The Circular Two-Phase Commit Protocol

Heine Kolltveit and Svein-Olaf Hvasshovd

Department of Computer and Information Science
Norwegian University of Science and Technology

Abstract. Distributed transactional systems require an atomic commitment protocol to preserve atomicity of the ACID properties. However, the industry leading standard, 2PC, is slow and adds a significant overhead to transaction processing. In this paper, a new atomic commitment protocol for main-memory primary-backup systems, C2PC, is proposed. It exploits replication to avoid disk-logging and performs the commit processing in a circular fashion. The analysis shows that C2PC has the same delay as 1PC, and reduces the total overhead compared to 2PC.

1 Introduction

Main memory prices have dropped significantly over the last years, and the state of many applications and databases can now be fitted entirely in main memory. To make the state both persistent and available, it can be replicated instead of written to disk. For instance, a backup replica (backup for brevity) takes over the processing if a primary replica (primary for brevity) fails. A backup is kept up to date by receiving the same operations as the primary (active replication [1]) or log records from the primary (passive replication [2]). The backup can either apply the log records to its own state or periodically receive checkpoints from the primary. Assuming that the *mean time to fail*, MTTF, is orders of magnitude larger than the *mean time to repair*, MTTR, the system only needs to be single fault tolerant to completely avoid the need for disk accesses. MTTR can be made very short by employing on-line self-repair mechanisms [3]. In addition, since disk accesses are slow compared to both RAM accesses and network latencies, replication can result in an improvement in performance.

A transaction is a collection of operations that transfers a system reliably from one state to another, while providing the ACID properties [4]: *Atomicity*, *consistency*, *isolation* and *durability*. Commonly, transaction termination and atomicity is satisfied by an *atomic commitment protocol*, ACP. The ACP has been showed to be an important factor of total transaction processing time and, in particular, the current industry leading standard, the Two-Phase Commit protocol, 2PC [5], is slow [6–8]. The delay caused by two rounds of messages and multiple log records flushed to disk cause a significant overhead. Also, a failure of the coordinator might block the participants from completing a transaction [9, 10].

ACP performance and resilience to failures is a well established research field, but optimizations that will have significant effect is still possible under a parallel and replicated paradigm. Thus, this paper presents an ACP called Circular Two-Phase Commit protocol, C2PC. It is an optimized version of 2PC for primary-backup systems. The protocol takes advantage of replication to trade costly flushed disk writes for cheaper message sends and RAM accesses. The idea is to send the vote and decision to the backup instead of a disk. This provide availability for the transaction participants and coordinator and renders 2PC *non-blocking* [9, 11]. To give better performance, the vote and decision are sent in a ring instead of back and forth between the primary and backup. The protocol is always single fault-tolerant and these methods could be favorably applied in a shared-nothing, fault-tolerant DBMS like ClustRa [3].

The rest of the paper is organized as follows: Section 2 summarizes related work. Section 3 presents the system model and Section 4 defines the non-blocking atomic commitment problem. Section 5 gives an overview and a detailed description of C2PC, proves the correctness of the protocol and outlines a one-phased version called C1PC. Then, an evaluation of the protocols is given in Section 6. Finally, the conclusion and further work are presented in Section 7.

2 Related Work

Several atomic commitment protocols and variations have been proposed over the years. Many approaches have been concerned with either developing a non-blocking protocol or the performance issues. However, only a few deal with both.

In a non-replicated environment, 2PC may block if the coordinator and a participant fail [9, 10]. 3PC [12] decreases the chance of blocking failures by adding an extra round of messages, thus favoring resilience over performance. 3PC has been extended to partitioned environments [13], and the number of communication steps has been reduced to the same as 2PC by using consensus [14], causing an increase in the number of messages or requiring broadcast capabilities.

Several 2PC-based modifications where performance issues are handled exist [15]. Presumed commit and presumed abort [16] both avoid one flushed disk write, by assuming that a non-existent log record means that the transaction has committed or aborted, respectively. Transfer-of-commit, lazy commit and read-only commit [9], sharing the log [16, 17] and group commit [18, 19] are other optimizations. An optimization of the presumed commit protocol [7] reduces the number of messages, but requires the same number of forced disk writes.

