EasyCommit: A Non-blocking Two-phase Commit Protocol

Suyash Gupta, Mohammad Sadoghi
Exploratory Systems Lab
Department of Computer Science
University of California, Davis

ABSTRACT

Large scale distributed databases are designed to support commercial and cloud based applications. The minimal expectation from such systems is that they ensure consistency and reliability in case of node failures. The distributed database guarantees reliability through the use of atomic commitment protocols. Atomic commitment protocols help in ensuring that either all the changes of a transaction are applied or none of them exist. To ensure efficient commitment process, the database community has mainly used the two-phase commit (2PC) protocol. However, the 2PC protocol is blocking under multiple failures. This necessitated the development of the non-blocking, three-phase commit (3PC) protocol. However, the database community is still reluctant to use the 3PC protocol, as it acts as a scalability bottleneck in the design of efficient transaction processing systems. In this work, we present Easy Commit which leverages the best of both the worlds (2PC and 3PC), that is, non-blocking (like 3PC) and requires two phases (like 2PC). Easy Commit achieves these goals by ensuring two key observations: (i) first transmit and then commit, and (ii) message redundancy. We present the design of the Easy Commit protocol and prove that it guarantees both safety and liveness. We also present a detailed evaluation of EC protocol, and show that it is nearly as efficient as the 2PC protocol.

1 INTRODUCTION

Large scale distributed databases have been designed and deployed for handling commercial and cloud-based applications [11, 14–18, 35, 37, 46–48, 56, 57]. The common denominator across all these databases is the use of transactions. A transaction is a sequence of operations that either reads or modifies the data. In case of geo-scale distributed applications, the transactions are expected to act on data stored in distributed machines spanning vast geographical locations. These geo-scale applications require the transactions to adhere to ACID [22] transactional semantics, and ensure that the database state remains consistent. The database is also expected to respect the atomicity boundaries that is either all the changes persist or none of the changes take place. In fact atomicity acts as a contract and establishes trust among multiple communicating parties. However, it is a common knowledge [39] that the distributed systems undergo node failures. Recent failures [19, 38, 55] have shown that the distributed systems are still miles away from achieving undeterred availability. In fact there is a constant struggle in the community to decide the appropriate level of database consistency and availability, necessary for achieving maximum system performance. The use of strong consistency semantics such as serializability [6] and linearizability [30] ensures system correctness. However, these properties have a causal effect on the underlying parameters such as latency and availability. Hence, a requirement for stronger consistency leads to a reduction in system availability.

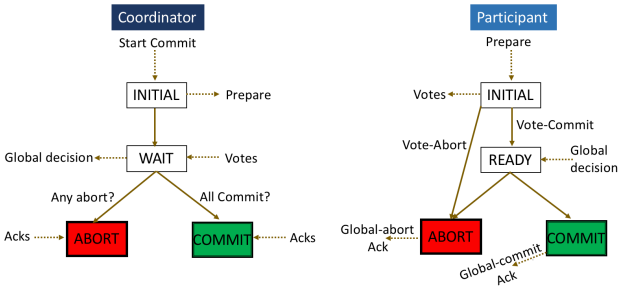


Figure 1: Two-Phase Commit Protocol

There have been works that try to increase the database availability [2, 3]. But, more recently the distributed systems community has observed a shift in paradigm towards ensuring consistency. A large number of systems are moving towards providing strong consistency guarantees [15, 16, 18, 34, 41, 56]. Such a pragmatic shift has necessitated the use of agreement protocols such as Two-Phase Commit [21]. Commit protocols help in achieving the twin requirements of consistency and reliability in case of partitioned distributed databases. Prior research [9, 18, 42, 54, 56] has shown that data partitioning is an efficient approach to reduce contention and achieve high system throughput. However, a key point in hindsight is that the use of commit protocol should not be a cause for an increase in WAN communication latency in geo-scale distributed applications.

Transaction commit protocols help in reaching an agreement among the participating nodes when a transaction has to be committed or aborted. To initiate an agreement each participating node is asked to vote its decision on the operations on its transactional fragment. The participating nodes can decide to either commit or abort an ongoing transaction. In case of a node failure, the active participants take essential steps (run the termination protocol) to preserve database correctness.

