Hindawi Publishing Corporation
Journal of Computer Networks and Communications
Volume 2012, Article ID 782517, 14 pages
doi:10.1155/2012/782517

Research Article

Deco: A Decentralized, Cooperative Atomic Commit Protocol

Daniel J. Buehrer and Chun-Yao Wang

Department of Computer Science and Information Engineering and Institute of Computer Science and Information Engineering, National Chung Cheng University, No. 168, Sec. 1, University Rd., Min-Hsiung Township, Chia-yi County 621, Taiwan

Correspondence should be addressed to Chun-Yao Wang, wcy@cs.ccu.edu.tw

Received 3 April 2012; Accepted 11 September 2012

Academic Editor: Liansheng Tan

An atomic commit protocol can cause long-term locking of databases if the coordinator crashes or becomes disconnected from the network. In this paper we describe how to eliminate the coordinator. This decentralized, cooperative atomic commit protocol piggybacks transaction statuses of all transaction participants onto tokens which are passed among the participants. Each participant uses the information in the tokens to make a decision of when to go to the next state of a three-phase commit protocol. Transactions can progress to ensure a uniform agreement on success or failure, even if the network is partitioned or nodes temporarily crash.

1. Introduction

1.1. Network Computing. Network-based computing has several programming frameworks, such as peer-to-peer (P2P), grid, and web service computing [1]. For applications which involve many companies or individual users with their own databases (e.g., network supply chain or distributed health-care monitoring), the P2P approach seems to be a natural infrastructure. Each company or user can set the security on their local databases and decide which data to share with other members of the distributed application, based on an authenticated query sender (i.e., application executor).

Some network applications will require the ability to execute actions involving several databases, such as deciding which products to purchase in a supply chain or a battlefield command/control decision on whether or not there are sufficient resources for an attack to succeed. It has been proven [2] that such network-wide decisions can be delayed (i.e., blocked) for an indefinitely long time as the members of a transaction alternately fail and recover, or as the network communications suffer temporary disconnections. It is necessary to make the assumption that eventually all computers and network connections will be working long enough for a decision to be made, even though this assumption may not be true in the real world. In the case of long-term failures, our protocol will usually time out and abort. However, as we see below, there is a small possibility of blocking when the clocks are disabled during the final commit decision.

In the P2P environment, it has long been recognized that providing a persistent, consistent distributed commit protocol is a very important and difficult issue [3]. Many e-business systems, such as electronic reservations, funds transfer, and inventory control, have benefited from distributed atomic commit protocol (ACP) technology. These ACP protocols typically work on a multitier architecture on a WAN, (e.g., [4–7] and in the study of [8]), and they use a coordinator to handle the processing of the transactions. Unfortunately, these protocols will be difficult or even impossible to apply in decentralized environments (e.g., mobile, cloud, and ad hoc) due to the higher probability of failures, including host crashes, message loss, and network partitioning [9]. Also, a coordinator becomes a bottleneck for passing the information to/from participants. In the real world, UDP and low-level broadcasts are often blocked by routers, and the coordinator is forced to use a sequence of messages instead of broadcast or multicast.

1.2. Eliminating Bottlenecks. One of the major sources of blocking is when the coordinator of a transaction becomes unreachable. Participants must block until they are able to reach the coordinator. If a new coordinator is elected, and the old coordinator recovers and is not able to reach the new coordinator, there may be a "split-brain" syndrome where both coordinators are temporarily active.

Because of these problems with using a coordinator, this paper describes how to implement distributed transactions which do not require a coordinator. This method is based on the idea of "virtual synchrony" [10] for distributed computing. Virtual synchrony bases distributed computing on the states of a finite-state automaton, where loops are not permitted. All participants of a transaction progress asynchronously from one state of the automaton to the next and eventually toward a final state. The information about the states of other participants can be used to make decisions locally. For instance, we will use a 3-phase commit protocol in which a local participant goes from the precommit state to a commit state only when all other participants have also finished precommitting. They precommit by writing their transactions to persistent storage, so that they can be sure of being able to finish committing or aborting the transaction even if their power goes off temporarily.

One question is how to share the information about other members of the transaction. This can be solved by using an application-level multicast (ALM) protocol. Various ALM protocols [11, 12] have been developed for P2P overlay networks. For instance, video may be sent through a tree with decreasing baud rates as you go down to slower computers and networks. From the research of [13, 14], a ring structure has proven to be a simple structure easily capable of skipping nonreachable nodes, although a tree of rings would result in less round-trip delay time.

The major advantage of ALM protocols over low-level multicast is that they can be very complex, using all of the information about geographic locations, loading of nodes, stress of network connections, probabilities of failure, summaries of message probability distributions, and so forth.

When a large transaction is initiated by any node, there could be thousands of members for the transaction. Much research has shown that using a specific host to send information simultaneously from/to large number of members of a transaction is impractical [12, 15–17]. But from the study of [4, 18–20], all of the proposed distributed ACP protocols were designed with specific coordinators. Even with an ALM broadcast, such a coordinator can easily become a bottleneck unless the ALM provides a means of merging (i.e., accumulating) the returned results.

1.3. Contributions. In this paper, we propose a decentralized, collaborative transaction protocol for distributed transaction processing. This paper continues the work in [21]. It comes from the need to provide a decentralized and collaborative transaction protocol for Cadabia (Class Algebra Data Base and Intelligent Agent). A rough description about Cadabia is provided in Section 2.1. In addition to [21], we provide a correctness proof, theoretical analysis, and new simulation results for our protocol. To the best of our knowledge, this is the first proposal for using the token accumulation approach to design a decentralized atomic commit transaction protocol. Our protocol has the following characteristics which can shorten the response time of a distributed transaction and eliminate the bottleneck of messages of a centralized ACP.

(i) State vector: We define a vector of partially ordered states, called a "State Vector", to represent the votes of all participants. The term "participant" means a host who participates in a transaction. Benefiting from the monotonic, nondecreasing values of the state vector, each participant can merge multiple tokens into a single token (by using the "max" of each participant's state) to get the most recent statuses of all participants, without worrying about the order of receipt of the tokens.

(ii) Token accumulation approach: Each participant saves its latest state into the token and propagates the token among the other participants. Therefore, each participant can evaluate global decisions locally and collaboratively according to the state vectors that were piggybacked on the token. The participants do not have to wait to contact unreachable participants before making decisions.

(iii) Can work with any multicast protocol: There is much research on how to multicast messages to a group in a decentralized system [22]. Since we do not care about the order of message arrivals, our protocol is suitable for any efficient ALM protocol that can eventually propagate correct messages to all members.

The remainder of this paper describes our model, algorithm, and implementation. In Section 2 we introduce the system model. In Section 3 we describe the protocol in detail. In Section 4 we provide the correctness proof of our protocol. Section 5 describes the implementation and experiments. Finally, in Section 6 we state our conclusions.

2. System Model

2.1. The Background of Cadabia. Cadabia is browser-, OS-, and machine-independent. It is an object-oriented, P2P middleware storage system [23, 24]. Cadabia uses network-wide object identifiers for both objects and their OWL-like class definitions. Like other SQL/XML middle-ware, such as Google Web, the objective of Cadabia is to share components and services across a wide-area network. Persistent storage agents like Cadabia should provide certain basic services. Cadabia's inner kernel defines classes, object identifiers, attributes, binary relations, and Java-based methods for query and updates. Successive kernel layers will add transactions, security, caching, failover and load balancing, and backup/recovery for cloud computing.