Optimistic commit protocols are designed to give better response time during normal processing, but will need extra recovery after failures or aborts. They release locks when the transaction is prepared, but must be able to handle cascading aborts by using semantic knowledge [20]. PROMPT [8] uses optimistic locking in the sense that locks can be lent to other transactions after the participant has voted yes. A transaction that lends locks will not reply to the request until the locks are fully released by the previous transaction, and only one transaction at a time can lend a lock. This approach avoids cascading aborts while it may yield better performance because of increased concurrency.

One-phased commit protocols have also been proposed [17, 21–24]. These are based on the early prepare or unsolicited vote method by Stonebraker [25] where the prepare message is piggybacked on the last operation sent to a participant. In this way, the voting phase is eliminated. However, these approaches inflicts strong assumptions and restrictions on the transactional system [22]. For instance, it requires either the participants to prepare the transaction for each request-reply interaction, or the coordinator must be able to identify the last request for a transaction to be able to piggyback a prepare-request. Otherwise, the performance of 1PC degrades.

A few approaches that render 2PC non-blocking by replication have been proposed. The first replicates the coordinator, but not the participants [11]. In addition to sending log records to the backup, they are forced to disk, causing a decrease in performance. Also, the backup only finishes transactions already started. No new transactions can be initiated by the backup. This approach has also been adapted to multiple backups [26].

The second combines optimistic commit and replication [27]. A replicated group of commit servers is used to keep the log records not yet written to the log by the participant available, thus ensuring resilience to failures. This approach uses multicast and has the same latency as 2PC, but requires more messages to be sent.

A third approach [28] is the most similar to the approach adopted in this paper. The differences are that it incurs unnecessary overhead by sending the "start of prepare" and the commit log records to the backup, and it forces log records to the disk even if both the primary and the backup work correctly. The performance is thus degraded.

3 System Model

The system is composed of a number of processes or nodes connected through a communication network. Each process has both a functional unit (application or database server) and a transaction manager. A process executes two kinds of actions. (1) Change state and (2) send or receive a message. When correct, they execute at arbitrary speeds, but eventually make progress. Processes fail by crashing, causing them to lose state. Such events are, however, rare. A failed process is recovered and brought up-to-date by the system.

Communication is *asynchronous* and *reliable*. Thus, there are no bounds on communication delays and messages are not corrupted or lost if both the receiving and the sending process behaves correctly, i.e. do not crash.

In an asynchronous system, a failure detector is needed to make the system reliable [29]. An *eventually strong* failure detector can solve the atomic commitment problem [30]. However, to simplify the problem descriptions and explanations a perfect failure detector that eventually suspects every faulty process and never suspects a correct process is assumed.

For the purpose of this paper no disks are used. State is stored entirely in main-memory. Thus, to make the state persistent and the system highly available the *primary-backup approach* [2] is used. This approach assumes that MTTF is orders of magnitude larger than MTTR, thus both the primary and backup do not fail at the same time.

Following [16], the costs of execution in this system are twofold. (1) The computation cost is the total number of messages sent, and (2) the delay is serialized messages. The main memory operations associated with the atomic commitment protocol are only a small fraction of the load on the system, thus their costs are assumed to be negligible. Also, as long as the processors are not fully utilized, there are no queueing effects.

4 The Non-Blocking Atomic Commitment Problem

An atomic commitment protocol ensures that the participants in a transaction agrees on the outcome, i.e. ABORT or COMMIT. Each participant vote, YES or NO, on whether they can guarantee the local ACID properties of the transaction. All participants has a right to *veto* the transaction, thus causing it to abort. The *Non-Blocking Atomic Commitment* problem, NB-AC, has these properties [10, 30]:

NB-AC1 <*uniform agreement*> All processes that decide reach the same decision.

NB-AC2 <*integrity*> A process cannot reverse its decision after it has reached one.

NB-AC3 <*uniform validity*> COMMIT can only be reached if *all* processes voted YES.

NB-AC4 <*non-triviality*> If there are no failures and no processes voted NO, then the decision will be to COMMIT.

