Structuring the Commit Tree for Better Performance of Two Phase Commit Processing

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

Department of Computer Science, University of Cyprus, 1678 Nicosia, Cyprus
Department of Computer Science, University of Pittsburgh, Pittsburgh, 15260 PA, USA
email: {cssamara,kyrou}@cs.ucy.ac.cy, 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 two-phase commit protocol, either for the normal or failure case. Yet, 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 flattening-of-the-commit-tree optimization, in particular the combination of flattening with the read-only optimization. Our simulation results reveal a major pitfall of flattening when dealing with large trees. A new restructuring method is proposed that performs better than flattening even when dealing with large trees. Required protocol modifications to support the suggested optimizations are also addressed.

Keywords: Atomic Commit Protocols, Commit Optimizations, Distributed Transaction Processing

1 Introduction

A transaction provides reliability guarantees for shared access to data and is traditionally defined so as to provide the properties of *atomicity, consistency, isolation, and durability* (ACID) for any operation it performs. 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 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 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 [1, 12].

It has been found that commit processing consumes a substantial amount of a transaction's execution time [20]. This is attributed to the following three factors:

- **Message complexity,** which deals with the number of messages that are needed to be exchanged between the systems participating in the execution of a transaction to reach a consistent decision regarding the final status of the transaction.
- **Log complexity**, which refers to the amount of information that needs to be recorded at each participant site in order to achieve resiliency to failures. Typically, log complexity is expressed in terms of 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 the disk. During forced log writes, the commit operation is suspended until the log record is guaranteed to be in stable storage.
- **Time complexity**, which corresponds to the number of rounds or sequential exchanges of messages that are required in order for a decision to reach the participants.

Since any delay in making and propagating the final decision to commit a transaction, decreases the level of concurrency and adversely affects the performance of a distributed transaction processing system, it is important to reduce it. In fact, over the years the 2PC protocol has been extensively optimized to improve performance and system throughput in terms of reliability, savings in log writes and net-

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

work traffic, and reduction in resource lock time. This work resulted in a number of commit variations [14, 11, 22, 19, 8, 17, 13, 6, 3, 5] and an even greater number of commit optimizations [18]. Yet, 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 flattening-of-the-commit-tree optimization, originally proposed in [18]. We show via simulation how it improves distributed commit processing by minimizing processing delays and allowing log writes to be performed in parallel. A major shortfall of flattening when dealing with large trees is also being investigated both analytically and quantitatively. This shortcoming is unnecessarily exacerbated when dealing with partially read-only transactions. In an attempt to combine flattening with the read-only optimization, we invent a new restructuring method which we call restructuring-the-commit-tree around update participants (RCT-UP). RCT-UP avoids the disadvantages of flattening while at the same time sustains (and in some cases improves) the performance benefits of flattening. Based on simulation results, we show that restructuring around update participants, which in essence is a combination of the flattening-the-commit-tree and the read-only optimization, provides an overall superior performance.

The paper is organized as follows. In Section 2, we present a brief description of distributed transaction processing, commit processing and commit optimizations. The flattening-the-commit-tree optimization is discussed and evaluated in Section 3. Section 4 deals with the inefficiencies of flattening and investigates a new restructuring method exploiting read-only participants. In Section 5, presents an overview of the Simulator used to support our analyses/evaluations. Section 6 concludes.

2 Distributed Transaction Concepts

The following three sections describe the system model, the basic two-phase commit protocol and the needed commit optimizations that are used throughout this paper.

2.1 Distributed Computations

A distributed system consists of a set of computing nodes linked by a communications network. The nodes cooperate with each other in order to process distributed computations. For the purpose of cooperation, the nodes communicate by exchanging messages via the network. A transaction consists of a set of operations that are executed to perform a particular logical task, generally making changes to data resources such as databases or files [2]. The changes to these resources must be committed or aborted before the next transaction in the series can be initiated.

A distributed transaction is associated with a tree of processes that is created as the transaction executes (Fig. 2-1). Processes may be created at remote sites (or even locally) in response to the data access requirements imposed by the transaction. Consequently, there exists a parent-child relationship between the processes. The tree may grow as new sites are accessed by the transactions. Sub-trees may disappear in response to application logic or because of a site or communication link failure.

