Two-Phase Commit Processing with Restructured Commit Tree

George Samaras[1⋆], George K. Kyrou[12⋆⋆], and Panos K. Chrysanthis[2⋆⋆]

[1] Department of Computer Science
University of Cyprus
Nicosia, Cyprus
{cssamara,kyrou}@cs.ucy.ac.cy

[2] Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, USA
panos@cs.pitt.edu

Abstract. Extensive research has been carried out in search for an efficient atomic commit protocol and many optimizations have been suggested to improve the basic two-phase commit protocol, either for the normal or failure case. Of these optimizations, the read-only optimization is the best known and most widely used, whereas the flattening-of-the-commit-tree optimization is the most recent one proposed for Internet transactions. In this paper, we study in depth the combined use of these two optimizations and show the limitations of the flattening-of-the-commit-tree method in committing large trees. Further, we propose a new restructuring method of the commit tree and show using simulation that it performs better than flattening method even when dealing with large trees.
Keywords: Atomic Commit Protocols, Commit Optimizations, Distributed Transaction Processing

1 Introduction

A transaction provides reliability guarantees for accessing shared data and is traditionally defined so as to provide the properties of *atomicity, consistency, isolation, and durability* (ACID) [8]. In order to ensure the atomicity of distributed transactions that access data stored at multiple sites, an *atomic commit protocol* (ACP) needs to be followed by all sites participating in a transaction's execution to agree on the transaction's final outcome despite of program, site and communication failures. That is, an ACP ensures that a distributed transaction is either *committed* and all its effects become persistent across all sites, or *aborted*

⋆ This work was supported in part by the European IST project DBGlobe, IST-2001-32645

⋆⋆ This work was supported in part by the National Science Foundation under grant IIS-9812532.

and all its effects are obliterated as if the transaction had never executed at any site. The *two-phase commit protocol* (2PC) is the first proposed and simplest ACP [7, 14].

It has been found that commit processing consumes a substantial amount of a transaction's execution time [22]. This is attributed to the following three factors:

- **Message complexity,** that is the number of messages that are needed to be exchanged among the sites participating in the execution of a transaction in order to reach a consistent decision regarding the final status of the transaction. This captures the cost of delays due to network traffic and congestion.
- **Log complexity**, that is the amount of information that needs to be recorded at each participant site in order to achieve resilience to failures. Log complexity considers the required number of *non-forced* log records, which are written into the log buffer in main memory, and the *forced* log records, which are written into the log on a disk (stable storage) that sustains system failures. Typically, however, log complexity is expressed only in terms of forced log writes because during forced log writes, the commit processing is suspended until the log record is guaranteed to be on stable storage. Thus, this captures the cost due to I/O delays.
- **Time complexity**, that is the number of *rounds* or *sequential exchanges* of messages that are required in order for a decision to reach the participants. This captures the cost due to propagation and network latencies.

Increasing any one of these factors, it will increase the delay in making the final decision to commit a transaction and consequently will increase the transaction response time. Further, any delay in making and propagating the final decision to commit a transaction, it delays the release of resources that decreases the level of concurrency and adversely affects the overall performance of a distributed transaction processing system. For these reasons, extensive research has been carried out over the years to reduce message, log and time complexities in an effort to optimize 2PC, improving both response time and system throughput during normal processing or during recovery from failures.

These research efforts resulted in a number of 2PC variants and new atomic commit protocols [16, 13, 24, 21, 10, 19, 15, 5, 1, 4] and an even greater number of commit optimizations for different environments [20, 6]. Of these optimizations, the *read-only* optimization is the best known and most widely used, whereas the *flattening-of-the-commit-tree* optimization is the most recent one originally proposed in [20] for *Internet transactions*, distributed across sites in a wide area network. Interestingly, the performance effects on transaction processing when combining some of these optimizations have not been studied in depth.

In this paper, we concentrate on the combined use of read-only and flattening-of-the-commit-tree optimizations. We show using simulation how the flattening-of-the-commit-tree optimization improves distributed commit processing by minimizing propagation latencies and allowing forced log writes to be performed in parallel. A major shortfall of the flattening method when dealing with large

trees is identified and investigated both analytically and quantitatively. This shortcoming is unnecessarily exacerbated when dealing with partially read-only transactions. During our attempt to combine flattening of the commit tree with the read-only optimization, we develop a new restructuring method which we call *restructuring-the-commit-tree around update participants* (RCT-UP). RCT-UP avoids the disadvantages of the flattening method while at the same time retains (and in some cases improves) the performance benefits of the flattening method and read-only optimization. Based on simulation results, we show that RCT-UP provides an overall superior performance.

The paper is organized as follows. In the next section, we provide a brief introduction to distributed transaction processing and in Section 3, we describe our assume system model and simulator which we use to support our analysis and evaluation of the commit optimizations examined in the remaining of the paper. In Section 4, we discuss the basic 2PC and the read-only optimization and in Section 5, we examine 2PC in the context of multi-level commit trees. In Section 6, we introduce and evaluate the flattening-the-commit-tree 2PC optimization, whereas we discuss its inefficiencies in Section 7. Also in Section 7 we propose and evaluate a new restructuring method that exploits read-only participants. In Section 8, we present our final conclusions.

2 Distributed Transaction Concepts

A distributed system consists of a set of computing nodes linked by a communications network. The nodes cooperate with each other in order to carry out a distributed computation or a network application. For the purpose of cooperation, the nodes communicate by exchanging messages via the network.

In a distributed environment, transactions are particularly useful to support consistent access to data resources such as databases or files [8], stored at different sites. A transaction consists of a set of operations that perform a particular logical task, generally making changes to shared data resources. These data access operations are executed within processes and can invoke additional data operations that may initiate other processes at the local site or different sites. Thus, while a transaction executes, processes may be dynamically created at remote sites (or even locally) in response to the data access requirements imposed by the transaction.

In general, a distributed transaction is associated with a tree of processes, called *execution tree* that is created as the transaction executes, capturing the parent-child relationship among the processes (Figure 1). At the root of the tree is the process at the site at which the transaction is initiated. For example, in the Client-Server model as Figure 1 shows, the root is the process at the client site. The execution tree may grow as new sites are accessed by the transaction and sub-trees may disappear as a result of the application logic or because of site or communication link failure.

