2.5-PC: A Faster and Non-Blocking Atomic Commit Protocol

Adrian Berding
Department of Computer Science
University of Michigan
Ann Arbor, USA
aberding@umich.edu

Sanjay Sri Vallabh Singapuram
Department of Computer Science
University of Michigan
Ann Arbor, USA
singam@umich.edu

Drake Svoboda
Department of Computer Science
University of Michigan
Ann Arbor, USA
drakes@umich.edu

***Abstract*—We present 2.5PC, a non-blocking atomic commit protocol. 2.5PC is a modification of 3PC for synchronous settings with reliable network channels. 2.5PC waits for the same number of message delays as 2PC while also being non-blocking in the presence of non-total failures. We provide an informal proof of 2.5PC's correctness, and formally prove the safety of 2.5PC during normal operation using Dafny. Although 2.5PC relies on impractical network settings, the proof of 2.5PC improves on the current understanding of non-blocking atomic-commit protocols. Using this understanding, we show a practical improvement to 3PC regarding the coordinators timeout delay for the `ACK` messages during the third phase of 3PC. This improvement has practical implications for other protocols that use empty acknowledge messages.**

***Index Terms*—atomic commitment, two-phase commit, three-phase commit, non-blocking commit protocols, TLA, Hoare-logic verification**

I. Introduction

Atomic Commitment has been a main-stay problem in distributed systems since it's inception. Several real-world systems rely on atomic commit protocols (ACPs) for consistency [1]–[4]. Two widely used ACPs are 2-phase commit (2PC) and 3-phase commit (3PC) [5]. There are many versions of 2PC and 3PC, this paper will focus on those versions that operate under the assumption of synchronous message channels. As the names suggest, 3PC takes an additional phase to reach termination during normal execution. This additional phase allows 3PC to be non-blocking. That is, 3PC does not block in the presence of non-total failures, whereas 2PC's progress can stall in the presence of partial failures. For this reason, we call 3PC a non-blocking ACP. Despite 3PC being non-blocking, 2PC is favored in most practical systems since it takes one less phase (i.e. one less round trip time). In this paper we discuss why 3PC needs an additional phase to be non-blocking. We then describe 2.5PC, a new ACP that is both non-blocking and takes only 2 phases.

2.5PC is a modification of 3PC that relies on reliable message channels. The assumption of reliable channels is generally impractical. Despite having limited practical application, the development of 2.5PC improves upon the current understanding of why 3PC is non-blocking. Using this understanding, we present one potential optimizations for 3PC that applies in more practical network settings.

Section VIII describes a machine verification of the safety of 2.5PC during normal execution. Machine verification is the gold standard for proving the correctness of distributed protocols. Distributed systems are often infinitely subtle and can be exceedingly difficult to reason about. In fact, the original specification of 3PC is incorrect in certain cases of multiple processes failures [6]. Machine verification can be used to illuminate and eliminate such errors.

II. Related Work

IronFleet shows that formal machine verification is practical at the scale of real-world implementations of distributed systems [7]. An older yet established system called TLA+ [8], has been used to specify and verify distributed systems for over a decade, including on Amazon's AWS services. IronFleet improves over TLA+ by enabling the specification to also act as an implementation and also proves certain liveness properties. mCRL2, another specification language, was used to model and verify 2PC and 3PC under different network conditions [6].

III. Overview of Atomic Commitment and 3PC

This section briefly goes over Atomic Commitment and 3PC. Consider a transaction T whose execution relies on several distributed processes $S_1, S_2, .., S_n$. T can only be executed successfully if each process commits to executing the transaction. If T is committed by some processes, but aborted by others, then T will be terminated inconsistently. An atomic commit protocol (ACP) is needed to ensure that T is either committed at every process, or aborted at every process, even if processes are allowed to fail. Concretely, a correct ACP satisfies the following properties [9]:

- AC1: All processes that reach a decision reach the same one.
- AC2: A process cannot reverse its decision after it has reached one.
- AC3: The `COMMIT` decision can only be reached if all processes voted `YES`.
- AC4: If there are no failures and all processes voted `YES`, then the decision will be to `COMMIT`.
- AC5: If all failures are repaired and there are no more failures, then all processes will eventually decide.

	Aborted	Uncertain	Pre-commited	Committed
Aborted	✓	✓	×	×
Uncertain	✓	✓	✓	×
Pre-committed	×	✓	✓	✓
Committed	×	×	✓	✓

Fig. 1. **NB Property Coexistence Matrix.** Check-marks indicate that two processes in the given states can coexist. The **NB** property disallows the coexistence of uncertain and committed processes.

