, we implemented a simulator as a single java-based application. In the simulator, all peers are run within the same Java Virtual Machine. Each peer is implemented as a distinct thread and implements a FIFO queue for message communication. The environment consisted of a single Windows XP machine with Intel Pentium 4 CPU 3.40 GHz and 1 GB of RAM. Each peer is connected to a database that is instantiated as a MySQL 5.0 database. The experiments were evaluated with different size of networks, namely 100, 200, 300, 400, and 500. For each of the networks, the simulator generated schemata and contents of the peers' databases, as well as the peers' acquaintances. The operations of a transaction are MySQL select (read operation) and update (write operation) commands. Since all the peers ran on the same machine, there were no network delays. On the other hand, some delays were introduced because of database access times. To detect a conflict between transactions, we only consider write-write conflicts between transactions. Note that a conflict is considered in tuple level. Therefore, when a transaction executes in a peer, the conflict detection module determines the conflict based on the key value of tuples accessed by two transactions.

The first goal of the experiment is to compare the response time of a transaction in different types of networks, namely, in tree, chain, and arbitrary networks, which contain cycles, for evaluating the efficiency of the proposed protocol. The result of the evaluation is shown in Figure 7. The number of peers in the networks is 100. The size of the transaction is 5. The transaction size means the number of update operations in a transaction. We observe that the changes in response time of a transaction in different networks are not large. This is due the fact that each peer directly communicates with the initiator for processing a transaction. In a chain network, the response time is little higher, but compared to the time in other networks the change is not large. In a chain network, a transaction is executed along the chain of 100 peers. The initiator receives the final response lately from the last peer in the network. Overall, the response time deviates slightly in different types of networks, this proofs the efficiency of the protocol.

Figure 7 Response time of a transaction in different types of network



We also evaluate the response time of a transaction of the proposed protocol considering the different size of networks. The network size means number of peers in the network. For each network, the topology is arbitrary which may contain cycles. The result of the evaluation is depicted in Figure 8. We observe that response time increases linearly with the size of networks. This shows the scalability of the protocol. We are also concerned about the number of messages generated for executing a transaction in each network. The result is shown in Figure 9. We observe that number of messages increases linearly with the size of networks.

Figure 8 Response time of a transaction in different size of networks

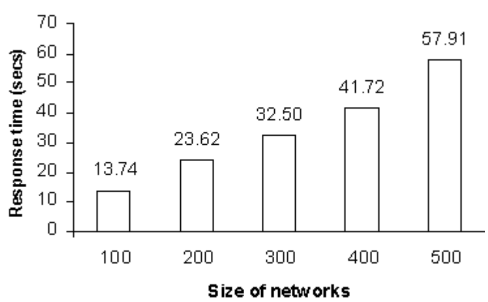
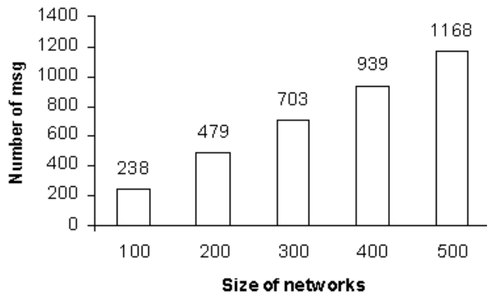
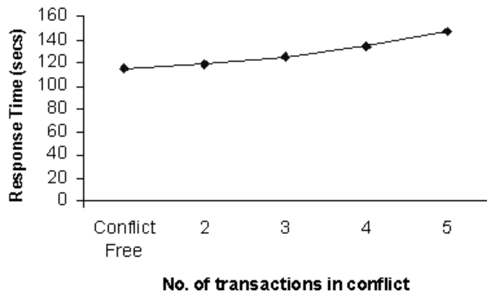


Figure 9 Number of messages in processing a transaction



We also evaluated the transaction processing protocol considering a conflict situation among transactions. Mainly, we wanted to observe, how the conflict resolution protocol affects the execution time of transactions. For this experiment, the transactions are generated concurrently from ten peers in a 100 peers network. The size of each transaction is 5. The transactions are generated in such a way that they involve in a conflict in increasing number. In the first case, there is no conflict among the transactions. We call it conflict free. In the second case two transactions are involved in a conflict, in the third case three transactions, and so on. We consider maximum five transactions are involved in a conflict. The result of the experiment is shown in Figure 10. Our observation from the result is that the execution time increases with increase number of conflicts but the impact on execution time is not a major inhibiting factor. We see that the execution time grows gradually with the increased number of conflicts. This shows the efficiency of the conflict resolution protocol.