A Cadabia query starts from the persistent objects in the extent of a given class, and then follows filtered binary relations to other persistent objects. The explicit binary relations are all stored as 2-way pointers, and implicit

relations are defined as sequences of binary relations, so they can also be traversed in either direction. The two-way pointers are used to do garbage collection within a short-term transaction, thus preventing dangling references and memory leaks. The relationships are stored as sorted object identifier (i.e., Oid) sets, that is, pointers to persistent objects on possibly thousands or millions of computers. Cadabia's fetch plans allow queries to obtain statistically significant outcomes, even though many clients with a similar ontology may be unreachable. Unlike most other Java-based approaches (e.g., JDO), no attempt is made to hide persistency from programmers. Attributes can store/retrieve any Java serialized values.

Cadabia's first-level kernel offers programmers short-term transactions, which are then used to implement long-term transactions. As well as coordinating the 2-way pointer updates of relationships, short-term transactions are also used for coordinating changes to the members of a long-term transaction. A short-term transaction is used to make sure that all current members agree on any acquaintances [25, 26] that have joined or left the transaction, so that all members have the same membership lists. This is especially important for mobile and cloud computing environments, where changes in membership must be coordinated fairly often. Long-term transactions are implemented as a subscription service, with members receiving update notifications of attributes or relationships that have been read or updated within the long-term transaction. The updated query state can be cached until the long-term transaction is completed. When the User Interface component showing the queried information is closed, the transaction can be committed, and the cached information can be deleted.

2.2. Assumptions and Environment. We consider a decentralized, P2P, asynchronous system model. A set of nodes, say TP, represent all of the peers. The peers can act as both clients and back-end servers. All nodes $\in$ TP have a unique identifier, and they can locate each other through a lookup protocol, such as [27, 28]. In Cadabia, a query consists of a class followed by relations interspersed with selections. The "import" relationships among ontologies determine the possible ranges of the relations, including the peers which may have appropriate data. "Fetch plans" may be used to limit the number of objects in the result, or the number of servers queried.

A User Interface (UI) component on the client may reflect the results of querying/updating data on one or more peers. We would like this data to remain consistent, but showing changes to those attribute/relationship values that cause objects to join/leave the query result. A short-term transaction can be used to get a consistent snapshot of the peers, temporarily locking any seen queried attributes/relationships until the commit/abort operation. A long-term transaction is used to guarantee delivery of updates.

The result of traversing a relationship is a set of object identifiers (Oids), and these Oids indicate the peers on which they are stored. Each traversal of a relationship results in a unique set of Oids, and thus a unique set of nodes. The client node $p_c \in$ TP queries the "import" relationships among the ontologies and the range of the given relations to find the potential peers in those ranges. The import and range relationships are locked until the short-term transaction initializes the set of participants of a long-term transaction. The short-term transaction can then commit, freeing the locks on the relationships and attributes. The long-term transaction must still use the backend databases to write changes to these attributes/relationships until the transaction is committed or aborted.

A UI component may make many queries and updates, and the peers involved in these queries are all added to the long-term transaction's participants by using short-term transactions to update the participants on all peers. In this paper, we assume that the short-term transactions have a fixed set of participants. Let the short-term transaction participants be represented by $\mathrm{TP}_{\mathrm{TRN}} = \{p_0, \ldots, p_{n-1}\}$,$\mathrm{TP}_{\mathrm{TRN}} \subseteq$ TP and $|\mathrm{TP}_{\mathrm{TRN}}| = n$. We assume that p_c can find an efficient way to route messages to all members in $\mathrm{TP}_{\mathrm{TRN}}$ by using an ALM protocol, such as [11, 12, 15], where every node acts as both a router and a forwarder for messages sent by other nodes.

Short-term transactions can be used within long-term transactions to coordinate updates to relations and attributes on a fixed set of peers. The long-term transactions can have a clock which is updated every time that a short-term transaction succeeds in updating the participants on all peers. The clock can be used to ignore this transaction's old messages with their old *dblists*. Other than this, most of the codes for reaching consensus on long-term transactions are similar to the codes for short-term transactions, which we describe below. However, a long-term transaction should also allow a UI component to check if its values are out-of-date.

Notice that a long-term transaction still has to maintain the transaction log so that it can either commit all changes or abort to the original state. Similar to [1], we do not discuss the actual database operations within a traditional transaction; we assume each node works on top of a local database system which satisfies the ACID properties, as in the Cadabia architecture, and we use these databases as a basic service to save/retrieve data and transaction information. So, for instance, if a back-end says that it has successfully precommitted, this means that the operations on the database have been backed up onto a transaction log. The transaction is guaranteed to be capable of successfully committing or aborting by using that transaction-log data, even if the power goes off and the server has to reboot.

All of the nodes satisfy the crash/recovery model of [29], where it is assumed that nodes will recover from crashes and eventually stop crashing long enough to successfully contribute to the computation of a decision value. We can then prove below that if the participants of the transaction keep retrying, they will eventually be able to reach a commit/abort decision and communicate it to all participants.

The network may partition into several groups due to network failure, but some or all of the partitioned groups will subsequently remerge. Participants may become detached

due to crash failure or network partition. Message omissions can occur, but not every received message will be corrupt. We assume that all participants are friendly, and that no Byzantine failure will occur. No participant will be allowed to falsify other participant's statuses, and all the participants in the transaction must run their protocols faithfully. Moreover, there should be some way, such as arbitrary CRC checks and message digests, to make sure that the code on all peers is the same.

2.3. e-Transactions, Web Consensus. In traditional transactional systems, a transaction has four properties that must be satisfied: *Atomicity*, *Consistency*, *Isolation*, and *Durability*. These four properties have the acronym, ACID properties [30]. In addition to the ACID properties, a recently proposed abstraction called an e-Transaction (exactly-once Transaction) [4] takes the whole process of a network interaction into consideration. It proposes seven properties to ensure that whenever a client issues a request, the application server will compute a corresponding result, the database servers will eventually commit or abort the result, and the result will eventually be delivered to the client. The seven properties belong to three categories: *Termination*, *Agreement*, and *Validity*. The *Termination* property means that unless the client crashes, if the client delivers a request, it will eventually receive an outcome, and if any database server votes for a request, it will eventually commit or abort the request. The *Agreement* property means that all the database servers that join the transaction will reach a uniform agreement, and no server will commit more than once for the same request. Also, unless the database servers have all committed the request, no result will be delivered to the client. The *Validity* property means that if a client sends a request and receives an outcome for the request, then the request must have been computed by the application server, and unless all database servers can vote to commit the request, no database server will commit the request [4] until it knows that the other database servers have precommitted the request, which means that they can guarantee that they can complete the request even if they temporarily crash.

In a P2P system, each node will behave as both a client and a server, so each node will play the role of both an application server and a database server as specified in the e-Transaction properties. In this paper, we will focus on how to satisfy the e-Transaction properties for a decentralized transaction protocol that uses an ALM on a P2P overlay network.

2.4. Scenario and Connection Topology. The execution scenario of a transaction is as follows. The p_c issues a request and evaluates a set of participants that need to join the transaction. The p_c produces a *dblist* of participants, each with a state vector. The state vectors are sorted according to the identifiers of the participants. Then p_c generates a token containing the *dblist* and starts a transaction by delivering the token to all participants according to the ALM algorithm. The participants manage data and execute the transaction operations, save their own transaction information into the token, learn about the current statuses of other participants from the token, make a decision, and save and forward the token to the other participants. If a sender p_i suspects a receiver p_j to have failed, p_i can bypass p_j, and deliver the token to the "next" participant according to the multicast overlay policy. If p_i is not able to deliver the token to any participant, it will suspend the delivery for a given period of time, and then resume the delivery. This process will continue until reaching a uniform agreement to *commit* or *abort*.