The client process at the root is the overall initiator of the commit protocol. *Consequently, the commit protocol tree is the same as the process tree so that the parent-child relationship implies the coordinator-subordinate relationship for executing the commit protocol.*

Once the computations of a transaction are completed, the application instructs the transaction manager (TM) of its site (i.e., P1) to initiate and coordinate the commit protocol. The coordinator TM must arrive at a commit or abort decision and propagate that decision to all subordinates. Subordinate TMs propagate the decision to their subordinate TMs.

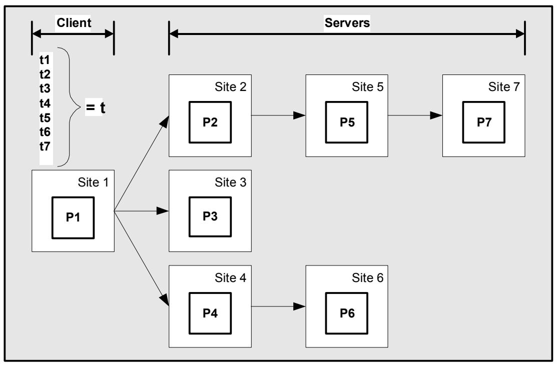


Figure 2-1. A process tree within the client server model.

2.2 The Two-Phase Commit Protocol

The **two-phase commit protocol** (2PC) [1, 12], as the name implies, involves two phases: a voting (or prepare) phase and a decision (or commit) phase (Figure 2-2). During the voting phase, it is decided whether to commit or abort the transaction and at the decision phase, all participants apply the decision. To provide fault-tolerance the protocol uses a log to record its progress.

The Voting Phase

During the voting phase, the coordinator requires via the "prepare" message that all participants in a transaction agree to make the transaction's changes to data permanent. When a participant in a transaction has completed the voting phase "positively", it is considered prepared. Being prepared means that the participant has received the prepare message, has processed the request, has forced-logged a prepare log record and has returned a message (namely the "Yes" message) to the coordinator stating that it has completed the prepare phase. Once a participant in a transaction is prepared, it can no longer unilaterally decide to abort or commit the transaction and it is considered blocked. 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.

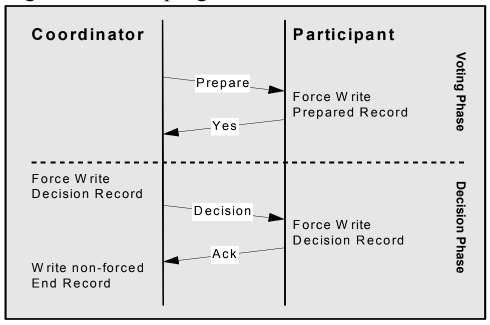

Coordinator
Participant
Voting Phase
Prepare
Force Write Prepared Record
Yes
Force Write Decision Record
Decision
Force Write Decision Record
Decision Phase
Ack
Write non-forced End Record

Figure 2-2. **Simple Two-Phase Commit Processing.**

The Decision Phase

When it has received a positive vote (i.e. the "Yes" vote) from each of the participants, the coordinator force writes a commit log record. Logging the commit decision ensures that even in case of failures and when all the participating systems are again available, the transaction can complete successfully. If any of the participants voted "No" the coordinator force-writes instead the abort log record.

After the coordinator has forced written a commit (or abort) log record, is ready to initiate the decision phase. The coordinator sends a *"Commit"* (or *"Abort"*) message to all the voting "Yes" participants. Each participant 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. Once a commit log record has been written, the changes made by the transaction can be applied to the database.

From the above it follows that, during normal processing the cost to commit or abort a transaction executing at ***n*** participants 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 participant), and the message complexity to **4(n-1)** messages. The time complexity of 2PC is **4** rounds, first is the sending of the "prepare" requests, second is the sending of votes, third is the sending of the decision message and last is the sending of the acknowledgement message.

2.3 Read-Only Optimization

Traditionally, a transaction is called *read-only* if all operations it has performed on behalf of a transaction were 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.