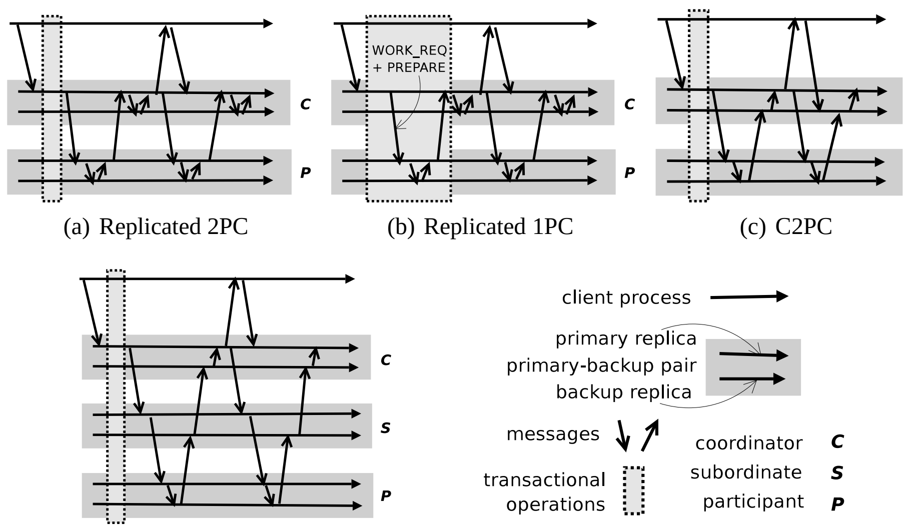
NB-AC5 <*termination*> Every correct process eventually decides.

5 The Circular Two-Phase Commit Protocol

This section presents the *Circular Two-Phase Commit protocol*, C2PC, for main memory primary-backup systems.

Normally, 2PC requires both forced and non-forced disk writes [16, 9]. In a primary-backup environment these disk writes can be replaced by, respectively, synchronous (blocking) and asynchronous (non-blocking) logging to the backup node. Figure 1(a) illustrates this. The small arrows between each primary-backup pair is the logging.

2PC (Figure 1(a)) consists of two phases, a voting phase and a decision phase. In the voting phase the votes are collected by a coordinator, and the coordinator makes



(a) Replicated 2PC (b) Replicated 1PC (c) C2PC

(d) C2PC with subordinate (e) Legend for Figure 1

Fig. 1. Execution of various atomic commitment protocols

a decision depending on the votes and persistently stores the decision. In the decision phase, the outcome is sent to the participants which send an acknowledgement back to the coordinator. Each participant must persistently store its vote and the outcome before replying to the coordinator in, respectively, the voting and decision phase. After the decision has been made persistent, the coordinator can give an *early answer* [3] to the client. Thus, the response time seen from the client is less than what it would be if the second phase had to be completed before the reply.

1PC (Figure 1(b)) piggybacks the prepare-message on the last work request for the transaction. Thus, the first phase of the voting is eliminated. However, each participant's vote must be persistently stored to the backups before replying to the coordinator.

The C2PC protocol is a modified version of 2PC for main memory primary-backup systems. Similarly to 2PC, C2PC has two phases and logs the votes and decision to the backups. However, it allows the backup to reply to the backup coordinator. This is shown in Figure 1(c). Instead of sending votes and acknowledgments back and forth the votes and decision are sent in a ring for each branch of the commit tree. This is a case of the transfer-of-commit optimization [9] where the authority to commit is passed via the participants to the backup root coordinator.

C2PC reduces both the number of messages in the critical path and the total number of messages to commit a transaction. The critical path is the delay until the transaction coordinator can give an early answer to the client. For instance, comparing Figure 1(a) and 1(c), the added delay has been reduced from six to four messages and the added number of messages from thirteen to nine. By comparison, 1PC (Figure 1(b)) has an added delay of four, two within the transactional operations frame and two after, and a total overhead of eleven messages.

During normal processing, the communication goes through each ring twice, one for each phase as seen in Figure 1(c). In the first round, the primary coordinator, *pc*, votes and piggybacks its own vote on the prepare message to the primary participant, *pp*. Each *pp* vote and sends its vote along with the vote of the *pc* to the backup participant, *bp*. *Bp* adds its own vote and forwards it to the backup coordinator, *bc*. *Bc* makes a decision based on the received votes and its own. The decision is then made persistent by sending it to the *pc*, which gives an early answer to the client and initiate the second phase.

The protocol also handles subcoordinators, or *subordinate* processes [16]. A subordinate acts as a participant to the coordinator and as a coordinator to the participants. A subordinate can also act as a participant to another subordinate. During the first phase a primary subordinate, *ps*, votes and forwards the vote to each of the subparticipants. The backup subordinate, *bs*, collects the votes from all the subparticipants before it sends its vote to the *bc*. During the second phase the decision is propagated in the same fashion.