During the compute phase, no matter which protocol is used for multicast, there exists a propagation path with the largest cost (distance) between two participants. If there are not any branches in this path, then this is a linear topology, which is also the worst-case topology. In order to simplify our discussion of worst-case runtimes in this paper, we assume that all of the participants of a transaction are connected with a linear overlay topology. When a sender fails to forward a message to a receiver on the current path, the sender may skip it and forward the message to the next receiver. We note again that, benefitting from the "State Vector" and token accumulation approach, our protocol can work correctly with any connection topology that eventually correctly propagates messages to all members.

2.5. The State Vector and Tokens. In a transactional system, unlike a synchronous totally ordered broadcast [27, 31–33], all participants should proceed with the transaction and make decisions regardless of the order in which messages are sent/received. For example, a transaction decision can change a participant's status from "prepared for transaction" to "commit" if and only if all other participants have "prepared for the transaction" statuses. In order to let the participants make a decision without an explicit coordinator, we use a "*State Vector*" which contains each participant's transaction state and an integer clock counter. When the participant experiences an internal event, it will increase the clock counter of its element in the state vector. With this mechanism, every participant can merge multiple tokens into a single token to obtain latest information about all participants. Transaction states with smaller clock values are replaced by those with larger clock values.

The state vector tells the state of all participants. Each element is composed of (1) an integer clock counter, (2) the transaction status of a given participant, as discussed later, (3) the "outcome received flag" (OR_F), that will be turned on after the participant makes a decision "Transaction Committed" (S_TCd) or "Transaction Aborted" (S_TAd). Such decision codes (i.e., automaton edges) are described in Section 3. In this section we first describe the states (i.e., automaton nodes).

A token carries the transaction information, which consists of the following. (1) The transaction identifier. (2) The *dblist*, which contains the state vector (3) The "outcome delivery flag" (OD_F), which will be turned on after the outcome is delivered to p_c successfully.

As for the transaction state of each participant, "TS" for short, we have the following set of permissible values.

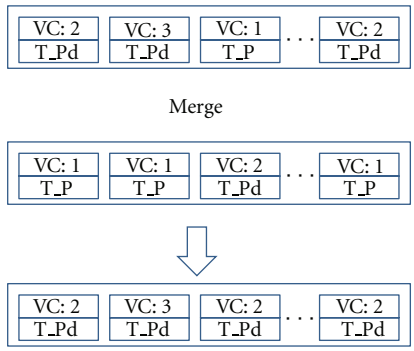

Figure 1: Merging tokens.

(TS_R1) Trans_Prepare (T_P). This vote means that the participant received a token which asked it to join this transaction. The participant will also set up a timer for the transaction.

(TS_R2) Trans_Prepared (T_Pd). This vote means this participant is ready to commit the transaction.

(TS_R3) Trans_Abort (T_A). The participant decided to abort the transaction, and it will execute the abort procedure right away. Any participant whose timer has expired will also vote "T_A".

(TS_R4) Trans_Aborted (T_Ad). The participant has finished aborting the transaction.

(TS_R5) Trans_Commit (T_C). The participant has found that all members are prepared ("S_TPd"—see Section 3). It will vote "T_C" and disable the timer.

(TS_R6) Trans_Commit_ReadOnly (T_CR). This means the transaction is read-only for the participant, and no data will be locked. No recovery procedure for this participant is needed.

(TS_R7) Trans_Committed (T_Cd). This vote means that the transaction has been successfully committed on this participant.

(TS_R8) Trans_Null (T_N). This vote means that this participant has not yet voted for the transaction.

We let each participant establish a queue for every distinct transaction as the part of its flow control strategy; the received tokens will be inserted into the queue according to the transaction identifier that the token carries. We define a primitive method "*Token merge (Token, Token)*" for each participant. This merges two tokens into a single token by replacing each smaller state clock by the larger one, and choosing the value that is stored with the larger clock value. This process is shown in Figure 1.

3. The Deco Protocol

3.1. The Deco Protocol and the Transaction Decision Codes. We did not introduce the transaction decision codes in Section 2 because we need a full section to discuss what these codes mean. To let the participants work collaboratively, we define a set of constants that represent the possible decisions within the transaction, called the "*Transaction Decision Codes*" (TDC). These are the edges of a partially ordered (i.e., acyclic) finite state automaton.

Each TDC code has an associated evaluation condition. When this condition is satisfied, the edge is traversed to the next state in the finite state automaton. For the participant $P_i \in \mathrm{TP_{TRN}}$, we use the primitive method "TDC getDec(P_i)" to obtain the decision of P_i, and we use the primitive method "TS getstate(P_i)" to obtain the Transaction State of P_i.

Before further discussion, let us first introduce the TDC definitions for our protocol. The transition graph of TDC is shown in Figure 2.

(P_R1) S_Trans_Abort (S_TA). If $\exists p_i \in \mathrm{TP_{TRN}}$, and getState($p_i$) $\subseteq$ Trans_Abort, Trans_Aborted, then the abort procedure is started locally.

(P_R2) S_Trans_Aborted (S_TAd). If for all $p_i \in \mathrm{TP_{TRN}}$, getState($p_i$) $\subseteq$ Trans_Aborted, then turn on the outcome received flag (OR_F).

(P_R3) S_Trans_Prepare (S_TP). If for all $p_i \in \mathrm{TP_{TRN}}$, getState($p_i$) $\subseteq$ Trans_Prepare, Trans_Prepared, Trans_Null, Trans_Commit_ReadOnly .

(P_R4) S_Trans_Prepared (S_TPd). If for all $p_i \in \mathrm{TP_{TRN}}$, getState($p_i$) $\subseteq$ Trans_Prepared, Trans_Commit, Trans_Commit_ReadOnly.

(P_R5) S_Trans_Commit (S_TC). If for all $p_i \in \mathrm{TP_{TRN}}$, getState($p_i$) $\subseteq$ Trans_Commit, Trans_Committed, Trans_Commit_ReadOnly, the commit procedure can be started.

(P_R6) S_Trans_Committed (S_TCd). If for all $p_i \in \mathrm{TP_{TRN}}$, getState($p_i$) $\subseteq$ Trans_Committed. Each node that concludes "S_TCd" will turn on the outcome received flag (OR_F).

In a decentralized and distributed system model, some tokens received by nodes will contain old states of some participants. Therefore, if we use a timer and allow participants to abort a transaction when the timer expires, we cannot reach a uniform agreement by a majority vote based on the received tokens. For example, when $p_1 \in \mathrm{TP_{TRN}}$ evaluates a token it received, it may find that all of the participants have voted "T_C", but, at the same time, there may be a participant p_a who votes "T_A" due to its timer expiring. If we allow p_1 to execute the commit process as soon as it votes "T_C", then $\mathrm{TP_{TRN}}$ will not reach a uniform agreement.

To prevent our protocol from having the above situation happen, we add the constraint that each participant who votes "T_C" must first disable its timer. This participant must block until all participants vote "T_C" (i.e., "S_TC" is concluded), or until it finds some participant who voted "T_A" (i.e., "S_TA" is concluded). The commit procedure of a participant will be executed if and only if it concludes "S_TC" (i.e., all participants have voted "T_C"). Thus, the participants who vote "T_C" will not vote "T_A" unless some others vote "T_A". If "S_TC" is concluded, it means that all participants have voted "T_C", and this implies that no participant will ever vote "T_A" again. For the same reason, if

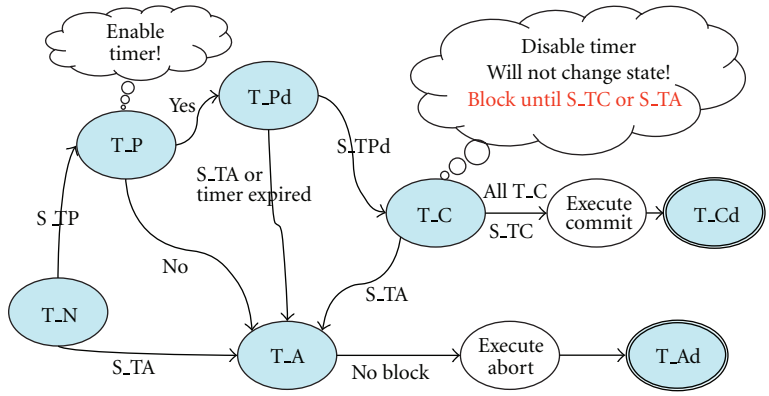

Figure 2: The TDC transition graph.

some participant had voted "T_A" for any reason, "S_TC" will never be concluded, and this implies that "S_TCd" will never be concluded, either. According to the above discussion, we can conclude that all participants will reach a uniform agreement.