When all the operations of a distributed transaction complete their execution, the process at the root of the tree becomes the *coordinator* for committing the

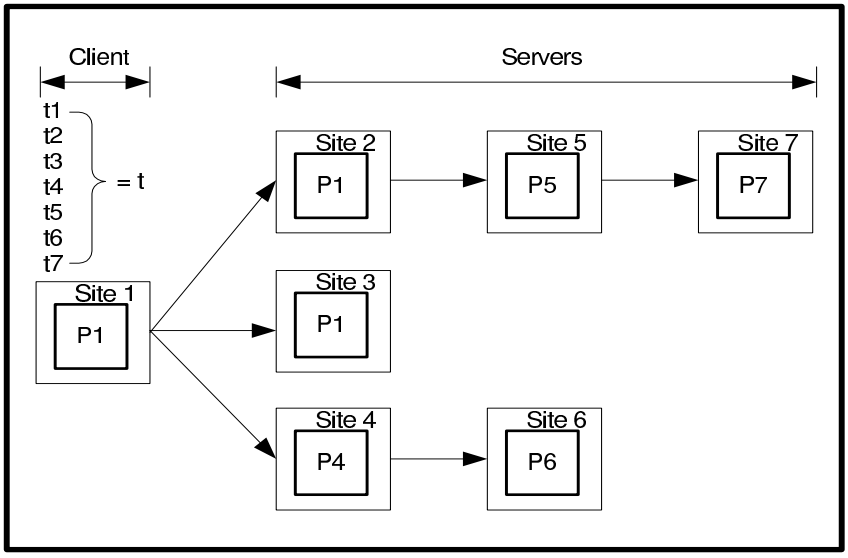

Fig. 1. A process tree within the Client-Server model

transaction and initiates an atomic commit protocol such as 2PC. Typically, the interactions between the coordinator of the transaction and any process have to go through all the intermediate processes that have caused the creation of a process. That is, a parent-child relationship in the execution tree implies a coordinator-subordinate relationship for executing the commit protocol. Consequently, the *commit protocol tree* is the same as the execution tree.

3 System and Simulation Model

In order to evaluate reliably the performance of the optimizations studied in this paper, we have developed a realistic simulator of a distributed transaction processing system. Our simulator is based on a synchronous discrete event simulation model and implemented in Object Pascal of Delphi 5 [11].

The simulator has been built to effectively capture not only distributed commit processing at the participating sites but also the communication processing as well. By modeling the details of the network infrastructure, our simulator is capable to simulate multi-level commit trees of any depth. In this respect, our simulator is different from the ones used in [18, 12] which are limited to flat, two-level, execution and commit trees, and to our knowledge, any other simulator of distributed commit processing.

The high-level components of the simulator, as in real systems, are the computing nodes, the communication links and routers (Figure 2).

Each computing node consists of a *transaction manager* (TM), a *log manager* (LM), a *communication manager* (CM) and a *storage manager* (SM). The trans-

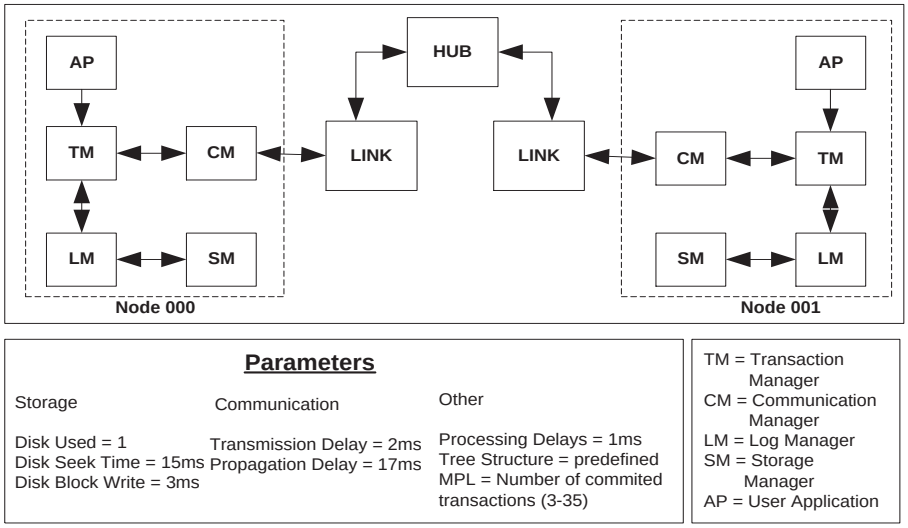

Fig. 2. Simulation Model

action manager is responsible for handling requests to initiate new transactions and requests to initiate and coordinate the commitment of transactions. The log manager processes any log request. For forced log writes that require disk access, the log manager invokes the storage manager to handle it. The disk read/write delay is achieved using a queue, while the blocking/unblocking of transaction processes through callback routines. Finally, the communication manager is responsible for queuing out-going messages and redirecting incoming messages to the transaction manager.

All of these components share a single processing unit (CPU) except from the storage manager which encapsulates the hard disk drive and controller and thus can function independently. The various commit protocols and optimizations are coded in special classes called protocol handlers[3] which are assigned to transaction processes that are executed by the transaction manager in a round-robin fashion. All components are built so that their full functionality is emulated as closely as possible. The parameters used, closely resemble those of a real world system (see Figure 2).

Simulating the network infrastructure precisely, was also a very important requirement so that we could capture not only the transmission and propagation delays of communication networks, but also the queuing delays when a link is highly utilized. All links support full-duplex communication, hence they implement two separate queues. Message routing is accomplished, with a rout-

[3] The protocol handler is probably the most complex class in our simulator. It is built around a state machine with states and event handlers implementing all the transaction commit states and actions.

ing device. For all experiments presented in this paper, we have used only one switching hub.