If, during the first phase, one of the participants or subordinates votes NO, the vote is propagated back to the *bc*, while each subordinate along the way makes the decision to abort. The decision is then sent out to all remaining undecided participants and subordinates.

The protocol handles failures of both the primary and the backup. These failure scenarios might occur:

- If one of the primaries fails during the first phase, the transaction is aborted as the backup cannot be sure that it has all the log records.
- If one of the backups fails during the first phase, the preceding node in the ring sends the vote message to the primary instead.
- If one of the participating primaries (resp. backups) fails during the second phase, the preceding node in the ring sends the decision or acknowledgement message to the backup (resp. primary) instead.

Rerouting the messages to the non-failed primary or backup in the last two scenarios above works since the primary and backup is assumed never to fail at the same time.

First, a detailed explanation is given, second, the correctness of the protocol is proven and, third, a one-phase version of C2PC, C1PC, is outlined.

5.1 Detailed Description

This section presents the C2PC protocol in detail. Listings 1.1 to 1.6 present the protocol in failure free scenarios for all types of nodes.

Each process has a Transaction Table (TT) which holds the state (*active*, *prepared*, *committed* or *aborted*) and known participants of each transaction. Also, it is told which processes have failed from the local failure detector. Log records marked with a Log Sequence Number (LSN) [10] are shipped asynchronously to the backup. The backup checks that it has received all LSNs and acknowledges the greatest LSN received so far. The TT of the primary holds the greatest LSN acknowledged so far by the backup, and the backup TT is updated as log records are received and acknowledged from the primary. When voting, any unacknowledged log records are piggybacked on a `VoteMsg`. The TT can also be changed by receiving a vote message, `VoteMsg`, from a participant.

First, the protocols for the coordinator and the participants are presented. Then, the protocols for the subordinates are given.

Coordinator and Participants As seen in Listing 1.1 the *pc* of the transaction initiates the protocol by attaching its own vote to a `VoteMsg` and sending it to each of the participants.

Some necessary information is included in all messages going down in the commit tree: (1) The transaction identifier, (2) the address of the primary and backup of the *pc* and (3) the address of the client. The first identifies the transaction to be committed, while the second allows *bp* to contact *bc*. The third allows *bc* to contact the client to complete the transaction should *pc* fail. Also, included in at least one of the vote messages are (4) the unacknowledged log records of the transaction at *pc* and (5) a list of the participants of the transaction. The fourth ensures that *bc* has all the log records generated by *pc* of the transaction before committing it. Finally, the fifth guarantees that *bc* waits for `VoteMsg`s from all the participants before making a decision and enables it to complete a transaction in case *pc* fails.

Each *pp* (Listing 1.3) of the transaction receives a `VoteMsg`. If the received vote or its own is NO, the decision is ABORT, and a new `VoteMsg` with a NO-vote is sent to the backup. If the vote is YES, *pp* adds its unacknowledged log records for the transaction to the vote message and forwards it to the backup.

```
atomic_commitment:                                    1
if (myVote == NO) {
   decide(ABORT);
   voteMsg = new VoteMsg(txn,No);                     4
   send (voteMsg) to all participants;
} else {
   voteMsg = new VoteMsg(txn,Yes);                    7
   send (voteMsg) to all participants;
   receive(DecisionMsg) from backup {
      if (decision is COMMIT) decide(COMMIT);         10
      else decide(ABORT);
      dMsg = new DecisionMsg(txn,decision);
      send reply to client;                           13
      send (dMsg) to all participants;
      if (decision is COMMIT) {
         receive (AckMsg) from backup;                16
         on timeout {resend dMsg;}
} } }
```

Listing 1.1. Primary coordinator

```
atomic_commitment:
receive (voteMsg) from all partipants;                20
if (receivedVotes == NO || myVote == NO) {
   decide(ABORT);
   dMsg = new DecisionMsg(txn,ABORT);                 23
} else {
   decide(COMMIT);
   dMsg = new DecisionMsg(txn,COMMIT);                26
}
send (dMsg) to primary;
if (decision is COMMIT) {                             29
   receive (DecisionMsg) from all;
   receive ack from client;
   send (AckMsg) to primary;                          32
}
```