If a participant in a transaction is read-only, it does not matter whether the transaction is finally committed or aborted since it has not modified any data. Hence, the coordinator of a read-only participant can exclude that participant from the second phase of commit processing. This is accomplished by having the read-only participant vote "*Read-Only*" when it receives the "prepare" message from its coordinator [14]. Then, without writing any log records, the participant releases all the resources held by the transaction.

The read-only optimization [14] can be considered one of the most significant optimizations, given that read-only transactions are the majority in any general database system. In fact, the performance gains allowed by the read-only optimization provided the argument in favor of Presumed Abort (PrA) (PrA is a well known variation of the 2PC [14]) to become the current choice of commit protocol in the OSI/TP standard [15] and other commercial systems.

3 Flattening-the-Transaction-Tree Optimization

The typical 2PC protocol treats the distributed transaction as a multi-level tree[2]. Each intermediate participant propagates the various 2PC coordination messages down and up the transaction tree in a sequential manner, level by level. This is illustrated in Figure 2-3 and in Figure 3-1 (a). The cascading of the protocol means that the transaction manager (TM) at site TP-1 (in figure 3-1 (a)) must receive and process the "prepare" message before TM-2[3] can be sent the cascaded prepare from TM-1. *This serialization of the 2PC messages increases the duration of 2PC processing as the tree depth grows.*

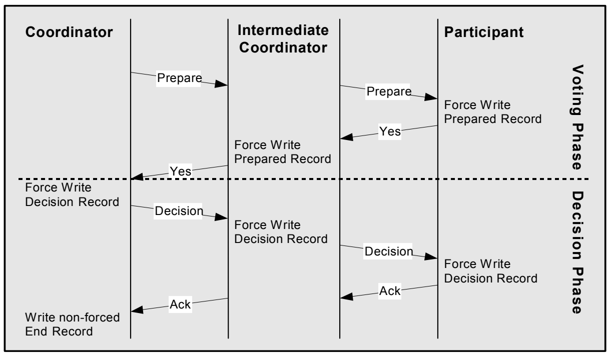

Coordinator
Intermediate Coordinator
Participant
Prepare
Prepare
Force Write Prepared Record
Yes
Force Write Prepared Record
Yes
Force Write Decision Record
Decision
Force Write Decision Record
Decision
Force Write Decision Record
Ack
Ack
Write non-forced End Record
Voting Phase
Decision Phase

Figure 2-3. Two-phase commit with intermediate coordinator.

An alternative to this 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, the identity of the cascaded subordinate TMs can be returned to the coordinator when the child replies to its parent. Gaining knowledge of the commit-tree enables the coordinator to communicate with all the participants directly. As a result a multi-level tree is flatten or transformed into a 2-level one during commit processing [18].

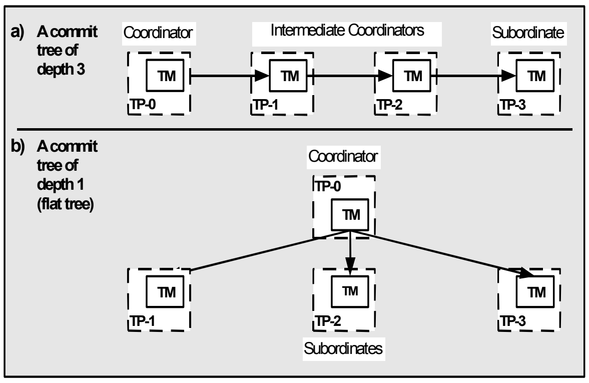

a) A commit tree of depth 3
Coordinator
Intermediate Coordinators
Subordinate
TM
TP-0
TP-1
TP-2
TP-3
b) A commit tree of depth 1 (flat tree)
Coordinator
TP-0
TM
TP-1
TP-2
TP-3
Subordinates

Figure 3-1. A commit tree of depth 3 and its flattened counterpart.

For example, consider the distributed transaction tree of depth 3, shown in Figure 3-1 (a). When the application finishes its processing, it issues commit to the Transaction Manager at site TP-0. TM-0 initiates the 2PC protocol and sends prepare to TM-1. Having subordinate transaction managers, TM-1 propagates prepare to TM-2. Similarly, TM-2 sends prepare to TM-3. After TM-3 receives the prepare-to-commit message it force writes a prepared log record and responds with a "Yes" vote (or "No" if it wants to abort) to its immediate coordinator, TM-2. When TM-2 (and every intermediate coordinator) receives the prepare-vote from all its subordinates, it force writes a prepared log record and replies to its immediate coordinator accordingly. The decision phase follows an analogous scenario.