Also, in all the experiments discussed in the paper, we have not considered data processing that takes place before commit processing. Although this would have a significant effect on the overall system performance, we do not expect data processing to affect the relative performance between the various 2PC optimizations and in particular the ones under evaluation in this paper[4]

In all the experiments, we have measured the transaction throughput, which is the total number of committed transactions per unit time by varying the multiprogramming levels (MPLs). The MPL represents the total number of transactions executing at any given site and at any given point in time (since the system operates at full capacity). Specifically, commit processing is initiated when an application (AP)[5] at a node triggers a new transaction. In response the transaction manager spawns a new process and associates it with a protocol handler to execute the request. Based on a predetermined commit tree it also informs all the participating nodes that a new transaction has been initiated. When the process is given CPU time to execute, it will initiate the commit protocol by executing the proper actions of the protocol handler.

4 The Two-Phase Commit Protocol and Read-Only Transactions

In this section, we discuss the details of the *two-phase commit* protocol (2PC) in the context of two-level commit trees and its read-only optimization.

4.1 Basic Two-Phase Commit

The basic 2PC [7, 14] assumes a two-level execution tree where the root process spawns all the (leaf) processes. Consequently, it assumes a two-level, or depth 1, commit tree with the coordinator being the root and all the participants being at the leaf nodes.

As its name implies, 2PC involves two phases: a *voting* (or prepare) phase and a *decision* phase (Figure 3). During the voting phase, it is determined whether or not a transaction can be committed at all participating sites, whereas, during the decision phase, the coordinator decides either to commit or abort the transaction and all participants apply the decision. To provide fault-tolerance the protocol records its progress in logs at the coordinator and participating sites.

The Voting Phase: During the voting phase, the coordinator requests via the "prepare" message that all participants in a transaction agree to make the transaction's changes to data permanent. When a participant receives a "prepare"

[4] For one-phase commit protocols such as Early Prepare (EP) [23], Implicit-Yes Vote (IYV) [1] and Coordinator Log (CL) [24] that require part of commit processing to be performed during data processing, data processing is essential.

[5] The application is also responsible for maintaining the MPL level at each coordinator node.

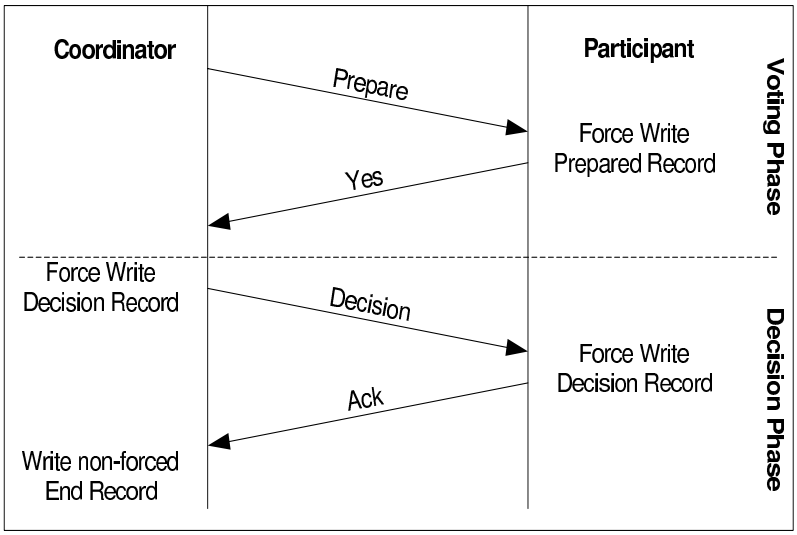

Fig. 3. Simple Two-Phase Commit Processing

message, it votes *Yes* if it is prepared to commit the transaction. A participant becomes *prepared to commit* a transaction after making sure that the transaction can be committed locally and force writing a Prepared log record that will enable it to resume the execution of the protocol after a site failure. If a participant has voted *Yes*, it can no longer unilaterally decide to abort or commit the transaction and it is considered blocked until it receives the final decision from the coordinator.

If a participant cannot prepare itself (that is, if it cannot guarantee that it can commit the transaction), it must abort the transaction and respond to the prepare request with the *No* vote.

The Decision Phase: When the coordinator has received a positive vote (i.e., *Yes* vote) from all of the participants, it decides to commit the transaction and force writes a Commit log record. Logging the commit decision ensures that even in case that the coordinator fails, when it becomes available after restart, the transaction can still commit successfully. If any of the participants has voted *No*, the coordinator decides to abort the transaction and force writes instead an Abort log record. After the coordinator has forced written a Commit (or Abort) log record, it sends a *"Commit"* (or *"Abort"*) message to all the voted *Yes* participants.

When a participant receives the final decision, it complies with the decision, force writes a Commit (or Abort) log record to indicate that the transaction is committed (or aborted), releases all resources held on behalf of the transaction, and returns an acknowledgment (namely the *"Ack"* message) to the coordinator. The coordinator discards any information that keeps in main memory regarding

the transaction when it receives acknowledgments from all the participants and forgets the transaction.

From the above it follows that, during normal processing the cost to commit or abort a transaction executing at ***n*** sites is the same. The log complexity of 2PC amounts to **2n-1** force log writes (one forced write at the coordinator's site and two at each of the *n-1* participants), and the message complexity to **3(n-1)** messages. The time complexity of 2PC is **3** rounds, the first one corresponds to the sending of the "prepare" requests, second one to the sending of votes, and the last one to the sending of the decision messages. We have not included in the message and time complexities the **n-1** acknowledgement messages and their corresponding round because these messages are used for bookkeeping purposes after the transaction has been committed or aborted.

4.2 Read-Only Optimization

Traditionally, a transaction is called *read-only* if all its operations are reads. On the other hand, a transaction is called *partially read-only* if some of its participants have executed writes as well. Otherwise, a transaction is called an *update* transaction.

Similarly, participants in the execution of a transaction can be distinguished into *read-only* participants, if they have performed only read operations on behalf of the transaction, and *update* participants, if they have performed at least one write operation on behalf of the transaction.