Additionally, an ACP is considered non-blocking if operational processes can always reach a decision by examining their local states, even when other processes have failed.

3PC satisfies AC1-AC5 and is non-blocking in the presence of partial failures. 3PC designates a single coordinator processes who oversees the execution of the protocol for the other participating processes. The execution of 3PC is as follows:

1) The coordinator broadcasts VOTE−REQ to each participant
2) Each participant receives a VOTE−REQ and responds with a YES vote or a NO vote. If a participant votes NO, it decides ABORT.
3) If the coordinator receives all YES votes, it broadcasts PRE−COMMIT. If the coordinator receives a NO vote, it decides ABORT and broadcasts ABORT.
4) Each participant receives a PRE−COMMIT or an ABORT. If a participant receives ABORT, it decides ABORT and terminates. If a participant receives PRE−COMMIT, it responds with an ACK.
5) The coordinator receives the ACKs, decides COMMIT, and broadcasts COMMIT.
6) Participants receive COMMIT and decide COMMIT.

IV. Why is 3PC Non-Blocking?

A processes uncertainty period is the period of time after that processes has voted YES, but before it has received a PRE−COMMIT or ABORT message. During this period, the processes is uncertain of what decision will be reached. 3PC is non-blocking because it satisfies the following non-blocking **NB** property.

> **NB**: If an operational processes is in its uncertainty period, then no processes has decided COMMIT.

The ACK message received in step 5 informs the coordinator that each processes has exited its uncertainty period. After receiving these messages, it becomes safe for the coordinator to decide COMMIT.

This version of 3PC operates under the assumption of synchrony. Synchrony allows for a processes to become certain of a another process's death after some timeout delay. After step 3, the coordinator sets a timeout of 2δ where δ is the upper bound on message delivery. If the coordinator fails to receive an ACK message from each process within 2δ time, it can be assured that any uncertain process has died and can safely COMMIT without violating **NB**.

Due to the **NB** property, if an uncertain process discovers the death of the coordinator, the uncertain process can reason that the dead coordinator has not decided COMMIT; therefore, the uncertain processes do not need to wait for the dead coordinator to recover to make progress.

We consider a weaker non-blocking (**WNB**):

> **WNB**: If a process has decided COMMIT, each alive uncertain processes will exit it's uncertainty period within θ time.

We define θ as the lower bound on how long it takes to be certain of another processes death. In synchronous systems, where timeouts are used to detect death, θ depends on δ, the upper bound for message delivery.

The **WNB** property allows for the coexistence of committed processes and alive and uncertain processes; however, if an alive and uncertain processes learns of the death of another processes, the alive and uncertain process knows that the dead processes has not decided COMMIT. If an ACP satisfies the **WNB** property, then alive and uncertain processes do not need to wait for the recovery of dead processes to make progress.

V. 2.5PC

We propose 2.5 phase commit (2.5PC), a non-blocking atomic-commit protocol that satisfies the **WNB** property. We modify the coordinator in 3PC to send COMMIT messages immediately after sending PRE−COMMIT messages, thus eliminating the ACK message and saving 1 round of communications. 2.5PC operates under the assumption of synchrony and reliable channels. Additionally, a process can only learn of the death of another process by means of a timeout (that is, Falcon [10] or something similar isn't used).

2.5PC follows a similar structure to 3PC and adopts all of its timeout actions. The execution of 2.5PC is as follows:

1) The coordinator broadcasts VOTE−REQ to each participant
2) Each participant receives a VOTE−REQ and responds with a YES vote or a NO vote. If a participant votes NO, it decides ABORT.
3) If the coordinator receives all YES votes, it broadcasts PRE−COMMIT, decides COMMIT, then broadcasts COMMIT. If the coordinator receives a NO vote, it decides ABORT and broadcasts ABORT to the processes that voted YES.
4) Each participant receives a PRE−COMMIT, COMMIT or ABORT message. If a participant receives ABORT, it decides ABORT and terminates. If a participant receives PRE−COMMIT, it enters the pre-committed state and waits for the COMMIT message from the coordinator. If a participant receives COMMIT, it decides COMMIT and terminates.
5) Participants waiting to receive the COMMIT message receive COMMIT and decide COMMIT.