Figure 10 Response time of transactions in conflict situations



6 Related work

In the following, we analyse some related works and show the differences with our model.

Haller et al. (2005) proposed a concept of transaction processing in P2P environment relying on a decentralised serialisation graph. In this model, each peer and each transaction maintain a local serialisation graph. The serialisation graph of the peer reflects the dependencies of the transactions that invoked service calls on that peer whereas the serialisation graph of the transaction includes the dependencies in which the transaction is involved. However, in our approach, peers are involved in resolving the conflict not the transactions themselves. This reduces the overhead of the transaction message. Moreover, in Haller et al. (2005), the peers that will be involved in processing a transaction is predetermined. Therefore, clients of a peer should have the global knowledge of the resources. However, in our framework, users are only aware of the local sources. Transactions are processed progressively in other peers in the network based on the mappings with the local peer where the transaction is submitted.

Cetintemel et al. (2003) proposed a decentralised peer-to-peer transaction approach in a replicated system. The protocol uses the concept of voting. The protocol assumes that number of peers is fixed and each peer owns an equally distributed currency value. The total value of the currency in the network is 1.0. A transaction commits in the system when it is guaranteed that no conflicting transaction can obtain more votes.

In our approach, the number of peers is unknown in the system; therefore no fixed currency can be applied in each peer. A candidate transaction is selected from the conflicting transactions using a leader election protocol. The transaction which becomes the leader finally executes in the network. Moreover, a transaction may not execute in all peers in the network. Therefore, we can not assume a fixed currency for each peer. A transaction acquires a level progressively during its execution. Based on the level a candidate transaction is selected.

Taylor and Ives (2006) proposed a database reconciliation mechanism in a decentralised collaborative data sharing environment. Here conflicts are resolved using the priority of updates and the provenance information. The approach requires centralised provenance information for resolving conflicts. Otherwise, same update may be accepted by one peer and rejected by another peer.

However, in our approach, initiators of the updates resolve a conflict using majority consensus policy.

Terry et al. (1995) proposed a replicated database system to support collaboration among users in a weakly connected network. Transactions are broadcast between sites using an epidemic propagation protocol. It first executes transaction in their tentative order, then rolls back and replays them in final order. If the transaction is accepted by a primary site, the final timestamp is assigned to the update. Hence, the final execution of transactions relies on a primary site that enforces a global continuous order on a growing prefix of history.

However, in our system there is no primary site. Every peer executes transactions and resolves conflicts independently. Also, in our system, updates are propagated along the mappings between peers. Using a primary site may as mentioned in Terry et al. (1995) constitutes a congestion point, and, anyway is not suitable in a peer-to-peer system.

In Androutsellis-Theotokis et al. (2004), the authors presented a preliminary proposal for a peer-to-peer e-business transaction processing system. More specifically, the paper focuses on requirements analysis of different aspects of the collaboration and transaction procedure. However, it lacks precise semantics of transactions and does not describe the execution semantics of transactions.

SchÄutt et al. (2008) presented a distributed key/value store based on the Chord structured overlay with symmetric data replication and a transaction layer implementing ACID properties. The protocol works very well for asynchronous collaborative applications where data are symmetric. In Mejas and van Ro (2010), the protocol has been extended to support eager locking, making it feasible to build synchronous collaborative applications. In both cases, locks are the only way to guarantee atomicity, concurrency control and strong consistency. Unfortunately, locks are not the best abstraction for P2P systems, and it is highly desirable to avoid them whenever possible.

Logoot (Weiss et al., 2009) is a scalable optimistic replication algorithm for collaborative editing on P2P Networks. Logoot ensures causality, consistency and intention preservation criteria. In Logoot, a single operation is considered. It mainly works with replicated system where multiple edit transactions may execute concurrently on the same data item. However, we consider a database transaction model which is consists of many data operations. Our approach works with a P2P data sharing environment where data is shared but not replicated.

7 Concluding remarks

In this paper, we present an approach for executing concurrent transactions in a peer-to-peer database network. Our approach is scalable because a participant does not need any global knowledge of the execution status of a transaction and there is no global coordinator. Transactions are processed by each peer independently. Only the initiators of the transactions are responsible for monitoring the global execution of the transactions. We also present a candidate transaction selection protocol from the conflicting transactions that run in the network concurrently.