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

- A leaf participant will not receive the prepare message sent by the root coordinator until that message has been processed by all the intermediate 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

[2]In the client/server model, this is the same as the commit tree

[3] From this point forward, we refer to the Transaction Manager of side TP-i as TM-i.

processing of another transaction.

For a transaction tree of depth three, as in our example, the prepare-to-commit vote 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.

By flattening the transaction tree, as shown in Figure 3-1 (b), we eliminate the sequencing of messages. In this way, the root coordinator sends coordination messages directly to, and receives messages directly from, any participant. This avoids the propagation delays due to cascading of managers. By executing the prepare and decision phases in parallel across all participants, this optimization not only avoids the serial processing of messages, but also allows forced log writes to be performed in parallel. In fact, time and log complexities are greatly reduced making this optimization a big performance winner in distributed transactions that contain deep trees.

3.1 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 3-1 (a), the minimum time to successfully commit a transaction is approximately $(4*3M+2*3L+L)P$. In general, for a balanced commit tree of depth D the minimum time required for processing commitment is $(4DM+2DL+L)P$; 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 3-2). The multiplier 4 accounts for the number of messages and 2 for the number of forced log writes required by the basic 2PC protocol. This shows that

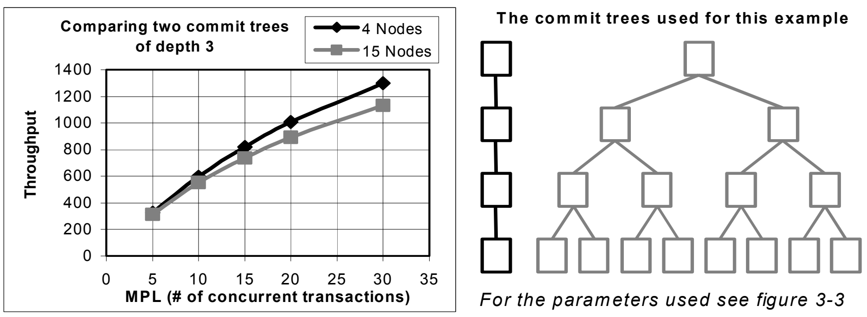


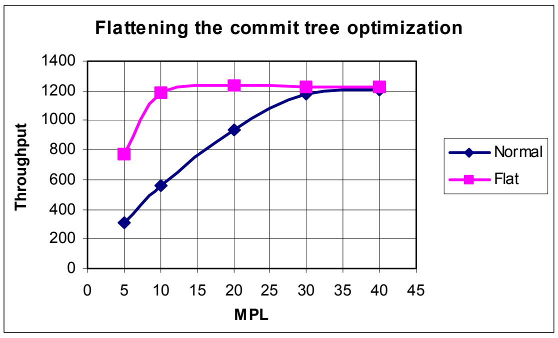
Figure 3-2. Demonstrating the depth domination over the number of nodes when measuring the performance of 2PC processing**.**

the cost is dominated by the depth and not as much by the number of participants.

For the flattened commit tree of figure 3-1 (b) (and any commit tree of depth 1) the time cost is only $(4M+2L+L)P$. 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[4]. Based on this, the formula is transformed into $(4M+3L)P+2(N-1)Td$, 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 tree is (theoretically) almost D times better than the performance of its counterpart of depth D.

Our simulator indicated similar results. Figure 3-3 shows the simulation results for the example trees in figure 3-1. The deviation from our estimate is most probably attributed to the transmission delay. The boundary throughput value (~1200), on which both lines converge is the maximum system throughput. At that point, the communication-link of the coordinator is 100% utilized. Similarly, a boundary system throughput can be invoked 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 exe-

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



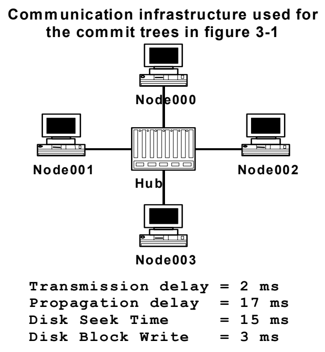

Communication infrastructure used for the commit trees in figure 3-1
Node000
Node001
Node002
Node003
Hub
Transmission delay = 2 ms
Propagation delay = 17 ms
Disk Seek Time = 15 ms
Disk Block Write = 3 ms

Figure 3-3. Simulation results for the normal and flat commit trees of figure 3-1.

cuted by a single system.