Listing 1.2. Backup coordinator

```
atomic commitment:
receive(voteMsg);
if (receivedVote == NO || myVote == NO) {             35
   decide(ABORT);
   vMsg = new VoteMsg(txn,NO);
   send (vMsg) to backup;                             38
} else {
   txnLog = getLog(txn);
   vMsg = new VoteMsg(txn,vote,info);                 41
   send (vMsg) to backup;
   receive(decisionMsg) {
      decide(decisionMsg.decision);                   44
      send (decisionMsg) to backup;
} }
```

Listing 1.3. Primary participant

```
atomic_commitment:                                    47
receive (voteMsg);
if (receivedVote == NO || myVote == NO) {
   decide(ABORT);                                     50
   vMsg = new VoteMsg(txn,NO);
   send (vMsg) to parentBackup;
} else {                                              53
   vMsg = new VoteMsg(txn,YES);
   send (vMsg) to parentBackup;
   receive (decisionMsg) {                            56
      decide(decisionMsg.decision);
      send (AckMsg) to parentBackup;
} }                                                   59
```

Listing 1.4. Backup participant

When a YES-vote is received by a *bp* (Listing 1.4), the log records from the *pp* are removed from the `VoteMsg` and applied to the local log. The local vote is then collected and forwarded to the backup of the parent. If the local vote or the received one is NO, the decision is ABORT and a `VoteMsg` containing a `No`-vote is forwarded to *bc*.

Listing 1.2 shows the algorithm for *bc*. Upon receiving a `VoteMsg`, it checks if the message contains a list of participants. If so, it checks whether or not it has received a `VoteMsg` from all of them. If a list is not included, it knows that there are more messages coming. Either way, it waits until all participants' votes have been collected (line 20), and then makes a decision: ABORT if any NO-votes have arrived or itself votes NO, otherwise COMMIT. Any unacknowledged log records sent from *pc* are appended to the local log and a decision message, `DMsg`, is then sent to *pc*.

When the *pc* receives a `DMsg`, it decides the same and then forwards the decision to the client and the participants. If the decision is COMMIT it waits to receive a confirmation from *bc* saying that all participants have committed before the transaction can be removed from TT.

After voting, the participants waits for a decision. When received, the decision is made, and the message is sent from the *pp* to the *bp* to the *bc*. Note that since the *pc* and

```
atomic_commitment:
receive(voteMsg);
if (receivedVote == NO || myVote == NO){
   decide(ABORT);
   vMsg = new VoteMsg(txn,NO);
   send (vMsg) to participants;
} else {
   txnLog = getLog(txn);
   voteMsg = new VoteMsg(txn,vote);
   send (voteMsg) to participants;
   receive(decisionMsg) {
      decide(decisionMsg.decision);
      send (decisionMsg) to participants;
} }
```

Listing 1.5. Primary subordinate

```
atomic_commitment:
receive (voteMsg) from all subpartipants;
if (receivedVotes == NO || myVote == NO) {
   decide(ABORT);
   vMsg = new VoteMsg(txn,NO);
   send (vMsg) to parentBackup;
} else {
   vMsg = new VoteMsg(txn,YES);
   send (vMsg) to parentBackup;
   receive (decisionMsg) {
      decide(decisionMsg.decision);
      send (DecisionMsg) to parentBackup;
} }
```

Listing 1.6. Backup subordinate

the *bc* are assumed not to fail at the same time, a termination protocol is not needed for the participants, because the coordinator ensures the liveness of the transaction.

Subordinate processes The previous subsection is necessary to make an atomic commitment, but internal nodes in the commit tree can also exist. These nodes are called *subordinates* [16] and are characterized by acting as a coordinator for some participants, while being a participant itself for the coordinator or other subordinate.

The protocol for a primary subordinate, *ps*, is given in Listing 1.5. When a `VoteMsg` is received, it decides ABORT if the received or its own vote is NO. Otherwise, the unacknowledged log records are appended to at least one of the outgoing `VoteMsg`s along with a list of the participants. Either way, the address of the current *ps* and *bs* is sent to the participants along with the vote and the information received in the `VoteMsg`.

A backup subordinate, *bs*, (Listing 1.6) waits, as the *bc*, until a `VoteMsg` is received from all its participants and then makes a decision based on the received vote and, if all votes are YES, the result of applying the log records received from the primary. The information from the parent primary is added to the `VoteMsg`, and it is sent to the parent backup.