A future goal is to investigate the transaction processing considering the dynamic behaviour of peers. Further, we are interested to propose a recovery mechanism of transaction in a peer-to-peer database network. We also plan to investigate the efficiency of the protocol considering a large network by comparing with existing protocols.

References

Androutsellis-Theotokis, S., Spinellis, D. and Karakoidas, V. (2004) *Performing Peer-to-Peer E-business Transactions: A Requirements Analysis and Preliminary Design Proposal*, IADIS.

Bernstein, P., Hadzilacos, V. and Goodman, N. (1987) *Concurrency Control and Recovery in Database Systems*, Addison Wesley, Reading, MA.

Breitbart, Y. and Silberschatz, A. (1988) *Multidatabase Update Issues*, ACM SIGMOD.

Breitbart, Y., Garcia-Molina, H., Silberschatz, A. (1992) 'Overview of multidatabase transaction management', *VLDB Journal*, Vol. 1, No. 2.

Cetintemel, U., Keleher, P.J., Bhattacharjee, B. and Franklin, M.J. (2003) 'Deno: a decentralized, peer-to-peer object-replication system for weakly connected environments', *IEEE Transactions on Computers*, Vol. 52, No. 7.

Dayal, U., Hsu, M. and Ladin, R. (1991) *A Transactional Model for Long-running Activities*, September, VLDB, Barcelona, Spain.

Ganarski, S., Naacke, H., Pacitti, E. and Valduriez, P. (2007) 'The leganet system: freshness-aware transaction routing in a database cluster', *Information Systems*, Vol. 32, No. 2.

Garcia-Molina, H. and Salem, K. (1987) *Sagas*, ACM SIGMOD.

Gray, J. and Reuter, A. (1993) *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers.

Halevy, A., Ives, Z., Suciu, D. and Tatarinov, I. (2003) *Schema Mediation in Peer Data Management System*, ICDE.

Halevy, A.Y., Ives, Z.G., Madhavan, J., Mork, P., Suciu, D. and Tatarinov, I. (2004) 'The piazza peer-data management system', *IEEE Transactions on Knowledge and Data Engineering*, Vol. 16, No. 7.

Haller, K., Schuldt, H. and TÄurker, C. (2005) *Decentralized Coordination of Transactional Processes in Peer-to-Peer Environments*, CIKM.

Kementsietsidis, A., Arenas, M. and Miller, R.J. (2003) *Mapping Data in Peer-to-Peer Systems: Semantics and Algorithmic Issues*, SIGMOD.

Masud, M. and Kiringa, I. (2007) *Acquaintance Based Consistency in an Instance-Mapped P2P Data Sharing System During Transaction Processing*, CoopIS.

Mejas, B. and van Ro, P. (2010) 'Beernet: building self-managing decentralized systems with replicated transactional storage', *International Journal of Adaptive, Resilient and Autonomic Systems (IJARAS)*, Vol. 1, No. 3, pp.1–24.

Moss, J. (1985) 'Nested transactions: an approach to reliable distributed computing', PhD thesis, MIT Press, Cambridge, MA.

Rodriguez-Gianolli, P., Garzetti, M., Jiang, L., Kementsietsidis, A., Kiringa, I., Masud, M., Miller, R. and Mylopoulos, J. (2005) *Data Sharing in the Hyperion Peer Database System*, VLDB.

Santoro, N. (2006) *Design and Analysis of Distributed Algorithms*, Wiley InterScience Publisher.

SchÄutt, T., Schintke, F. and Reinefeld, A. (2008) 'Scalaris: reliable transactional p2p key/value store', *ERLANG'08: Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, pp.41–48.

Schuldt, H., Alonso, G., Beeri, C. and Schek, H. (2002) 'Atomicity and isolation for transactional processes', *ACM Trans. Database System*, Vol. 27, No. 1.

Serafini, L., Giunchiglia, F., Molopoulos, J. and Bernstein, P. (2003) *Local Relational Model: A Logocal Formalization of Database Coordination*, Technical report, Informatica e Telecomunicazioni, University of Trento.

Taylor, N.E. and Ives, Z.G. (2006) *Reconciling while Tolerating Disagreement in Collaborative Data Sharing*, SIGMOD.

Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J. and Hauser, C.H. (1995) 'Managing update conflicts in Bayou, a weakly connected replicated storage system', *Proc. of the ACM Symposium on Operating Systems Principles*.

Weiss, S., Urso, P. and Molli, P. (2009) 'Logoot: a scalable optimistic replication algorithm for collaborative editing on P2P networks', *29th IEEE International Conference on Distributed Computing Systems*, pp.404–412.