A participant who did not vote "T_C" will not be blocked due to other participants' crash failures or network partitions. Instead, his timer will expire. However, when all participants vote "T_C" except for one, say p_f, and p_f crashes or becomes unreachable due to network partition failure, there is a time period T for which p_f cannot contribute to the computation of a decision value. Then all participants will block until p_f recovers, which may be unacceptably long.

We can improve this situation by using a heartbeat technique. Each participant who votes "T_Pd" should increase the counter in its state vector (i.e., send a heartbeat) before it forwards the token. After voting "T_Pd" and before voting "T_C", each participant should keep tracing the counters in state vectors of all participants. Each time a token arrives, if it shows that all the counters in state vectors do increase, this implies that all participants are still reachable. The timer is only turned off if all participants still have beating hearts, which means that they are less likely to crash while this participant's timer is turned off.

3.2. The Behavior of Participants. We have introduced the execution scenario of a transaction in Section 2.3. As we mentioned before, each participant uses a distinct queue of tokens each distinct transaction. For each transaction identifier, the participants also keep a copy of the token that has been processed last, say T_{SAV}, so that it can keep track of the current status of the transaction. Here, we use a composite data structure, called the "*Transaction Decision Structure*" (TDS) to group related information for each distinct transaction. The TDS contains the transaction identifier, the T_{SAV}, the queue, and a semaphore to synchronize the node and the threads created by nodes when they share the queue. Note that we assume that the content of the TDS will be mirrored to a stable storage to prevent loss of tokens from server crashes.

Algorithm 2 shows the behavior of each participant. When each participant starts, it will first enter the recovery mode. By restoring the TDS, each participant can find all unfinished transactions, and call the "*Proceed_Trans_Thread* (TDS)" primitive to process the unfinished transactions. After the recovery procedure, each participant will start to listen for any incoming tokens. As soon as it receives a token, it will put the token into a queue according to the token's transaction identifier. When the participant tries to retrieve a token from the queue but fails (i.e., the queue is empty), it will set up a timer, the "rec_timer" in Algorithm 2, and sleep until awoken by receipt of a token or by a timer-expired interrupt. If the thread awakens due to timer expiration, it will retransmit its tokens again to prevent the token from being lost due to network or participant failures. If the thread retrieves a token, say T_{REC}, successfully, and if T_{SAV} exists, it will invoke the primitive "*Token merge (Token, Token)*", whose abstraction is shown in Figure 2, to merge these two tokens and save the result to T_{SAV}. This process will be continued until the queue is empty. After merging the transaction tokens, the server will forward the T_{SAV} token to the next participant by using the "*void forward (Token)*" primitive which is described in Algorithm 1 and then evaluate the transaction decision according to the TDC rules. When the sender, say p_i, fails to deliver a token to a receiver, say p_R, the sender can bypass p_R and find the sibling of p_R, say p_{R1}, and deliver the token to p_{R1}. This process will repeat until the delivery is successful or p_i finds it has failed to deliver the token to all other participants. The OD_F flag is used to indicate whether p_c has received the result; if not, the participant will try to deliver the result to p_c and turn on OD_F if the delivery is successful.

Finally, a transaction starts due to p_c issuing a request. After the delivery is successful, the p_c will block and wait to receive the result of the transaction. This process will continue until the client receives a result of "S_TCd", which means that the transaction was successfully be committed by all participants. Algorithm 3 shows the behavior of p_c.

```
Token handling by p_i ∈ TP_TRN, 0 ≤ i < n.
void forward(Token) {
   Use the ALM protocol to evaluate a receiver p_R;
   do {
        Forward the message to p_R;
        If p_R is suspected unreachable during a predefined period {
             Use the ALM protocol to evaluate a another receiver and it to p_R;
        }
   } until (the delivery succeeds or p_R == p_i)
}
```

Algorithm 1: Rule to forward a token.

```
Token handling by p_i ∈ TP_TRN, 0 ≤ i < n.
p_c starts the transaction.
/* Transaction is finished ⇔ All OR_F in dblist and OD_F are turned on. */
Void run() {
   On recovery do {
      Restore the TDS of unfinished transactions from stable storage;
      For each TDS, Start Proceed_Trans_Thread(TDS);
   } /*recovery end*/
   While (True) {
      Block to accept a token;
      Put token into queue of related TDS;
      Signal or start related Proceed_Trans_Thread(TDS)
   } /*while (true) end */
}
Void Proceed_Trans_Thread(TDS) {
   Retrieve queue and T_SAV from TDS;
   While (Transaction not finished) {
      While queue is empty {
         Setup a rec_timer, sleep until receive a signal or timer expires;
         If awake due to rec_timer expiration, then Call forward (T_SAV);
      }
      While queue is not empty {
         get T_REC from queue and Merge T_REC to T_SAV;
      }
   Call forward (T_SAV);
   According the rules of the protocol described in Section 3.1
            evaluate decision and start to process related job;
   }//while
}
```

Algorithm 2: Behavior of a participant.

4. The Deco Protocol Correctness

In this section, we will provide a proof of correctness for our protocol with respect to the e-Transaction properties which were stated in Section 2.2. In a P2P, decentralized architecture, nodes can act as clients, application servers, and database servers. For the reader's convenience, we recall the e-Transaction properties and make them suitable for P2P architecture. In the following proof, we use "p_c" to represent the node that issues a request, and $\mathrm{TP}_{\mathrm{TRN}}$ is the set of participants that join the transaction, where $|\mathrm{TP}_{\mathrm{TRN}}| = n$, and "$p_i$" represents the participant, $p_i \in \mathrm{TP}_{\mathrm{TRN}}$.

Termination

(T.1) If the p_c (client) issues a request, then, unless it crashes, the p_c (client) eventually delivers a result.

(T.2) If any participant (database server) votes for a result, then all participants (database servers) eventually commit or abort the result.

```
TP_TRN is the set of participants, |TP_TRN| = n, p_i ∈ TP_TRN, 0 ≤ i < n.
dblist contains the set of state vectors of all p_i ∈ TP_TRN.
void main() {
    Result result = null;
    While (result != "S_TCd") {
        Generate a new id for a request, and a new token;
        Deliver token to p_i ∈ TP_TRN until successful.
        Wait for p_i ∈ TP_TRN;
    }
    Return result;
}
```

Algorithm 3: Behavior of p_c.

Agreement

(A.1) No result is received by the p_c (client) unless the result was committed by all participants (database servers).

(A.2) No participant (database server) commits two different results.

(A.3) No two participants (database servers) decide differently on the same transaction.

Validity

(V.1) If the p_c (client) receives a result, then the result must have been computed by a node (application server) with, as a parameter, a request issued by the p_c (client).

(V.2) No participant (database server) commits a result unless all participants (database servers) have voted yes for that result.