3.2 Acquiring Knowledge of the Commit Tree

Acquiring knowledge of the commit tree (and flattened[5] it) is achieved by requiring all participants to send their identity to the root coordinator when acknowledging the first operation. Thus, when commit processing is initiated, the coordinator can send coordination messages directly to all participants. In addition, assuming all participants know that this optimization is being used, they do not propagate any 2PC messages to any subordinates that they may have.

3.3 System and Communication Requirements

A limitation of this 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 [7]), may not know the identities of all agents. These protocols save time by not requiring a reply to every request. For those 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.

4 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 accommodate only a few participants and 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 have the following drawbacks:

- Because of the transmission delay, a participant may actually receive a "prepare" or "decision" message latter than would normally do (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 4-1 (b). For this tree, during the prepare phase the coordinator (P0) needs to exchange with its subordinates eight messages. Consequently, the last subordinate to receive the "prepare" message will do so after eight times the transmission delay. *Note that the computer system hosting the coordinating process P0 is connected to the communication network through a single line (figure 3-3)*. 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 is even greater.

Regardless of the networking protocol used, for large commit trees, consisting of hundreds of par-

[5] Note that we can only minimize the depth of the commit tree not the depth of the execution tree.

ticipants or for high volume transaction-systems, the cost of exchanging a huge number of messages can decrease the performance of two-phase commit processing dramatically. In fact, simulation results showed that for large trees, certain non-flat structures might perform better than the flat structure! For example, for a one-level tree with sixteen[6] subordinates and a transmission delay of *3ms* the two-level non-flat commit tree performs better than its flattened counterpart (Figure 4-2). The performance collapsing of the flattened tree is attributed to the transmission delay, which is exacerbated by the communication burden placed 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[7].

4.1 Restructuring Around Update Participants (RCT-UP)

To avoid the previous inefficiency we can take 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 send prepare to **all** participants, we only need to restructure[8] and directly send messages to the ones that actually modified data.

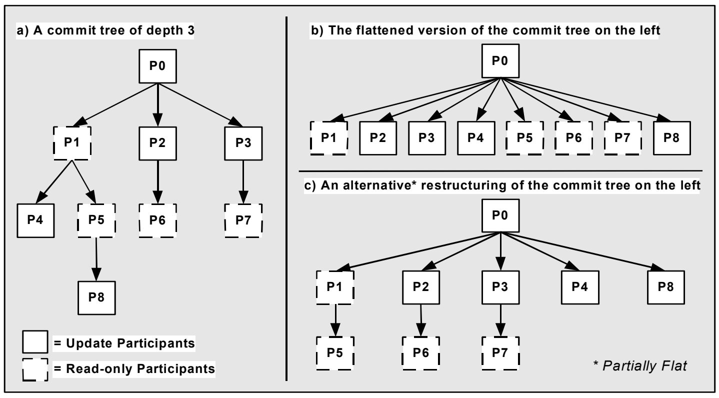

a) A commit tree of depth 3
P0
P1
P2
P3
P4
P5
P6
P7
P8
= Update Participants
= Read-only Participants
b) The flattened version of the commit tree on the left
c) An alternative* restructuring of the commit tree on the left
* Partially Flat

Figure 4-1. Flattening and an alternative Restructuring based on read-only status.

To accomplish this, **only** participants that modify data notify the coordinator of their identity. They also notify their intermediate coordinator (if they have one) so that during the prepare-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 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 4-1 (c) (and 4-3 (3)) illustrates this algorithm. For this tree, only processes P4 and P8 need to be restructured.