For a read-only participant in the execution of a transaction, it does not matter whether the transaction is finally committed or aborted since it has not modified any data at the participant. Hence, the coordinator of a read-only participant can exclude that participant from the second phase of commit processing, saving both in number of messages and forced log records. This is accomplished by having the read-only participant vote *Read-Only* when it receives the "prepare" message from its coordinator [16]. Then, without writing any log records, the participant releases all the resources held by the transaction.

In the case of a read-only transaction in which all participants are read-only, committing the transaction will require **0** forced-write log records and **2n-1** messages and have time complexity of **1**. Given that read-only transactions are the majority in any general database system, the read-only optimization can be considered one of the most significant optimizations. In fact, the performance gains allowed by the read-only optimization for both read-only and partially read-only transactions provided the argument in favor of the *Presumed Abort* protocol (PrA) (PrA is a variation of the 2PC [16]) to become the current choice of commit protocol in the OSI/TP standard [17] and other commercial systems.

5 Committing Multi-level Transactions

In the general case, distributed transactions can be associated with a multi-level execution tree of any depth greater than 1. At commit time, 2PC adopts the

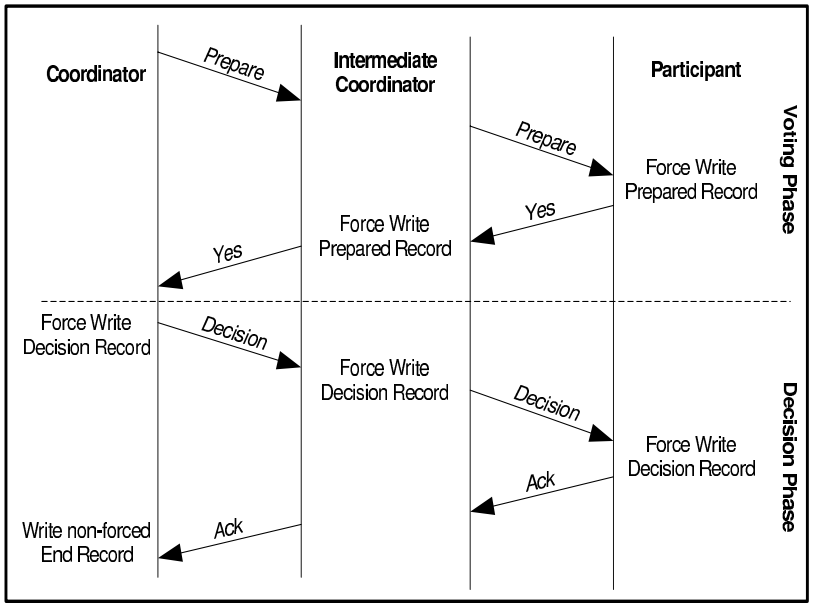

Fig. 4. Two-phase commit with intermediate/cascaded coordinator

same multi-level tree as the commit tree to propagate the coordination messages, making no assumption regarding any knowledge about the participating sites.

Specifically, each intermediate participant or *cascaded coordinator* propagates the various 2PC coordination messages down and up the transaction tree in a sequential manner, level by level. This is illustrated in Figure 4.

For example, consider the distributed transaction tree of depth 3, shown in Figure 5(a) in which all participants agree to commit the transaction. When the application finishes its processing, it issues commit to the Transaction Manager (TM) at site TP-0. TM-0[6] initiates the 2PC protocol and sends prepare to TM-1. Having subordinate transaction managers, TM-1 cascades (propagates) the "prepare" message to TM-2 and waits for its reply. Similarly, TM-2 cascades the "prepare" message to TM-3 and wait for its reply. After TM-3 receives the "prepare" message, it force writes a Prepared log record and responds with a *Yes* vote to its immediate cascaded coordinator, TM-2. When TM-2 (and similarly any subsequent cascaded coordinator) receives the vote from all its subordinates, it force writes a Prepared log record and replies to its immediate coordinator. Finally, when TM-0, the coordinator at the root receives the votes, it initiates the decision phase. The decision phase follows an analogous scenario.

This sequencing of the "prepare" and "decision" message implies that:

[6] From this point forward, we refer to the transaction manager of side TP-i as TM-i.

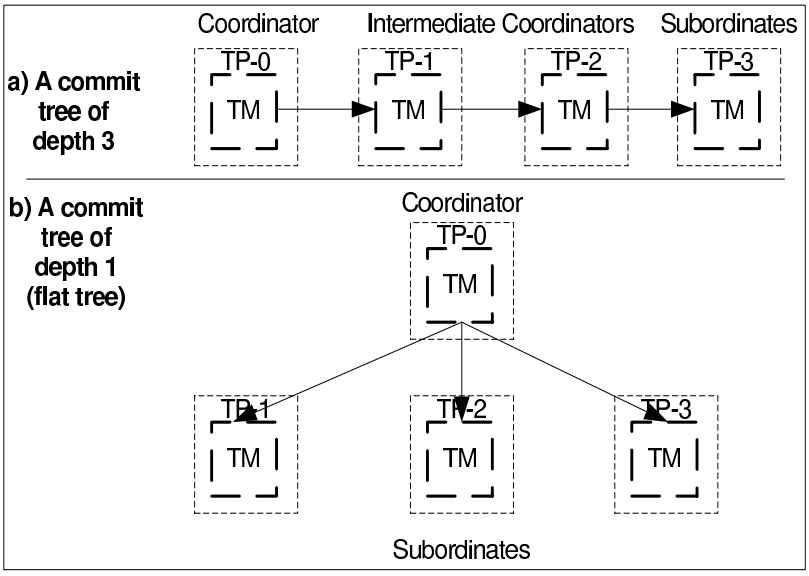

Fig. 5. A commit tree of depth 3 and its flattened counterpart

- A leaf participant will not receive the prepare message sent by the root coordinator until that message has been processed by all the intermediate, cascaded coordinators.
- An intermediate coordinator will not force write a Prepared log record until it has received the responses from all its subordinates.
- A leaf participant will not force write a commit log record until the commit message has been processed by all the intermediate coordinators.
- An intermediate coordinator can not acknowledge commitment to its coordinator until all acknowledgements from its subordinates are received, delaying the return to the application and the processing of another transaction.