In the same way as the participants, the subordinates waits for a decision after voting. When received, the decision is made, and the message is sent from the *ps* to a *pp* or another *ps*. The *bs* receives the decision from one or more *bp*s or *bs*s, and forwards it to the *bc*. For the same reasons as for the participants, a termination protocol is not needed here.

5.2 Correctness

This section proves the correctness of the C2PC protocol by proving each of the properties given in Section 4 in this order: **NB-AC2**, **NB-AC3**, **NB-AC4**, **NB-AC1** and **NB-AC5**.

Lemma 1. *NB-AC2: A process cannot reverse its decision after it has reached one.*

Proof. The algorithms for each of the processes use if-else statements to avoid deciding more than once per process.

Lemma 2. *NB-AC3: The* COMMIT *decision can only be reached if all processes voted* YES.

Proof. All processes can decide COMMIT during the second phase of the protocol. However, they can only decide COMMIT if they receive a message with a COMMIT decision. The only process that can decide COMMIT during the first phase is *bc* (line 25). This happens only if it has received YES-votes from all the participating processes including itself.

Lemma 3. ***NB-AC4****: If no process failed and no process voted* NO*, then the decision will be to* COMMIT.

Proof. If no process failed, and no process voted NO, then since the communication system is reliable, *bc* receives YES from all participants and subordinates. Thus, COMMIT is reached (line 25).

Lemma 4. ***NB-AC1****: All processes that decide reach the same decision.*

Proof. A process can only decide ABORT during the second phase, if a process decided ABORT during the first phase. Similarly, a process can only decide COMMIT during the second phase, if *bc* decided COMMIT during the first phase. As proved in Lemma 2, COMMIT can be decided (line 25) only if all processes voted YES. A process can only decide ABORT during the first phase if it votes NO. A process cannot both vote YES and NO, so two processes cannot decide differently.

Lemma 5. ***NB-AC5****: Every correct process eventually decides.*

Proof. To enable a process to decide in the presence of failures, all failure scenarios as well as the scenario with no failures must be handled. These scenarios can occur:

1: *Pc* fails before sending the vote to all participants.
2: *Pc* fails after initiating the voting, but before sending the decision to all participants.
3: *Pc* fails after sending the decision to all participants, but before receiving an `AckMsg` from *bc*.
4: *Bc* fails before sending the decision to *pc*.
5: *Bc* fails after sending the decision to *pc*, but before sending `AckMsg` to *pc*.
6: A *ps*, *bs*, *pp* or *bp* fails before sending the vote.
7: A *ps*, *bs*, *pp* or *bp* fails after sending the vote, but before sending the decision.
8: No node fails.

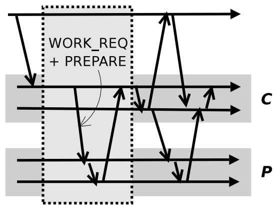
Scenario (1): None has voted, each of the participants can independently abort the transaction after a timeout has expired without causing inconsistencies in the system.

Scenario (2): When *bc* does not receive a decision from any of the participants and *pc* fails, *bc* can complete the transaction with the decided outcome.

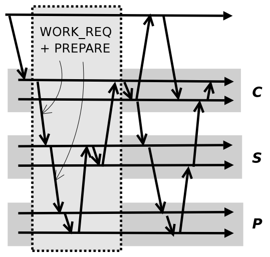
Scenario (3): The `AckMsg` is sent to *pc* to allow it to purge the transaction entry from its TT. However, this is not needed if *pc* fails, because it will have to update its TT as part of the recovery process.

Scenarios (4) and (6): If *pc* does not receive a decision within a given time limit it can send a message to *bc* and tell it about the timeout, then *bc* can decide Abort. If *bc* has failed, *pc* can safely abort the transaction.

Scenario (5): The transaction is completed, but if the decision is COMMIT the transaction entry will not be deleted from the TT of *pc* until an `AckMsg` is received. However, *pc* resends the decision (line 17) with updated backup information until it receives confirmation that all participants have decided.



(a) C1PC



(b) C1PC with subordinate

Fig. 2. Examples of C1PC execution

Scenario (7): When a process fails during the second phase, the decision must be sent via the backup on its way down the commit tree or via the primary on its way up the tree. *Pc* resends the decision (line 17) until it receives an acknowledgement, and the failed processes are bypassed.