4.1. Correctness Proof

Definition 1. Let TP_{TRN} be the set of participants that join the short-term transaction. This set is nonmodifiable. TP_{CMT} is a subset of TP_{TRN} that contains the participants that will be able to commit the transaction (i.e., reach state T_Pd) and TP_{ABT} is the subset of TP_{TRN} that *cannot* commit the transaction. $\text{TP}_{\text{TRN}} = \text{TP}_{\text{CMT}} \cup \text{TP}_{\text{ABT}}, \text{TP}_{\text{CMT}} \cap \text{TP}_{\text{ABT}} = \emptyset$. $|\text{TP}_{\text{TRN}}| = n, n \in N$, and $n > 0$. Token delivery failure can occur due to network partition, token omission, or process failure, all of which will eventually recover long enough for the tokens to be delivered.

Lemma 2. *If $TP_{CMT} = \emptyset$, then $TP_{TRN} = TP_{ABT}$. Even though token delivery between members of TP_{TRN} fails for a time T, TP_{TRN} will reach a uniform agreement eventually.*

Proof (Sketch). Because no members of TP_{ABT} can reach state T_Pd, they will either time out or receive a "T_A" message and then vote "T_A" themselves. Therefore P_R4 (i.e., for all $p_i \in \text{TP}_{\text{TRN}}$, getState($p_i$) $\subseteq$ Trans_Prepared, Trans_Commit, and Trans_Commit_ReadOnly) will never be satisfied, and no members will reach state T_C. According to our assumption, the delivery between members of TP_{TRN} will eventually be successful in transmitting T_A notifications to all members. Hence the claims follow. □

Lemma 3. *Let $0 < |TP_{CMT}| < n, |TP_{CMT}| + |TP_{ABT}| = n$. Even though the token delivery between members of TP_{TRN} fails for a time T, TP_{TRN} will reach a uniform agreement eventually.*

Proof (Sketch). The participants will vote "T_C" if and only if the preconditions P_R4 are satisfied, and only then is there a possibility for the transaction to commit. The premise assumes that some participants cannot commit the transaction, so these participants will never vote "T_Pd". Then P_R4 will never be satisfied, and therefore "S_TCd" will never be concluded. All we have to do is to prove that "S_TAd" will be concluded eventually, which means the members of TP_{CMT} will conclude "S_TA". Because P_R4 will never be satisfied, according to the transition graph shown in Figure 2, there are only two possible situations for the members of TP_{CMT}. The one is where some member of TP_{CMT} votes "T_A" because "S_Pd" cannot be concluded before its timer expires; the other situation is where some member of TP_{CMT} learns that some participants of TP_{ABT} voted "T_A" before its timer expires. After observing the above two situations, we can infer that all the members of TP_{TRN} will vote "T_A" eventually. According to the transition graph shown in Figure 2, the ones voting "T_A" will execute an abort procedure. Based on our assumption that the participants and network will be good long enough to contribute to the computation of a decision value, and according to the behavior of participants shown in Algorithm 2, all participants will keep trying to deliver tokens until "S_TAd" is concluded. All the members of TP_{TRN} will reach uniform agreement eventually. Hence the claim follows. □

Lemma 4. *Let $TP_{ABT} = \emptyset$, so then $TP_{TRN} = TP_{CMT}$. Even if the token delivery between members of TP_{TRN} fails for a time T, TP_{TRN} will reach a uniform agreement eventually.*

Proof (Sketch). There are two situations that need to be considered. One is where all members of TP_{TRN} can conclude "S_TPd" before their timer expires. The other is where at least

one participant cannot conclude "S_TPd" before its timer expires.

We discuss the former situation first. Recall that we assumed the participants and network environment would be good long enough to contribute to the computation of a decision value. From the definition of TS_R5 and transition graph shown in Figure 2, we can infer that all the participants will vote "T_C", disable their timer and then block until "S_TC" can be concluded. As soon as a participant concludes "S_TC", it will execute the commit procedure. Based on the premise that all members of TP_{TRN} can conclude "S_TPd" before their timer expires, none of the participants will vote "T_A" to abort the transaction due to the timer expiring. This means "S_TA" will never be concluded by any member in TP_{TRN}. According to the behavior of participants, the tokens will be transmitted until "S_TCd" is concluded. Therefore, all the participants can reach uniform agreement "S_TCd".

Now let us discuss situation 2. In this situation, at least one participant will vote "T_A" due to timer expiration, which may be caused by process failure or network failure. According to TS_R2, TS_R5 and the transition graph shown in Figure 2, we can conclude that if a participant voted "T_A" due to timer expiration, he will never vote "T_C" in the future. This means "S_TC" will never be concluded, which also implies that "S_TCd" will never be concluded. Based on the assumption of participants, network environment, and Lemma 3, all the participants will learn that someone voted "T_A", and then conclude "S_TA" and "S_TAd" eventually. Therefore, all the participants will reach a uniform agreement "S_TAd" eventually. Hence the claim follows. □

Lemma 5. *If p_c issues a request, then, it will receive a corresponding result from p_i eventually.*

Proof (Sketch). According to the behavior of p_c shown in Algorithm 3, p_c will not stop delivering a token to members of TP_{TRN} until the delivery is successful. As soon as the $p_i \in TP_{TRN}$ receives a token, it will save the token, and then forward the token and process the transaction. From Lemmas 2, 3, and 4, we can conclude the participants will reach a uniform agreement eventually. If the outcome delivered to p_c is "S_TAd", according to the behavior of p_c, p_c will generate a new transaction identifier for this request and issue a new request again, and this process will repeat until p_c receives an outcome with "S_TCd". After the token shows that all participants have reached a uniform agreement, the outcome will be delivered from p_i to p_c, but the outcome may be lost due to p_i or p_c crashing or network failure. According to the behavior of the participants, the token will keep circulating until the OD_F flag is turned on, which means that p_c received the outcome of the transaction. The OD_F will be turned on after "S_TCd" or "S_TAd" is concluded, which means that all the participants have reached a uniform agreement. Hence the claim follows. □

Termination: T.1. If p_c issues a request, then, unless it crashes, p_c eventually delivers a result.

Proof (Sketch). From Lemma 5, if p_c issues a request, p_c will receive a result eventually. The result has two cases: (1) the result is "*commit* (S_TCd)" and the client receives the result, hence the claim follows; (2) If the result is "*abort* (S_TAd)", then p_c will generate a new identifier for the transaction and restart the transaction again. This process will repeat a given number of times (as indicated by a user-defined system property) until the result sent to p_c carries "*commit* (S_TCd)", or p_c returns "*abort*" if each of the attempts results in S_TAd. Hence the claim follows. □

Termination: T.2. If any participants (database servers) vote for a result, then the participants (database servers) eventually commit or abort the result.

Proof (Sketch). According our assumption, every message received by the participant can be saved into stable storage successfully and can be retrieved from the stable storage successfully when recovering from failure. Regardless of whether the participants crash before or after voting for a result, any saved token which shows that uniform agreement has not yet been reached among participants will be retransmitted to other participants to continue the transaction.

Now consider a situation where there are no tokens being delivered between participants due to the token holder crashing or network partitions. The "rec_timer" of each participant, which is shown in Algorithm 2, can prevent transaction suspension in this situation. When the "rec_timer" expires, the timer owner will retransmit the token. Due to the monotonic nature of the state vectors, the duplicate tokens will not confuse the participants processing the transaction; therefore, the transaction will never be suspended. According to Lemmas 2, 3, and 4, all participants will reach a uniform agreement. □

Agreement: A.1. No result is delivered by the p_c (client) unless the result is committed by all participants (database servers).

Proof (Sketch). According to the behavior of participants, the token will be delivered to p_c only after all participants reach a uniform agreement, "S_TCd" or "S_TAd". That is, if p_c receives a token with result "S_TCd", then all participants must have reached a uniform agreement "S_TCd". On the other hand, if p_c has unsuccessfully tried the maximum number of retries and still goes to state "S_TAd", then all participants have agreed to abort. Hence the claim follows. □

Agreement: A.2. No participant (database server) commits two different results.