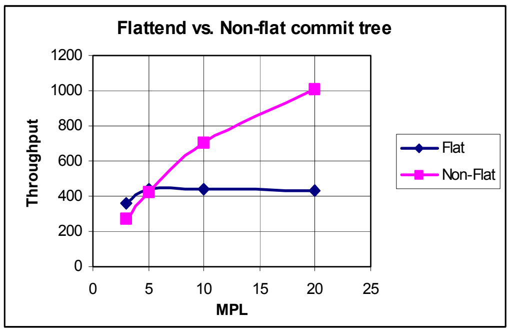
Clearly, this new restructuring technique is better than the standard flattening optimization as it relieves the communication from and to the coordinator, enhancing its multiprogramming efficiency (i.e., the ability to run many transactions concurrently). In addition, since read-only participants can be excluded from the second phase of 2PC processing, not flattening them does not affect the performance of the two phase commit 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 prepare record and respond to the coordinator **before** sending any messages to their subordinates· Of course, this is a divergence from the standard 2PC protocol, but that is the case with almost all optimizations! On the other hand, we can just send them a *read-only* message [4] and be done with them.

Figure 4-3 demonstrates the performance effects of the RTC-UP optimization. The least performing structure is the normal, non-flat tree (figure 4-3 (1)). An improvement in performance is demonstrated

[6] 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 collapse.

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

[8] 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.



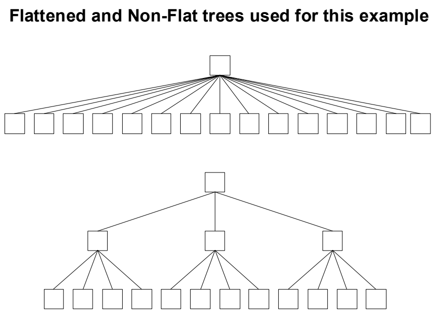
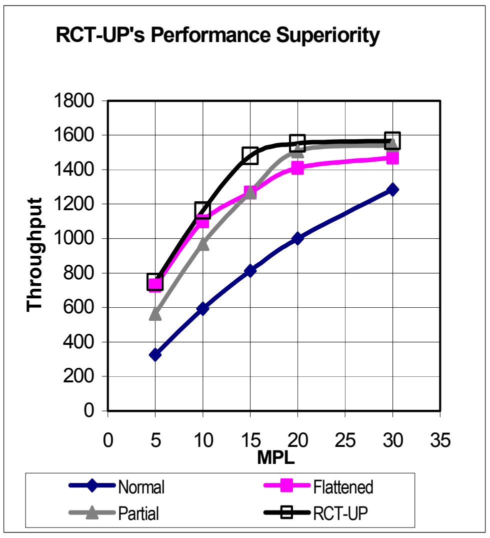


Figure 4-2. Demonstrating the performance collapsing of the flattened commit tree.

by the restructured tree (figure 4-3 (3)), so that all update participants communicate directly with the coordinator. The interesting observation though is the performance of the flattened tree (figure 4-3 (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[9]), no prior knowledge[10] of read-only participants has been used. However, when we take advantage of the existence of read-only participants (RCT-UP), the per-



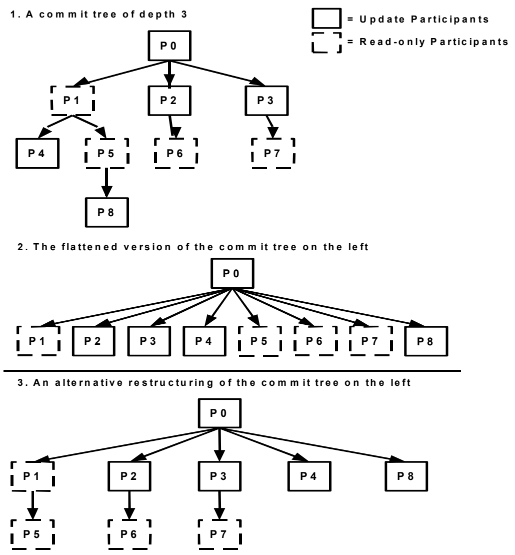

1. A commit tree of depth 3
= Update Participants
= Read-only Participants
P0
P1
P2
P3
P4
P5
P6
P7
P8
2. The flattened version of the commit tree on the left
P0
P1
P2
P3
P4
P5
P6
P7
P8
3. An alternative restructuring of the commit tree on the left
P0
P1
P2
P3
P4
P8
P5
P6
P7

Figure 4-3. Simulation results for the commit trees of figure 4-1 (reproduced for clarity).

formance gains are significant. The advantage of the RCT-UP optimization is evident.