For a transaction tree of depth three, as in our example, the vote message will arrive to the coordinator after three times the cost of receiving, processing and re-sending the prepare message and force writing a Prepared log record! The same applies for the "commit" and "Ack" message.

6 Flattening the Transaction Tree Optimization

It is clear that the serialization of the 2PC messages as well as of the forced log writes in multi-level commit trees increases the duration of commit processing as the tree depth grows. This serialization of messages/log writes is unavoidable in the general case in which the coordinator does not know all the participating sites in the execution of a transaction.

However, an alternative scheme that eliminates the sequencing of messages is feasible in communication protocols where a round trip is required before

commit processing. Remote Procedure Calls (RPC) or message-based protocols where each request must receive a reply, are examples of such communication protocols. With these protocols, during the execution of the transaction, the identity of the participating TMs can be returned to the (root) coordinator when the child replies to its parent. Gaining knowledge of the execution tree enables the coordinator to communicate with all the participants directly. As a result a multi-level execution tree is *flatten* or transformed into a 2-level commit tree during commit processing [20].

For example, by flattening the transaction execution tree of Figure 5(a), transforming it into the commit tree as shown in Figure 5(b), the root coordinator sends coordination messages directly to, and receives messages directly from, any participant. This eliminates the cascading coordinators and avoids the propagation delays (i.e., extra rounds of messages) due to cascading coordinators. By executing the prepare and decision phases in parallel across all participants, this optimization not only avoids the serial processing of messages, which significantly reduces time complexity, but also allows forced log writes to be performed in parallel, which significantly reduce the maximum duration of each phase. As illustrated below, the compound effect of the reduction in time complexity and phase duration is so significant on performance gains that makes this optimization as important as the read-only one for committing distributed transactions with deep execution trees.

6.1 Performance Analysis

Let M be the average time cost for transmitting a message, L the average time cost of force writing a log record and P the average time needed for processing a particular event. For the commit tree in Figure 5(a), the minimum time to successfully commit a transaction is approximately $(3*3M+2*3L+L)P$. In general, for a balanced commit tree of depth D the minimum time required for processing commitment is $(3*DM+2*DL+L)P$; the formula is solely based on the fact that for a balanced commit tree, commit processing (i.e., sending the various messages and force writing different log records) is done at each level in parallel (see Figure 6). The multipliers $3*$ accounts for the number of messages and $2*$ for the number of forced log writes required by the basic 2PC protocol. This clearly shows that the cost is dominated by the depth and not as much by the number of participants.

For the flattened commit tree of Figure 5(b) (and any commit tree of depth 1) the time cost is only $(3*M+2*L+L)P$. This is so because the cost of sending the various messages downstream is eliminated and the various force writes are now done in parallel across all the participants - transmitting the messages (namely the "prepare" and "commit" messages) in parallel involves only minor transmission delay[7]. Based on this, the formula is transformed into $(3M+3L)P+2(N-1)Td$, where Td is the transmission delay and N the total number of participants. This is a considerable improvement. In general, the performance of a flattened

[7] As we will see later this is not the case for large trees.

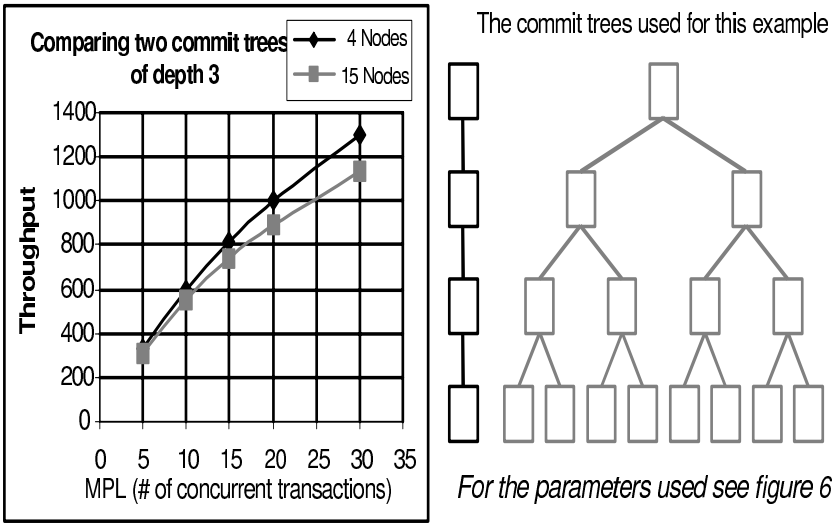

Fig. 6. Demonstrating the depth domination over the number of nodes when measuring the performance of 2PC processing

tree is (theoretically) almost D times better than the performance of its counterpart of depth D.

Our simulation experiments produced similar results. Figure 7 shows the simulation results for the example trees in Figure 5. The deviation from our estimate is most probably attributed to the transmission delay. The upper bound throughput value ($\sim$1200), on which both lines converge is the maximum system throughput. At that point, the communication-link of the coordinator is 100% utilized. Similarly, an upper bound system throughput can be imposed by the storage subsystem. This is merely dependent on the performance characteristics of the hard drives and of course, the maximum number of force writes executed by an individual system.

6.2 System and Communication Requirements

A limitation of the flattening-the-commit-tree optimization is that in some distributed systems, security policies of the participating nodes may not permit direct communication with the coordinator. Protocols that support security features that prohibit any-to-any connectivity cannot use this optimization without additional protocols to handle the case where a partner cannot directly connect with the commit coordinator.