Proof (Sketch). For the participants to commit two different results, the client must send a request at least two times. But according to the behavior of p_c, the p_c resubmits a request only after it received a token with the result "S_TAd". This occurs only after all participants have reached a uniform agreement "S_TAd". As a consequence, each time a new request is issued by the client, no previous request could

still be committed. Therefore, no two different results can be committed by a database server. Hence the claim follows. □

Agreement: A.3. No two participants (database servers) decide differently on the same transaction.

Proof (Sketch). According to the proof of Lemmas 2, 3, and 4, no two participants (database servers) decide differently on the same transaction. □

Validity: V.1. If p_c (the client) receives a result, then the result must have been computed by participants (application servers) with, as a parameter, a request issued by p_c (the client).

Proof (Sketch). The p_c delivers the result after it receives a token with "S_TCd" indicated for all of the participants, for a given transaction identifier. Such a token is sent to the p_c if and only if the participants have reached a uniform agreement "S_TCd". A transaction associated with an identifier is committed only after the participants have computed their transaction actions and all participants have committed their changes. Given that the participants compute and forward the request only after the participants have received the request token with that identifier from the p_c, then it is not possible that the p_c receives the result unless it has issued a request that has been computed and decided on by all participants of the transaction. Hence the claim follows. □

Validity: V.2. No participant (database server) commits a result unless all participants (database servers) have voted yes for that result.

Proof (Sketch). According to the definition of P_R4, "S_TPd" cannot be concluded unless all participants are prepared (vote T_Pd) for the transaction. According to the definition of P_R5, "S_TCd" is concluded if and only if P_R4 has been concluded and all participants vote "T_C". Because the participants will disable their timer before voting "T_C", if all participants vote "T_C", it implies that all participants had disabled their timers, so there is only one possibility—that all participants will eventually commit the transaction. That is what Lemma 4 proved. Hence the claim follows. □

We now present a theoretical analysis of our algorithm by evaluating the response time, T_{resp}, from the time that a peer, say p_c, starts a transaction until the time p_c receives an outcome on the system model described in Section 2. There could be any ALM topology, but we will assume a linear topology for easy analysis of the worst case. Each participant must act as a router and a forwarder in ALM. For simplicity of presentation, but with no loss of generality, we assume the transaction here is a nice run, where no failures occur during the process, and every participant has the same computational capability. We ignore the execution time of the transaction actions which can be performed after the participants send the outcome message to p_c because the cost of these operations does not contribute to the response time perceived by p_c.

5. Analysis and Simulation Results

Under the same model, we compare our protocol with a traditional 3-phase commit protocol, where we let p_c be a coordinator. The connection topology is shown in Figure 3(a), where RTT_1 represents the round-trip delay between p_c and p_1 and RTT_i represents the round-trip delay between p_{i-1} and p_i for $2 \leqq i \leqq n$. Figures 3(b) and 4 show an abstraction of the transaction execution for a 3-phase commit protocol and our protocol, respectively. The summary of the total number of messages and response times for the 3-phase commit protocol and our protocol are listed below:

$$\begin{aligned}
&\text{3-PC response time} \\
&\quad = 6 \times \sum_{i=1}^{n} \text{RTT}_i + T^n_{\text{ok}} + T^n_{\text{prepared}} + T^n_{\text{commit}}, \\
&\text{3-PC msg.} = 3 \times \left(\frac{n^2 + 3n}{2}\right) \text{msg}, \\
&\text{Our ACP response time} = 4 \times \sum_{i=2}^{n} \text{RTT}_i + 2 \times \text{RTT}_i + T^n_{\text{ok}} \\
&\qquad + T^n_{\text{prepared}} + T^n_{\text{commit}}, \\
&\text{Our ACP number of msg.} = 4 \times (n-1).
\end{aligned} \tag{1}$$

We use T^n_{ok}, T^n_{prepared}, and T^n_{commit} to represent the latency of p_n when executing the tasks for "Ok to Prepare", "Prepared", and "Commit", respectively. Under the linear topology described above, p_n will be the last one to receive the message from p_c. Considering the 3-phase commit protocol in a distributed transaction, although each participant can execute a task and reply ack to the coordinator independently, the ack message from p_n to p_c will be the last message to arrive for all ack messages from all participants. Therefore, the response time of the transaction is approximately the time from when p_c delivers the first message to the time p_c receives the ack message that p_n has committed the transaction. The summation of RTT_i means the one way propagation time from p_c to p_n, the time multiply by 6 will produce the total propagation time for 3PC to finish the transaction. The packet produce by 3PC will be huge. In phase 1, in order to broadcast "Prepare" message to all member, the p_c send to p_1, then p_1 forward to p_2 and finally to p_n, it will produce n packet. But, without token accumulation approach, every node in 3PC must forward message for others. Therefore, p_n delivery message to p_c produce n packet, p_{n-1} will produce $n-1$ packets, the summation of these packet will be $n(n+1)/2$.

According to the behavior of participants in our protocol, each participant will forward a token to its neighbor immediately if it is the first token of a new, distinct transaction.

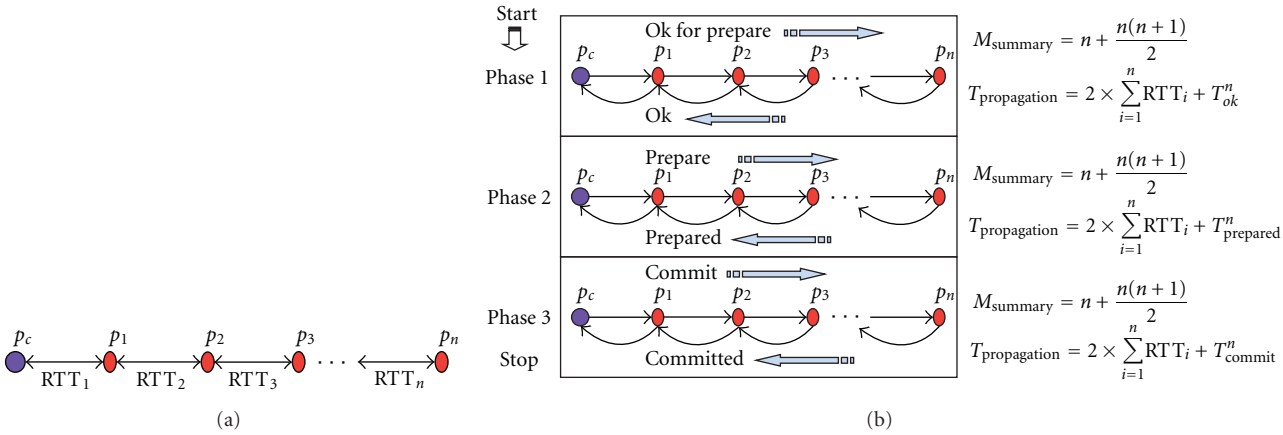

Figure 3: (a) Connection topology and (b) the abstraction of 3-phase commit protocol.

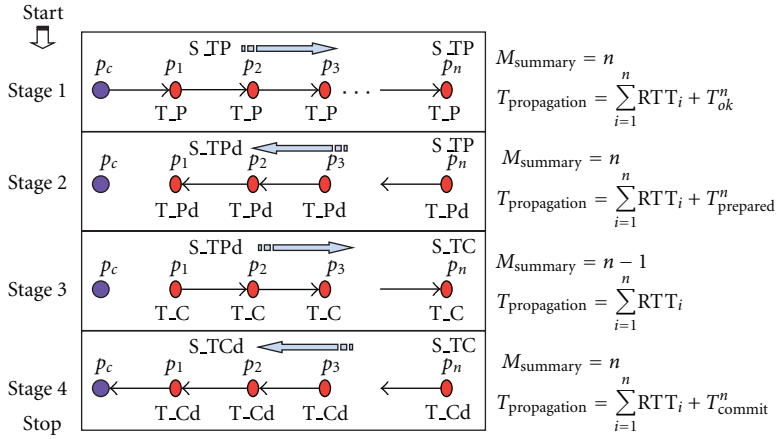

Figure 4: The abstraction of our protocol.