Scenario (8): This is proven similarly to Lemma 3. Since no process failed and the communication system is reliable, *bc* receives votes from all participants and subordinates. Thus, it decides either COMMIT in line 25 or ABORT in line 22. By the same argument each participant and subordinate eventually decides.

All scenarios are handled, thus, all correct processes eventually decides.

Theorem 1. *C2PC is a valid non-blocking atomic commitment protocol.*

Proof. Since C2PC satisfies properties **NB-AC1** - **NB-AC5** it solves NB-AC. ■

5.3 C1PC

Circular One-Phase Commit protocol, C1PC, is a circular version of 1PC and can be designed as shown in Figure 2. The main differences between C1PC and C2PC are: During the first phase (1) *pc* piggybacks `VoteMsg` on the last request and (2) *bp* replies to *ps* or *pc* (instead of *bs* or *bc*) because there might be results that are needed. During the second phase, (3) *pc* makes the decision to commit, and (4) *bc* replies to the client and sends the `DMsg` to the participants.

6 Evaluation

This section compares the performance of non-fault tolerant, replicated 2PC, replicated 1PC, C2PC and C1PC. We assume the normal operational mode where no participating processes fails and all participants vote YES. The purpose is to evaluate the costs associated with the various protocols.

The table in Figure 3 shows formulas for the added number of messages in the critical path and the total overhead to complete a transaction compared to the non-fault tolerant case. The critical path is the delay until the transaction coordinator can give an early answer to the client. Parallel and linear execution corresponds to a commit-tree of height 1 and $N-1$ respectively.

Protocol	Delay	Total
Non-fault tolerant	0	0
Replicated 2PC, parallel	6	$8N + 5$
Replicated 2PC, linear	$4N + 4$	$8N + 5$
Replicated 1PC, parallel	4	$6N + 5$
Replicated 1PC, linear	$2N + 2$	$6N + 5$
C2PC, parallel	4	$8N + 1$
C2PC, linear	$2N + 1$	$4N + 5$
C2PC, hybrid	4	$6N + 1$
C1PC, parallel	2	$4N + 3$
C1PC, linear	$N + 1$	$3N + 4$
C1PC, hybrid	2	$3N + 4$

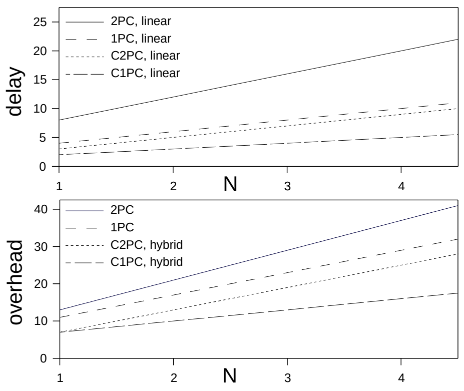


Fig. 3. Added delay until early answer to client and total overhead for various ACPs. N = # servers invoked by transaction excluding the coordinator, $N \geq 1$

The non-fault tolerant case is non-replicated and has zero delay and overhead to complete the request. It does not tolerate any failures and there is no coordination of the outcome.

For the transactional cases, the parallel versions of C2PC and C1PC have the shortest delay and the linear versions have the least overhead. This observation leads to the *hybrid* versions of C2PC and C1PC, where the voting phase is executed in parallel and the decision phase in linear. This minimize both the delay and the overhead.

The graphs in Figure 3 depicts the delay and overhead of selected protocols. The protocols with constant delay are not shown in the delay graph and in the overhead graph the linear and parallel circular protocols are not showed to avoid cluttering.

The delay of parallel and hybrid C2PC is equal to and two-thirds of the delay of 1PC and 2PC, respectively. C1PC halves the delay and almost halves the overhead compared to 1PC, but also inherits its restrictions and assumptions [22]. The overhead of the parallel and hybrid versions of C1PC is almost half of that of 1PC, and hybrid C2PC has less overhead than 1PC.

7 Conclusion

This paper has presented an atomic commitment protocol, Circular Two-Phase Commit (C2PC). It is an single fault-tolerant optimization of 2PC for replicated main-memory primary-backup systems. C2PC does not require any changes to the standard 2PC interface, and can be implemented in an asynchronous system with an unreliable failure detector. The protocol is unique in the sense that it does not log to disk and ensures liveness for both data, processing and transaction commitment.