Processes can timeout at steps (2), (3), (4), and (5). If a processes times-out in step (2), that processes can independently decide ABORT since no processes has yet to decide COMMIT. If the coordinator times out in step (3), it can also independently decide ABORT and broadcast ABORT to

the processes that voted YES. A process that times-out in steps (4) and (5) cannot independently decide. Instead a new coordinator is elected and the processes participate in 3PC's termination protocol described in [9]. This termination protocol satisfies the **NB** property and therefore the **WNB** property.

If the coordinator receives all YES votes in step (3), why does it bother sending PRE−COMMIT if it is going to send COMMIT anyway? The purpose of the PRE−COMMIT message is to ensure that the **WNB** property is maintained. After sending a vote to the coordinator, a processes sets a timeout for 3δ and expects to receive a message from the coordinator within that time. It chooses 3δ because there might still be a VOTE−REQ message in flight at the time of sending the vote. Each VOTE−REQ can take at most δ time to arrive. Each subsequent vote message will take at most δ time. And the resulting decision will take at most δ time, thus, 3δ. When the final YES vote arrives in step (3), the coordinator can reason that the earliest a processes set its timeout was 2δ time ago. From the perspective of the coordinator, the lower bound on when any processes can timeout is the remaining δ time. Due to our assumption of reliable channels, the coordinator can guarantee that each alive and uncertain processes will exit its uncertainty period within δ time by sending a PRE−COMMIT message to every-process. After sending PRE−COMMIT to each process, the coordinator broadcasts COMMIT. When a process receives a COMMIT message in step (4) or (5), it can reason that every alive and uncertain processes has a PRE−COMMIT message in flight. That is, if the coordinator died while broadcasting COMMIT, each uncertain processes will exit its uncertainty period before it learns of the coordinator's death. Thus, the process that received the COMMIT message can safely commit without violating the **WNB** property.

VI. Informal Proof of 2.5PC

In this section we sketch an informal proof of the correctness of 2.5PC that draws on the functional equivalence between the **WNB** and **NB** properties. The **NB** property is trivially stronger than the **WNB** property since the **NB** property does not allow for the coexistence of committed processes and alive and uncertain processes.

Since 2.5PC borrows much of it's protocol from 3PC, its correctness is predicated on the correctness of 3PC. 3PC and 2.5PC only differ in a single scenario when a process is uncertain while another process has decided COMMIT. All other possible states are covered under the old non-blocking property and the correctness of 3PC. Specifically, if any process decides COMMIT, we must ensure that any uncertain process will never decide ABORT and will eventually decide COMMIT.

Lemma 1. *If a coordinator decides COMMIT in step (4), no participant would have decided ABORT and every operational process will eventually decide COMMIT*

Proof. In the case where the coordinator does not die, then the coordinator will send a COMMIT message to each process.

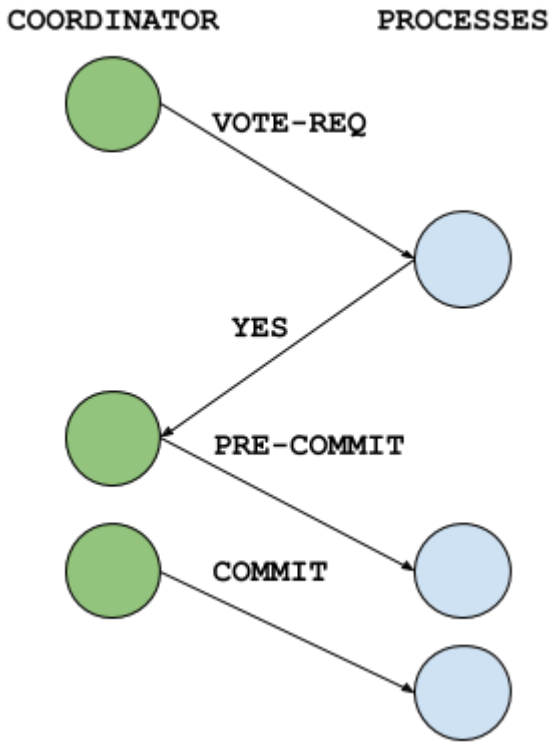

Fig. 2. **2.5PC**

From our network assumptions, each alive processes will receive the COMMIT message and decide COMMIT. Since the coordinator has decided COMMIT, it has received a YES votes from each processes. Thus, no processes has decided ABORT and the coordinator does not send an ABORT message.