Therefore, in stage 1, each participant will forward a token and then perform a related task for "S_TP", and p_n will forward a token to itself and keep the token in the queue, and then process the related task of "S_TP". Stage 2 starts when p_n finishes performing the related task of "S_TP", and it pulls the token from the queue and merges it with T_{SAV}, and then forwards it to p_{n-1}. The rest of the process of stage 2, stage 3, and stage 4 are similar to that of stage 1.

We may observe that the response time of our protocol is shorter than the 3-phase commit protocol by about 2 times the propagation time from p_c to p_n. In addition, as the number of participants increases, the difference in the total number of messages and the propagation cost between the 3-phase commit protocol and our protocol will become much larger. That is because in an ALM network where each participant has to forward messages by way of its neighbor, our protocol can benefit from the "token accumulation" approach. This makes it unnecessary for each participant to send its vote to the coordinator. This theoretical analysis result is also supported by our simulation result which is shown in Figures 5 and 6.

If we let each participant communicate directly with p_c, then p_c may suffer from an overload of messages, similar to p_c suffering from a DDoS attack. Considering the total number of messages and response times, it should be clear that our peer-to-peer protocol is a significant improvement over a traditional 3-phase commit protocol, regardless of the overlay network structure.

Since our protocol will be the transaction management service for Cadabia, which is implemented in the Java programming language, we also implemented our protocol in Java. As we had mentioned in Section 2, we put more emphasis on making sure that our DECO transactions can satisfy the properties of e-Transactions. Our simulation program was developed based on the same assumptions as for the theoretical analysis.

We used JDK 6.X and Eclipse 3.4.X to develop our program. We used a compound data structure to keep all the related information for each distinct transaction, called the TDS (Transaction Data Structure). That is, each distinct transaction has exactly one TDS. The TDS contains a token for T_{SAV}, a string for the transaction identifier, and a queue to keep each received token, T_{REC}. Each participant uses a

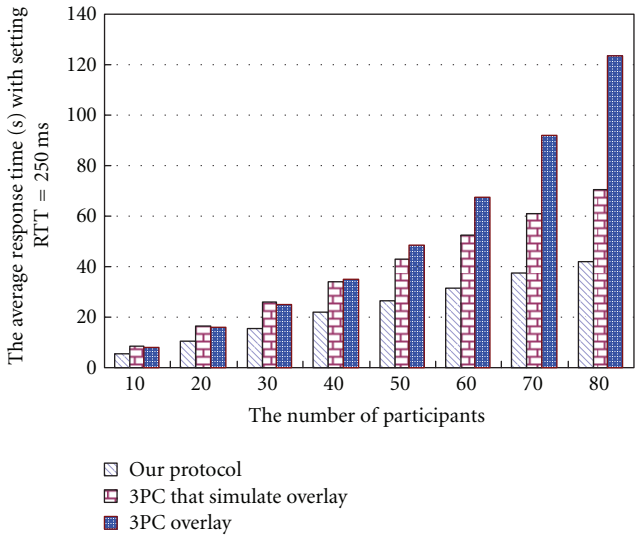

Figure 5: The average response time for each protocol.

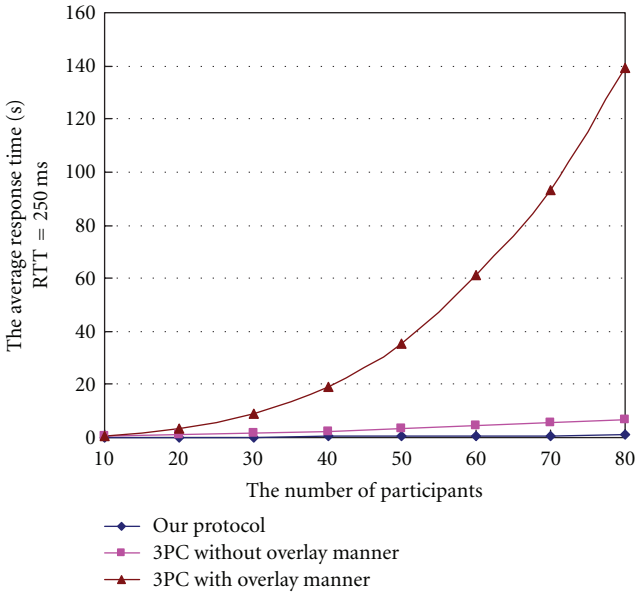

Figure 6: The average CPU time for each protocol.

hash table to keep distinct TDSs. As soon as a participant receives a token, it will use the transaction identifier as a hash key to determine whether a related TDS exists. If no related TDS exists, the participant will generate a new one, put the received token into the queue of TDS, save TDS to the hash table and start a new process to deal with the TDS.

Our simulation program had three versions. The first version implemented our protocol. The second and third versions implemented a traditional three-tier architecture, with a coordinator and 3-phase commit protocol. In order to figure out how serious the performance degradation becomes when a system suffers from huge numbers of messages, we used two different scenarios to simulate the behavior of participants when running the 3-phase commit protocol. In the first scenario, say $3PC_{SimulateOverlay}$, instead of letting each participant be responsible for forwarding messages, we let a participant, say p_n, (i.e., the one n nodes away from p_c) deliver ack messages directly to p_c after sleeping for n times a predefined round-trip delay time (RTDT, defined below). In the other scenario, say $3PC_{Overlay}$, we let each participant take the responsibility of forwarding messages for other participants to simulate the behavior of each host in a peer-to-peer overlay network. Because messages to the client (e.g., ack) are not merged, this model performs much worse than the others because p_c quickly becomes overloaded by messages.

The simulation starts from the main program delivery of a message to participants until the main program receives an outcome from participants. We reference the data in [5], and we assume the T^n_{ok}, $T^n_{prepared}$, and T^n_{commit} may take 10 msec. We only consider the case of no data contention, light system load, and a nice run (no participant crashes), because it helps us to evaluate the response time of each protocol more easily.

Figures 5 and 6 show the simulation results, which are the average response time and CPU time for each protocol after running 50 transactions. According to the investigation in [34], when all the participants are assumed to be spread on the Internet, the round-trip time will usually vary between 1 msec and 250 msec. We simulate the round-trip delay time (RTDT) of every delivery between participants by using a uniform random generator which produces a value from the lower bound of 1 msec to the upper bound of 250 msec.

We observe that, with an increasing number of participants, there are significant differences in response times between $3PC_{SimulateOverlay}$ and $3PC_{Overlay}$. We can observe a result that shows that the huge number of messages does seriously harm the system performance. On the other hand, the response time of our protocol increases linearly with the number of servers. During the simulation, the CPU utilization of $3PC_{Overlay}$ is much higher than $3PC_{SimulateOverlay}$. When the number of servers increases to 80, the CPU utilization of $3PC_{SimulateOverlay}$ remains below 10%, but the CPU utilization of $3PC_{Overlay}$ stays at 100% from the simulation start to the end. From Figure 6, the huge difference of CPU times between $3PC_{Overlay}$ and $3PC_{SimulateOverlay}$ can strongly support the above observation, where the CPU time of $3PC_{Overlay}$ grows very quickly; it also means the computational complexity of $3PC_{Overlay}$ is much worse than $3PC_{SimulateOverlay}$ and our protocol. We infer that the poor performance of $3PC_{Overlay}$ was caused by using a centralized approach in a P2P overlay network, and any ACP using coordinators will suffer from this syndrome. Our protocol performs very well in these simulations, not only having a lower response time but also having a much lower CPU time than $3PC_{SimulateOverlay}$ and $3PC_{Overlay}$. From these observations, we can see how a huge number of messages will degrade the 3PC system performance terribly. The simulations show that our protocol can be free from this drawback, and the worst-case response time of our protocol is directly proportional to the number of participants times the largest propagation cost among the participants. This shows the great potential of our protocol to be an ACP in a large-scale P2P application.