Another limitation is that a reply message is required so that the identity of all the partners is known to the coordinator before the voting phase of the 2PC protocol. Protocols that do not require replies, such as conversational protocols (for example, IBM's LU6.2 [9]), may not know the identities of all participants. These protocols save time by not requiring a reply to every request. For those

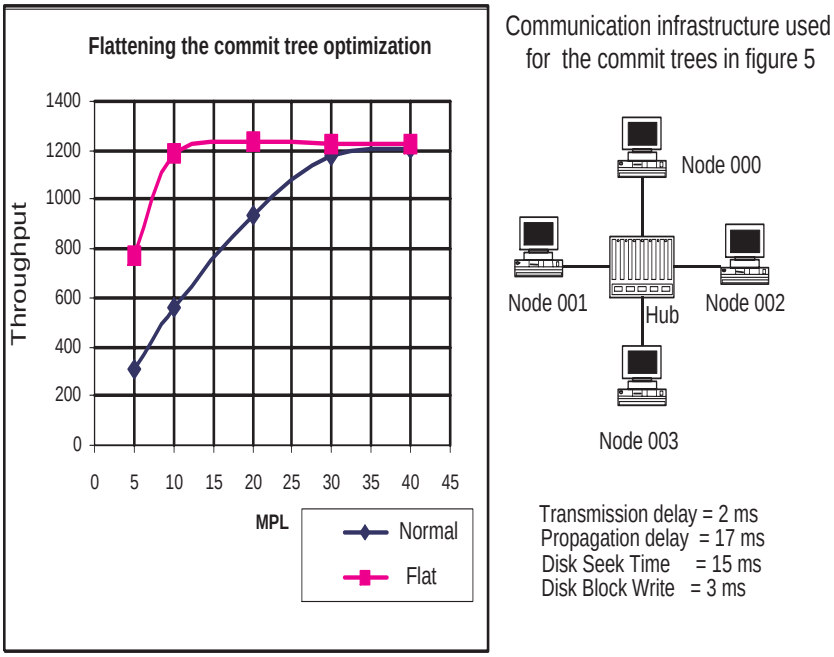

Fig. 7. Simulation results for the normal and flat commit trees of figure 5

protocols, it is possible to flatten the tree during the decision phase, by returning the identity of each subordinate to the coordinator during the reply to the "prepare" message, that is, as part of the vote message.

7 Restructuring, a New Approach

Flattening the tree can shorten the commit processing significantly at the cost of requiring extra processing on behalf of the coordinator. This is not much of a problem when transactions involve only a small number of participants or when the network infrastructure supports multicasting. However, when this is not the case, as with the TCP/IP communication protocol, sending commit messages to a great number of participants might lead to the following drawbacks:

- Because of transmission delays, a participant may actually receive a "prepare" or "decision" message later than what would normally have received (i.e., without flattening).
- Communication from and to the coordinator is overloaded, effectively reducing the overall system throughput.

To make these observations more profound consider the flattened commit tree of Figure 8(b). For this tree, during the prepare phase the coordinator (P0) needs to exchange with its subordinates eight messages. Consequently, the

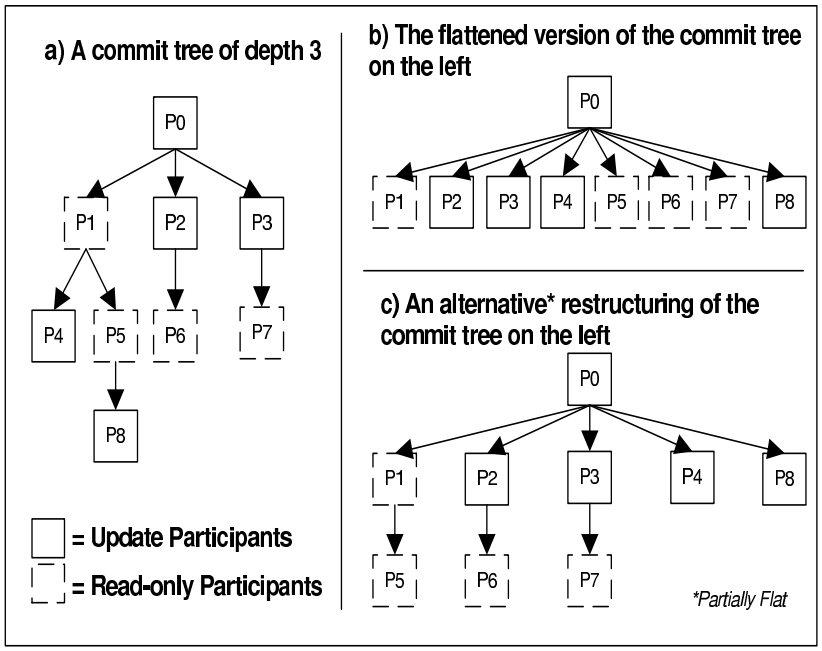

Fig. 8. Flattening and an alternative Restructuring based on read-only status

last subordinate receives the "prepare" message with an additional delay equal to at least seven times the transmission delay of the "prepare" message. *Note that the computer system hosting the coordinating process P0 is connected to the communication network through a single line* (Figure 7). In order to send eight messages, the hardware (i.e., the network card) requires eight times the cost (transmission delay) of sending one message. If the networking protocol used requires acknowledgment of message reception (round-trip protocols), the cost becomes even greater.

Regardless of the networking protocol used, for large commit trees, consisting of large number of participants or for high volume transaction-systems, the cost of exchanging a huge number of messages can decrease the performance of 2PC dramatically. In fact, our simulation results confirmed that for large trees, certain non-flat structures might perform better than the flat structure! For example, for a one-level tree with sixteen[8] subordinates and a transmission delay of *3ms* the two-level, non-flat commit tree performs better than its flattened counterpart (Figure 9). The performance degradation of the flattened tree is attributed to the transmission delay, which is exacerbated by the communication burden placed

[8] We used a large transmission delay to show the inefficiency of the flat tree for a small number of participants to ease the simulation. For a smaller transmission delay ($<$ 0.5ms) which is the normal for Ethernet networks, a larger number of participants would be needed for the performance of the flattened tree to significantly degrade.