In the case where the coordinator fails, since it has decided COMMIT, it has sent a PRE−COMMIT message to each processes. From our network assumptions, each processes will receive a PRE−COMMIT message within δ time. When the coordinator sends PRE−COMMIT, the earliest that a processes could timeout on the coordinator is also δ. Each process that does not received a COMMIT message will timeout on the coordinator only after receiving the PRE−COMMIT message and will enter 3PC's termination protocol in the PRE−COMMITTED state. From the correctness of the termination protocol, each processes will eventually decide COMMIT. □

Lemma 2. *If anyone has decided COMMIT, each dead uncertain process will not decide ABORT after recovering*

Proof. When an uncertain process recovers, there are two cases that need to be addressed.

The first case is a non-total failure. From Lemma 1, there exists an alive process that can eventually inform the recovering process of a COMMIT decision.

The second case is total failure. From the **WNB** property, when an uncertain process dies, it has not learned of the death of a process that has decided COMMIT. That is, the uncertain

process retains any process that could have decided COMMIT in its UP-Set. When such an uncertain process recovers, it will either learn of a COMMIT decision from processes that have already recovered, or block until a committed process in its UP-Set recovers. □

VII. The Acknowledgement Message

In this section we propose a small optimization to 3PC that works even in the presence of unreliable channels. One subtle aspect of 3PC is during step (5)—when the coordinator is waiting for ACKs—the coordinator will perform the same action whether it receives all ACKs or times-out. This is because either timing-out or receiving an ACK ensures the same thing—that the coordinator can safely decide COMMIT since the process has either exited its uncertainty period or crashed.

Before going into why the ACK message exists, it is important to understand how synchrony is guaranteed over un-reliable channels. In the presence of un-reliable channels, processes re-send their message multiple times ensuring—with extremely high probability—that at least one message will be delivered within δ time. This is how TCP guarantees reliable packet delivery. In 3PC, a coordinator sets a timeout for 2δ after sending a PRE-COMMIT message. If an alive coordinator times-out waiting for an ACK message after 2δ, it can be certain that the process it timed-out on has crashed. It is important that the coordinator is alive to conduct this reasoning because it needed to have been alive long enough to continuously re-send its message. If an alive coordinator waits 2δ time, it can safely reason that either the message it sent has been reliably delivered, or the receiving process has died. If an alive coordinator waits only a single δ time, it can also safely reason that either the message it sent has been reliably delivered, or the receiving processes has died and dropped the message. In either case, after δ time, it would be safe for the coordinator to decide COMMIT without violating the **NB** property. This is exactly the optimization we propose, the coordinator does not need to wait 2δ before deciding COMMIT.

So then why is there an ACK messages in 3PC? The coordinator could simply wait δ time after sending PRE-COMMIT without expecting an ACK message in return. The ACK messages will likely arrive well before δ time has elapsed, so it is likely faster to send the ACK message than to have the coordinator always wait a full δ time.

VIII. Machine Proof of 2.5PC

We prove the safety of 2.5PC inductively using Dafny, a programming language that supports formal specification [1]. We model the global state of 2.5PC as a sequence of Nodes (participant processes) and the messages that have been sent between them. We also model each state transition that the global state can undertake as an atomic protocol step. For simplicity, we only allow one processes to update its state during a global state transition. This is the approach taken by IronFleet [7] and can be shown to be equivalent to proofs of real systems. Figure 3 shows how the global state of the system is modeled in Dafny. Figure 4 shows one possible state transition for a single process.

[1]https://gitlab.eecs.umich.edu/drakes/2.5-phase-commit

```
0  datatype LS_State = LS_State(environment:PCEnvironment,
       servers:map<EndPoint,Node>)
1
2  predicate LS_Init(s:LS_State, config:Config)
3  {
4     LEnvironment_Init(s.environment)
5     && (forall index :: 0 <= index < |config| ==>
           NodeInit(s.servers[config[index]], index,
           config))
6  }
```

Fig. 3. **Initial State Predicate.** The environment variable models a synchronous network environment. The servers variable contains the set of participating processes. Line 4 indicates that the environment is in a valid initial state (no messages have been sent, etc.). Line 5 indicates that each participating processes is in a valid initial state (the processes hasn't decided, etc.).

```
0  predicate NodeDie(prev:Node, next:Node)
1  {
2     prev.is_alive && !next.is_alive
3     && prev == next.(is_alive := prev.is_alive)
4  }
```

Fig. 4. **Node Death Transition Predicate.** prev represents the previous state of a Node (a participant process) and next is the state of the node after the transition. Line 3 ensures that only the is_alive property is changed during the transition.

By defining the initial global state of our protocol and each possible state transition, we have reasonably defined the operation of 2.5PC (in an inductive manner) without implementing it.