5 Simulation Model

In order to evaluate reliably the performance of the optimizations studied in this paper (Flattening and RCT-UP), we have used a realistic simulator of a distributed transaction processing system. Our simulator, based on a synchronous discrete event simulation model, has been built to effectively capture not only distributed commit execution but also the communication processing as well. In this respect, by modeling the network infrastructure, our simulator is different from the one used in [16, 10] and to our knowledge, any other simulator of distributed commit processing. This approach has enabled us to

[9]We call partially flattened the RCT-UP optimization with no prior knowledge of read-only participants.
[10]No knowledge of the read-only participants means that we can only utilize the traditional read-only optimization.

simulate commit processing with intermediate coordinators, and thus efficiently compare the flat structure with other, deeper commit trees.

5.1 Simulator Components

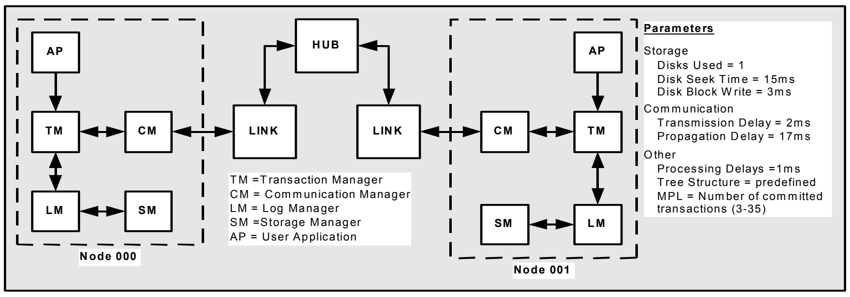


Figure 5-1. Simulation Model.

The high-level components of the simulator, as in real systems, are the computing nodes and the communication links and routers (figure 5-1).

Each computing node consists of a transaction manager (TM), a log manager (LM), a communication manager (CM) and a storage manager (SM). The transaction manager is responsible for handling requests to initiate new transactions and requests to initiate and coordinate the commitment of a transaction. The log manager executes request for forced log writes by invoking the storage manager to handle disk access. 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[11] 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 real-life functionality is emulated as closely as possible. The parameters used, closely resemble those of a real world system (see figure 5-1) [9].

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 routing device. For all experiments mentioned in this paper, we have only used a switching Hub.

5.2 Commit processing

Since this paper concentrates on commit optimizations over a specific commit variant (namely the basic two-phase commit) we have not utilized the data manager and thus we do not simulate (at this stage) the data exchange that takes place before commit processing. Although this will have a significant effect on the overall system performance, we do not expect it to affect the relative performance between the various optimizations.[12]

Commit processing is initiated when an application[13] 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.

[11]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.

[12]For protocols such as Early Prepare (EP) [21], Implicit Yes Vote (IYV) [3] and Coordinator Log (CL) [22] that require part of commit processing to be performed during data processing the data manager is essential.

[13]The application (AP) is also responsible for maintaining the MPL level at each coordinator node.