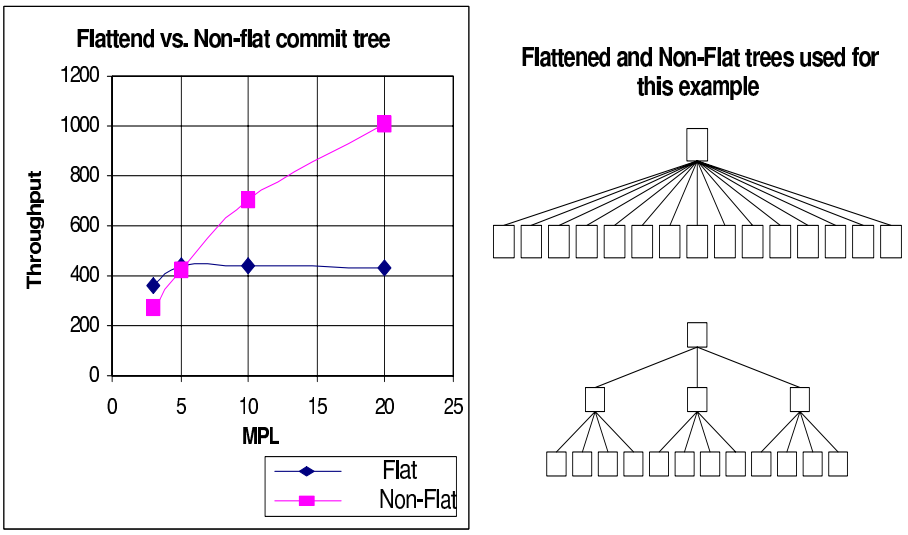

Fig. 9. Demonstrating the performance collapsing of the flattened commit tree

on the coordinator. When running more than five concurrent transactions (MPL 5) the coordinator's communication link is 100% utilized. This case, however, represents a non-typical scenario. It is worth noting that the number is not chosen at random but is derived with the aid of the previous formulas[9].

7.1 Restructuring Around Update Participants (RCT-UP)

The above drawback can be avoided by taking advantage of read-only participants that can be eliminated from the second phase of commit processing. Instead of flattening the commit tree completely and having the coordinator to send prepare to *all* participants, we can restructure[10] the tree in such a way so that the coordinator needs to directly send messages to the update participants that actually modified data and have to force write log records.

To accomplish this, *only* participants that modify data notify the coordinator of their identity. This can be done in a manner similar as in the *unsolicited update-vote* optimization where when a participant executes the first update operation on behalf of a transaction, it notifies the transaction's coordinator [3, 4]. They also notify their intermediate coordinator (if they have one) so that during the voting phase the intermediate coordinator can exclude these participants from its subordinates. This has the notion of removing update nodes and even sub-trees from the bottom of the commit tree and connecting them directly to

[9] The number of intermediate participants must be greater than P*(2M+L)(D-1)/Td.

[10] From this point forward, we use the term *restructuring* to refer to the partial flattening of the commit tree. The authors of this paper consider flattening a specific form of restructuring.

the root (i.e., the coordinator). A useful observation is that at the end of the data processing, in the transformed commit tree, all nodes that have depth greater than one are read-only. Figure 8(c) (and Figure 10(3)) illustrates this algorithm. For this tree, only processes P4 and P8 need to be restructured.

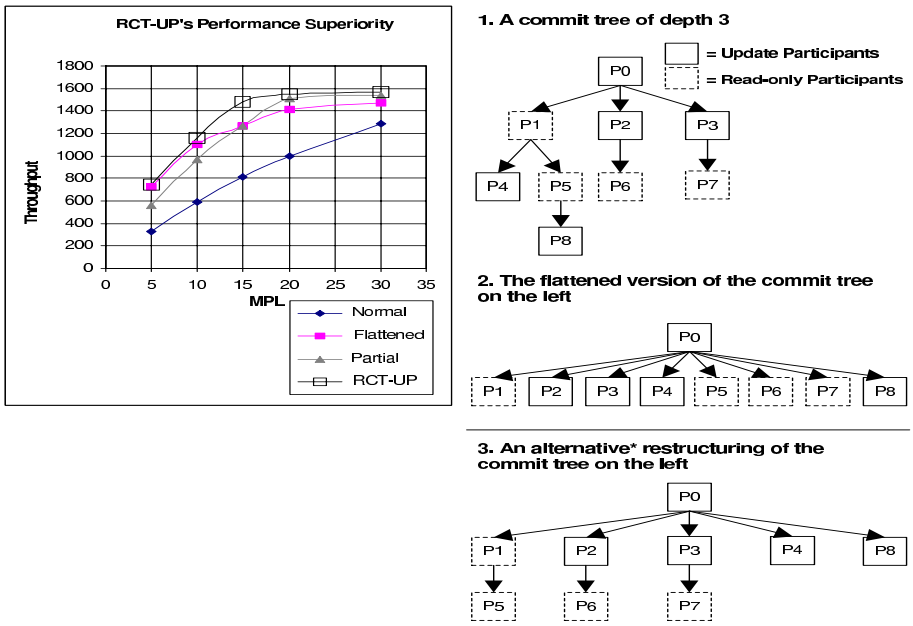

Fig. 10. Simulation results for the commit trees of figure 7 (reproduced for clarity)

Clearly, our proposed new *restructuring of the commit tree around the update participants* (RCT-UP) technique is better than the flattening optimization as it relieves the communication from and to the coordinator, enhancing its multi-programming efficiency (i.e., the ability to run many transactions concurrently). In addition, since read-only participants can be excluded from the second phase of 2PC processing, not flattening them does not affect the performance of 2PC processing. Recall that each immediate subordinate (of the coordinator) in this restructuring method *"knows"* that all its subordinates are read-only. Therefore, all immediate subordinates can force write a Prepared log record and respond to the coordinator *before* sending any messages to their subordinates. Of course, this is a divergence from the standard 2PC, but that is the case with almost all optimizations. On the other hand, we can just send them a *read-only* message [3] and forget about them.

Figure 10 demonstrates the performance effects of the RCT-UP optimization. The least performing structure is the normal, non-flat tree (Figure 10(1)). An improvement in performance is demonstrated by the restructured tree (Figure 10(3)), so that all update participants communicate directly with the coordina-

tor. The interesting observation though is the performance of the flattened tree (Figure 10(2)). Although initially it has a very high throughput, after MPL 15 it proves inadequate compared to the partially flat structure. This, as we have already explained, is attributed to the transmission delay. In both cases (flattened and partially flattened[11]), no prior knowledge[12] of read-only participants has been used. However, when we take advantage of the existence of read-only participants, the performance gains in the case of RCT-UP are significant. from this, the advantage of the RCT-UP optimization is evident.