For further work the protocol should be implemented and performance measures should be made to verify the analysis and evaluation in Section 6.

References

1. Schneider, F.B.: Replication management using the state machine approach. In: Distributed systems (2nd Ed.). ACM Press/Addison-Wesley Publishing Co. (1993) 169–197

2. Budhiraja, N., Marzullo, K., Schneider, F.B., Toueg, S.: Distributed systems. In Mullender, S., ed.: Distributed Systems. ACM Press. second edn. Addison-Wesley (1993) 199–216
3. et al., S.O.H.: The ClustRa telecom database: High availability, high throughput, and real-time response. In: Proc. of VLDB. (1995)
4. Härder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. **15** (1983) 287–317
5. Gray, J.: Notes on data base operating systems. In: Operating Systems, An Advanced Course, London, UK, Springer-Verlag (1978) 393–481
6. Spiro, P.M., Joshi, A.M., Rengarajan, T.K.: Designing an optimized transaction commit protocol. j-DEC-TECH-J **3** (1991) 70–78
7. Lampson, B., Lomet, D.: A new presumed commit optimization for two phase commit. In: Proc. of VLDB. (1993)
8. Haritsa, J.R., Ramamritham, K., Gupta, R.: The prompt real-time commit protocol. IEEE Trans. Parallel Distrib. Syst. **11** (2000) 160–181
9. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993)
10. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency control and recovery in database systems. Addison-Wesley Longman Publ. Co., Inc. (1986)
11. Reddy, P.K., Kitsuregawa, M.: Reducing the blocking in two-phase commit protocol employing backup sites. In: Proc. of CoopIS. (1998)
12. Skeen, D.: Nonblocking commit protocols. In: Proc. of SIGMOD. (1981)
13. Rabinovich, M., Lazowska, E.D.: A fault-tolerant commit protocol for replicated databases. In: Proc. of PODS. (1992)
14. Guerraoui, R., Larrea, M., Schiper, A.: Reducing the cost for non-blocking in atomic commitment. In: (ICDCS), Hong Kong (1996) 692–697
15. Samaras, G., Britton, K., Citron, A., Mohan, C.: Two-phase commit optimizations and tradeoffs in the commercial environment. In: Proc. of ICDE. (1993)
16. Mohan, C., Lindsay, B., Obermarck, R.: Transaction management in the R* distributed database management system. ACM Trans. Database Syst. **11** (1986) 378–396
17. Stamos, J.W., Cristian, F.: A low-cost atomic commit protocol. In: Proc. of SRDS. (1990)
18. Gawlick, D., Kinkade, D.: Varieties of concurrency control in IMS/VS Fast Path. IEEE Database Eng. Bull. **8** (1985) 3–10
19. Park, T., Yeom, H.Y.: A consistent group commit protocol for distributed database systems. Proc. of PDCS (1999)
20. Levy, E., Korth, H.F., Silberschatz, A.: An optimistic commit protocol for distributed transaction management. In: Proc. of SIGMOD. (1991)
21. Abdallah, M., Pucheral, P.: A single-phase non-blocking atomic commitment protocol. In: Proc. of DEXA. (1998)
22. Abdallah, M., Guerraoui, R., Pucheral, P.: One-phase commit: Does it make sense? In: Proc. of ICPADS, Washington, DC, USA (1998)
23. Lee, I., Yeom, H.Y.: A single phase distributed commit protocol for main memory database systems (2002)
24. Stamos, J.W., Cristian, F.: Coordinator log transaction execution protocol. Distributed and Parallel Databases **1** (1993) 383–408
25. Stonebraker, M.: Concurrency control and consistency of multiple copies of data in distributed ingres. IEEE Trans. Software Eng. **5** (1979) 188–194
26. Reddy, P.K., Kitsuregawa, M.: Blocking reduction in two-phase commit protocol with multiple backup sites. In: DNIS. (2000)
27. Jiménez-Peris, R., Patiño-Martínez, M., Alonso, G., Arévalo, S.: A low-latency non-blocking commit service. In: Proc. of DISC. (2001)
28. Mehrotra, S., Hu, K., Kaplan, S.: Dealing with partial failures in multiple processor primary-backup systems. In: Proc. of CIKM. (1997)
29. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43** (1996) 225–267
30. Guerraoui, R.: Revisiting the relationship between non-blocking atomic commitment and consensus. In: WDAG. (1995)