6 Conclusions

Flattening has been originally proposed in [18] but without any systematic analysis of its performance benefits or pitfalls. In this paper, we presented a detailed evaluation of the flattening-the-commit-tree optimization not only analytically but quantitatively as well with the aid of a simulation tool developed in house [9]. We demonstrated how it improves distributed commit processing by minimizing propagation delays and allowing log writes to be performed in parallel. A major shortfall of flattening when dealing with large transaction trees has been also discovered. 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 invent 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 show 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. Gray J.N.: “Notes on Data Base Operating Systems”, in “*Operating Systems - an Advanced Course”*, by Bayer R., Graham R. and Seegmuller G. (eds.), Springer-Verlag, LNCS Vol.60, 1978. Also Available as *IBM Research Report RJ2188*, IBM Almaden Research Center, Feb. 1978.
2. Gray J. and Reuter A.: “*Transaction Processing: Concepts and Techniques”,* Morgan Kaufman, 1993.
3. Al-Houmaily Y.J. and Chrysanthis P.K.: "An Atomic Commit Protocol for Gigabit-Networked Distributed Database Systems". *The Journal of Systems Achitecture*, 1997.
4. Al-Houmaily Y.J., Chrysanthis P.K. and Levitan S.P.: "Enhancing the Performance of Presumed Commit Protocol", *Proc. 12th ACM SAC Symposium*, pp.131-133, Feb. 1997.
5. Al-Houmaily Y.J., Chrysanthis P.K. and Levitan S.P.: "An Argument in Favor of Presumed Commit Protocol", *Proc. 13th IEEE International Data Engineering Conference*, Birmingham, U.K., April 1997.
6. Al-Houmaily Y.J. and Chrysanthis P.K.: "Dealing with Incompatible Presumptions of Commit Protocols in Multidatabase Systems." *Proc. 11th ACM SAC Symposium*, pp.186-195, Feb. 1996.
7. IBM Database System DB2, Version 3, Document Number SC30-3084-5, IBM, June 1994
8. Systems Network Architecture. SYNC Point Services Architecture Reference, Document Number SC31-8134, IBM, September 1994. It introduces and describes in detail IBM’s Presumed Nothing commit protocol. Authors: George Samaras, Kathryn Britton, Andrew Citron.
9. Kyrou G. and Samaras G.: "A Graphical Simulator of Distributed Commit Protocols" Available from: http://ada.cs.ucy.ac.cy/~cssamara/
10. Liu M., Agrawal D. and El Abbadi A.: "The Performance of Two-Phase Commit Protocols in the Presence of Site Failures", *Proc. 24th Intl. Symposium on Fault-Tolerant Computing*, June 1994.
11. Lampson B. and Lomet D.: "A New Presumed Commit Optimization for Two Phase Commit", *Proc. 19th VLDB Conference*, Dublin, Ireland, 1993.
12. Lampson B.: “Atomic Transactions”, in *“Distributed Systems: Architecture and Implementation - an Advanced Course”*, by Lampson B. (ed.), Springer-Verlag, LNCS Vol.105, pp.246-265, 1981.
13. Mohan C., Britton K., Citron A. and Samaras G.: "Generalized Presumed Abort: Marrying Presumed Abort and SNA's LU6.2 Commit Protocols", *An International Workshop on Advance Transaction Models and Architectures (ATMA’96)* (edited book), Goa, India, Sept. 1996.
14. Mohan C., Lindsay B. and Obermarck R.: “Transaction Management in the R* Distributed Data Base Management System”, *ACM Transactions on Database Systems*, Vol.11, No.4, Dec. 1986. Also available as IBM Research Report RJ5037, IBM Almaden Research Center, Feb. 1986.
15. 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, April 1992.
16. Gupta R., Haritsa J. and Ramamritham K.: "Revisiting Commit Processing in Distributed Database Systems". *Proc. ACM SIGMOD Conference*, Tucson, Arizona, May 1997.
17. Raz Y.: "The Dynamic Two-Phase Commitment (D2PC) Protocol", *Proc. 5th International Conference on Information Systems and Data Management (CISMOD'95),* Bombay, India, Nov. 1995.
18. Samaras G., Britton K., Citron A. and Mohan C.: “Commit Processing Optimizations in the Commercial Distributed Environment”, *Parallel and Distributed Databases,* Vol.3, No.4, pp.325-361, Oct. 1995.
19. Samaras G., Britton K., Citron A. and Mohan C.” “Enhancing SNA's LU6.2 Sync Point to Include Presumed Abort Protocol”, IBM Technical Report TR29.1751, IBM Research Triangle Park, Aug. 1993.
20. Spiro P., Joshi A. and Rengarajan T.K.: “Designing an Optimized Transaction Commit Protocol”, *Digital Technical Journal*, Vol.3, No.1, Winter 1991.
21. Stamos J.W. and Cristian F.: "A Low-Cost Atomic Commit Protocol", *Proc. 9th Symposium on Reliable Distributed Systems*, pp. 66-75, 1990.
22. Stamos J.W. and Cristian F.: "Coordinator Log Transaction Execution Protocol", *Distributed and Parallel Databases*, 1:383-408, 1993.