We prove correctness of 2.5PC inductively by first express-

```
0  predicate AC1(state: LS_State) {
1     forall i, j :: i in state.servers && j in
          state.servers
2        ==> (state.servers[i].state == Committed ==>
             state.servers[j].state != Aborted)
3  }
4
5  predicate AC2(state: LS_State, state': LS_State) {
6     forall i :: i in state.servers
7        ==> i in state'.servers
8        && (state.servers[i].state == Aborted ==>
             state'.servers[i].state == Aborted)
9        && (state.servers[i].state == Committed ==>
             state'.servers[i].state == Committed)
10 }
11
12 predicate AC3(state: LS_State) {
13    forall i, j :: (i in state.servers && j in
         state.servers)
14       ==> (state.servers[i].state == Committed ==>
             state.servers[j].vote == Yes)
15 }
```

Fig. 5. **Safety Properties:** Safety properties of ACP modeled in Dafny. AC2 compares multiple states where state' comes after state.

```
lemma Init(state: LS_State)
   requires LS_Init(state)
   ensures AC1(state)
   ensures AC3(state)
{}
```

Fig. 6. **Simple Proof.** Proof that the initial state doesn't violate AC1, AC2, or AC3. Note that AC2 is trivially satisfied for any state snapshot.

```
lemma InvIsStronger(state: LS_State)
   requires Inv(state)
   ensures AC1(state) && AC3(state)
{}

lemma Init(state: LS_State)
   requires LS_Init(state)
   ensures Inv(state)
{}

lemma Induce(state: LS_State, state': LS_State)
   requires Inv(state)
   requires LS_Next(state, state')
   ensures Inv(state')
   ensures AC2(state, state')
{}
```

Fig. 7. **Proof of 2.5PC.** The `Inv` predicate encapsulates each inductive invariant. First we prove that `Inv` $\rightarrow$ `(AC1 && AC3)`. We then show that the initial state satisfies `Inv`. Finally, we show that any transition from a state that satisfies `Inv` also satisfies `Inv` and `AC2`.

ing the Atomic Commit (AC) safety properties in Dafny. With these safety properties implemented, we can begin to write simple proofs such as the proof in Figure 7. To prove the inductive step, we show that given a safe state, any valid transition from that state is also safe. The difficulty in proving the inductive step is that all states that satisfy AC1-AC3 may not be reachable from the initial state. We define a series of inductive invariants to prune out un-reachable safe states that could lead to unsafe states. One such invariant is that a `COMMIT` message can not be in flight if there is a process that has decided `ABORT`. We must also show that our initial state also satisfies each inductive invariant. A complete abstraction of our proof is in figure 7.

At the time of this paper, we have formally verified the correctness of 2.5PC under normal operation without the termination protocol. We have learned that Dafny programs are very hard to debug and verification often takes a significant time to run; however, it is extremely satisfying when the terminal reads `"Dafny program verifier finished with 87 verified, 0 errors."`

IX. Conclusion

We present 2.5PC, a new atomic commit protocol that operates under the assumption of synchronous and reliable channels. 2.5PC is able to complete in only two phases of messages during normal operation while also being non-blocking in the presence of non-total failures. We formally prove the safety of 2.5PC in the normal case using a combination of TLA and Hoare-logic verification. Our proof is written in Dafny, a programming language that supports formal verification.

a) Future Work: We have left the machine proof of the safety of the the termination protocol to future work.

Acknowledgment

We thank Prof. Kapritsos for pointing us three padawans to the light, and his patient and quick crash-course in Dafny.

References

[1] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google’s globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013. [Online]. Available: http://doi.acm.org/10.1145/2491245

[2] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding fork-join parallelism into llvm's intermediate representation," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17. New York, NY, USA: ACM, 2017, pp. 249–265. [Online]. Available: http://doi.acm.org/10.1145/3018743.3018758

[3] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: A new paradigm for building scalable distributed systems," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 159–174, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1323293.1294278

[4] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," 2011.

[5] D. Skeen, "A quorum-based commit protocol," Ithaca, NY, USA, Tech. Rep., 1982.

[6] M. Atif, "Analysis and verification of two-phase commit three-phase commit protocols," in *2009 International Conference on Emerging Technologies*, Oct 2009, pp. 326–331.

[7] C. Hawblitzel, J. Howell, M. Kapritsos, J. Lorch, B. Parno, L. Stephenson, S. Setty, and B. Zill, "Ironfleet: Proving practical distributed systems correct," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2015.

[8] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[9] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[10] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish, "Detecting failures in distributed systems with the falcon spy network," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 279–294. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043583