8 Conclusions

The *flattening-of-the-commit-tree* optimization has been proposed to enhance the performance of distributed transactions in wide area networks but without any systematic analysis of its performance. In this paper, we presented such a detailed evaluation both analytically and quantitatively using simulation. We demonstrated how it improves distributed commit processing by minimizing propagation delays and by allowing log writes to be performed in parallel. A major shortfall of flattening when dealing with large transaction trees has been also identified. It was shown that this deficiency, attributed to the transmission delay and message congestion, is exacerbated when dealing with partially read-only transactions.

To effectively remedy this problem we proposed a new restructuring method, which we call *restructuring-the-commit-tree-around-update-participants* (RCT-UP), that avoids the disadvantages of flattening while at the same time improving upon its advantages. Based on simulation results, we showed that RCT-UP, which in essence is a combination of the flattening-the-commit-tree and the read-only optimization provides an overall superior performance.

References

1. Al-Houmaily, Y. J. and P. K. Chrysanthis. An Atomic Commit Protocol for Gigabit-Networked Distributed Database Systems. *The Journal of Systems Architecture*, 46(9): 809-833, 2000.
2. Al-Houmaily Y. J. and P. K. Chrysanthis. Atomicity with Incompatible Presumptions. *8th ACM Symp. on Principles of Database Systems*, pp. 306–315, 1999.
3. Al-Houmaily, Y. J., P. K. Chrysanthis, and S. P. Levitan. Enhancing the Performance of Presumed Commit Protocol. *12th ACM Annual Symp. on Applied Computing*, pp. 131-133, 1997.
4. Al-Houmaily, Y. J., P. K. Chrysanthis, and S. P. Levitan, An Argument in Favor of Presumed Commit Protocol. *13th Int'l Conf. on Data Engineering*, pp. 255–265, 1997.

[11] We call partially flattened the RCT-UP optimization with no prior knowledge of read-only participants.

[12] No knowledge of the read-only participants means that we can only utilize the traditional read-only optimization.

5. Al-Houmaily, Y. J. and P. K. Chrysanthis. Dealing with Incompatible Presumptions of Commit Protocols in Multidatabase Systems. *11th ACM Annual Symp. on Applied Computing*, pp. 186–195, 1996.
6. Chrysanthis, P. K., G. Samaras and Y. J. Al-Houmaily. Recovery and Performance of Atomic Commit Protocols in Distributed Database Systems. *Recovery in Database Management Systems*, V. Kumar and M. Hsu, eds., pp. 370–416, Prentice Hall, 1998.
7. Gray, J. N. Notes on Data Base Operating Systems, In *Operating Systems - An Advanced Course*, R. Bayer, R. Graham, and G. Seegmuller (Eds.), LNCS, Vol. 60 Springer-Verlag, 1978.
8. Gray, J. N., and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufman, 1993.
9. *IBM Database System DB2*, Version 3, Document Number SC30-3084-5, IBM, 1994.
10. *IBM Systems Network Architecture. SYNC Point Services Architecture Reference*, Document Number SC31-8134, 1994.
11. Kyrou, G., and G. Samaras. A Graphical Simulator of Distributed Commit Protocols. Internet Site: http://ada.cs.ucy.ac.cy/~cssamara/
12. Liu M. , D. Agrawal and A. El Abbadi. The Performance of Two-Phase Commit Protocols in the Presence of Site Failures. *24th Int'l Symp. on Fault-Tolerant Computing*, 1994.
13. Lampson B. and D. Lomet, A New Presumed Commit Optimization for Two Phase Commit. *19th VLDB Conf.*, pp. 630-640, 1993.
14. Lampson, B.W. Atomic Transactions, in *Distributed Systems: Architecture and Implementation - An Advanced Course*, B.W. Lampson (Ed.), LNCS, Vol. 105, Springer-Verlag, pp. 246-265, 1981.
15. Mohan, C., K. Britton, A. Citron, and G. Samaras. Generalized Presumed Abort: Marrying Presumed Abort and SNA's LU6.2 Commit Protocols. *Workshop on Advance Transaction Models and Architectures* (ATMA96), 1996.
16. Mohan, C., B. Lindsay, and R. Obermarck. Transaction Management in the R* Distributed Data Base Management System. *ACM Transactions on Database Systems*, 11(4):378–396, 1986.
17. Information Technology - Open Systems Interconnection - Distributed Transaction Processing - Part 1: OSI TP Model; Part 2: OSI TP Service, ISO/IEC JTC 1/SC 21 N, 1992.
18. Gupta R., J. Haritsa and K. Ramamritham. Revisiting Commit Processing in Distributed Database Systems. *1997 ACM SIGMOD Conf.*, pp. 486-497, 1997.
19. Raz Y. The Dynamic Two-Phase Commitment Protocol. *5th Int'l Conf. on Information Systems and Data Management*, 1995.
20. Samaras, G., K. Britton, A. Citron, and C. Mohan, Commit Processing Optimizations in the Commercial Distributed Environment, *Distributed and Parallel Databases Journal*, 3(4):325-361, 1995.
21. Samaras, G., K. Britton, A. Citron, and C. Mohan. Enhancing SNA's LU6.2 Sync Point to Include Presumed Abort Protocol, *IBM Technical Report TR29.1751*, IBM Research Triangle Park, 1993.
22. Spiro, P., A. Joshi, T. K. Rengarajan. Designing an Optimized Transaction Commit Protocol. *Digital Technical Journal*, Vol. 3, No. 1, Winter 1991.
23. Stamos J. W. and F. Cristian. A Low-Cost Atomic Commit Protocol. *9th Symp. on Reliable Distributed Systems*, pp. 66-75, 1990.
24. Stamos J. W. and F. Cristian. Coordinator Log Transaction Execution Protocol. *Distributed and Parallel Databases Journal*, 1(4):383-408, 1993.