6. Conclusions

We have described and presented our decentralized and collaborative atomic commit protocol for peer-to-peer overlay networks. Our protocol can work without any specific coordinator or replication protocol, and with any appropriate decentralized multicast protocol. In addition, the protocol does not create an unbearable number of packets when there are many transactions which have many participants. Our future work is to implement the most appropriate multicast protocol which incorporates the characteristics of our decentralized ACP. We have some preliminary results. Finally, we want to improve our protocol to make it suitable for grid, cloud, and ad hoc computing as well as P2P computing.

References

[1] C. Turker, K. Haller, C. Schuler, and H.-J. Schek, "How can we support grid transactions? Towards peer-to-peer transaction processing," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR '05)*, Asilomar, Calif, USA, 2005.

[2] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[3] B. Awerbuch and C. Tutu, "Maintaining database consistency in peer to peer networks," Tech. Rep. CNDS-2002-2, 2002.

[4] S. Frølund and R. Guerraoui, "E-transactions: end-to-end reliability for three-tier architectures," *IEEE Transactions on Software Engineering*, vol. 28, no. 4, pp. 378–395, 2002.

[5] F. Quaglia and P. Romano, "Ensuring e-transaction with asynchronous and uncoordinated application server replicas," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 3, pp. 364–378, 2007.

[6] P. Romano, F. Quaglia, and B. Ciciani, "A lightweight and scalable e-Transaction protocol for three-tier systems with centralized back-end database," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 11, pp. 1578–1583, 2005.

[7] P. Romano and F. Quaglia, "Providing e-Transaction guarantees in asynchronous systems with inaccurate failure detection," in *Proceedings of the 5th IEEE International Symposium on Network Computing and Applications (NCA '06)*, pp. 155–162, July 2006.

[8] P. K. Chrysanthis, G. Samaras, and Y. J. Al-Houmaily, "Recovery and performance of atomic commit processing in distributed database systems," in *Recovery Mechanisms in Database Systems*, V. Kumar and M. Hsu, Eds., Prentice-Hall, 1998.

[9] S. Böttcher, L. Gruenwald, and S. Obermeier, "A failure tolerating atomic commit protocol for mobile environments," in *Proceedings of the 8th International Conference on Mobile Data Management (MDM '07)*, pp. 158–165, May 2007.

[10] K. P. Birman, *Reliable Distributed Systems Technology, Web Service and Application*, Springer, 2005.

[11] J. Liebeherr, M. Nahas, and W. Si, "Application-layer multicasting with Delaunay triangulation overlays," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 8, pp. 1472–1488, 2002.

[12] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz, "Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination," in *Proceedings of the 11th International Workshop on Network and Operating System Support for Digital Audio and Video*, June 2001.

[13] A. Sobeih, J. Wang, and W. Yurcik, "Performance evaluation and comparison of tree and ring application-layer multicast overlay networks," in *Proceedings of the IEEE International Computer Engineering Conference (ICENCO '04)*, 2004.

[14] A. Sobeih and W. Yurcik, "A survey of ring-building network protocols suitable for command and control group communications," in *Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defence IV*, vol. 5778 of *Proceedings of the SPIE*, pp. 873–884, April 2005.

[15] M. Castro, P. Druschel, A. M. Kermarrec, and A. I. T. Rowstron, "Scribe: a large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 8, pp. 1489–1499, 2002.

[16] A. J. Ganesh, A. M. Kermarrec, and L. Massoulié, "Peer-to-peer membership management for gossip-based protocols," *IEEE Transactions on Computers*, vol. 52, no. 2, pp. 139–149, 2003.

[17] M. Portmann and A. Seneviratne, "The cost of application-level broadcast in a fully decentralized peer-to-peer network," in *Proceedings of the 7th International Symposium on Computers and Communications (ISCC '02)*, 2002.

[18] S. Antony, D. Agrawal, and A. El Abbadi, "P2P systems with transactional semantics," in *Proceedings of the 11th International Conference on Extending Database Technology (EDBT '08)*, pp. 4–15, March 2008.

[19] J. Bose, S. Bottcher, L. Gruenwald, S. Obermeier, H. Schweppe, and T. Steenweg, "An integrated commit protocol for mobile network databases," in *Proceedings of the 9th International Database Engineering & Application Symposium (IDEAS '05)*, Montreal, Canada, 2005.

[20] K. Haller, H. Schuldt, and C. Türker, "Decentralized coordination of transactional processes in peer-to-peer environments," in *Proceedings of the 14th ACM International Conference on Information and Knowledge Management (CIKM '05)*, pp. 28–35, Bremen, Germany, November 2005.

[21] C.-Y. Wang and D. J. Buehrer, "A ring-based decentralized collaborative non-blocking atomic commit protocol," in *IEEE/WIC/ACM International Conference on Web Intelligence (WI '08) Held in Conjunction with International Conference on Intelligent Agent Technology (IAT '08)*, Sydney, Australia, December 2008.

[22] C. K. Yeo, B. S. Lee, and M. H. Er, "A survey of application level multicast techniques," *Computer Communications*, vol. 27, no. 15, pp. 1547–1568, 2004.

[23] D. J. Buehrer, L. O. Tse-Wen, and H. Chih-Ming, "Cadabia: a distributed, intelligent database architecture," in *Intelligent Multimedia, Computing, and Communications*, M. Syed and O. Baiocchi, Eds., pp. 96–101, John Wiley and Sons, New York, NY, USA, 2001.

[24] D. Buehrer and T.-Y. Wang, "The cadabia database project," in *Proceedings of the 14th Workshop on Object-Oriented Technology and Applications*, pp. 385–392, Ywan Jr. University, September 2003.

[25] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov, "The piazza peer data management system," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 7, pp. 787–798, 2004.

[26] A. Kementsietsidis, M. Arenas, and R. J. Miller, "Mapping data in peer-to-peer systems: semantics and algorithmic issues," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD '03)*, pp. 325–336, June 2003.

[27] K. Aberer, P. Cudré-Mauroux, A. Datta et al., "P-Grid: a self-organizing structured P2P system," *SIGMOD Record*, vol. 32, no. 3, pp. 29–33, 2003.

[28] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup service for internet applications," in *Proceedings of the ACM Applications, Technologies, Architectures, and Protocols for Computers Communications (SIGCOMM '01)*, pp. 149–160, August 2001.

[29] R. Guerraoui, HurfinM, A. Mostefaoui, R. Oliveira, M. Raynal, and A. Schiper, "Consensus in asynchronous distributed systems: a concise guided tour," in *Advances in Distributed Systems*, vol. 1752 of *Lecture Notes in Computer Science*, pp. 33–47, Springer, 2000.

[30] P. Bernstein and E. Newcomer, *Principles of Transaction Processing*, Morgan Kaufmann, 1997.

[31] D. A. Agarwal, L. E. Moser, P. M. Melliar-Smith, and R. K. Budhia, "The totem multiple-ring ordering and topology maintenance protocol," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 93–132, 1998.

[32] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella, "Totem single-ring ordering and membership protocol," *ACM Transactions on Computer Systems*, vol. 13, no. 4, pp. 311–342, 1995.

[33] A. Schiper, K. Birman, and P. Stephenson, "Lightweight causal and atomic group multicast," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 3, pp. 272–314, 1991.

[34] "Internet Traffic Report," 2009, http://www.internettrafficreport.com.