One of the earliest and popular commitment protocol is the *two-phase commit* [21] (henceforth referred as 2PC) protocol. Figure 1 presents the state diagram [39, 52] representation of the 2PC protocol. This figure shows the set of possible states (and transitions) that a coordinating node[1] and the participating nodes follow, in response to a transaction commit request. We use solid lines to represent the state transitions and dotted lines to represent the inputs/outputs to the system. For instance, the coordinator starts the commit protocol on transaction completion, and requests all the participants to commence the same by transmitting *Prepare* messages. In case of multiple failures the two-phase commit protocol has been proved to be blocking [39, 51]. For example, if the coordinator and a participant fail, and if the remaining participants are in the `READY` state, then they cannot make progress (blocked!), as they are unaware about the state of the failed participant. This blocking characteristics

[1]The coordinating node is the one which initiates the commit protocol, and in this work it is also the node which received the client request to execute the transaction.

of the 2PC protocol endangers database availability, and makes it unsuitable for use with the partitioned databases[2]. The inherent shortcomings of the 2PC protocol led towards the design of resilient *three-phase commit* [50, 52] (henceforth referred as 3PC) protocol. The 3PC protocol introduces an additional PRE-COMMIT state between the READY and COMMIT states, which ensures that there is no direct transition between the non-committable and committable states. This simple modification makes the 3PC protocol non-blocking under node failures.

However, the 3PC protocol acts as the major performance suppressant in the design of efficient distributed databases. It can be easily observed that the addition of the PRE-COMMIT state leads to an extra phase of communication among the nodes. This violates the need of an efficient commit protocol for geo-scale systems. Hence, the design of a *hybrid* commit protocol, which leverages the best of both worlds (2PC and 3PC), is in order. We present the *Easy Commit* (a.k.a EC) protocol, which requires two phases of communication, and is non-blocking under node failures. We associate two key insights with the design of Easy Commit protocol that allow us to achieve the non-blocking characteristic in two phases. The first insight is to delay the commitment of updates to the database until the transmission of global decision to all the participating nodes, and the second insight is to induce message redundancy in the network. Easy Commit protocol introduces message redundancy by ensuring that each participating node forwards the global decision to all the other participants (including the coordinator). We now list down our contributions.

- We present the design of a new two-phase commit protocol and show it is non-blocking under node-failures.
- We also present an associated termination protocol, to be initiated by the active nodes, on failure of the coordinating node and/or participating nodes.
- We extend ExpoDB [45] framework to implement the EC protocol. Our implementation can be used seamlessly with various concurrency control algorithms by replacing 2PC protocol with EC protocol.
- We present a detailed evaluation of the EC protocol against the 2PC and 3PC protocol over two different OLTP benchmark suites: YCSB [10] and TPC-C [12], and scale the system upto 64 nodes, on the Microsoft Azure cloud.

The outline for rest of the paper is as follows: in Section 2, we motivate the need for EC protocol. In Section 3, we present design of the EC protocol. In Section 4, we present a discussion on assumptions associated with the design of commit protocols. In Section 5, we present the implementations of various commit protocols. In Section 6, we evaluate the performance of EC protocol against the 2PC and 3PC protocols. In Section 8, we present the related work, and conclude this work in Section 9.

2 MOTIVATION AND BACKGROUND

The state diagram representation for the two-phase commit protocol is presented in Figure 1. In 2PC protocol, the coordinator and participating nodes require at most two transitions to traverse from INITIAL state to the COMMIT or ABORT state. We use figure 3a to present the interaction between the coordinator and the participants, on a linear time scale. The 2PC commit protocol starts with the coordinator node transmitting a *Prepare* message

[2]Partitioned database is the terminology used by the database community to refer to the shared-nothing distributed databases, and should not be intermixed with the term network partitioning.

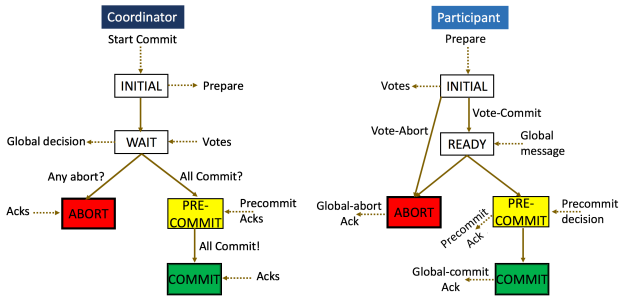
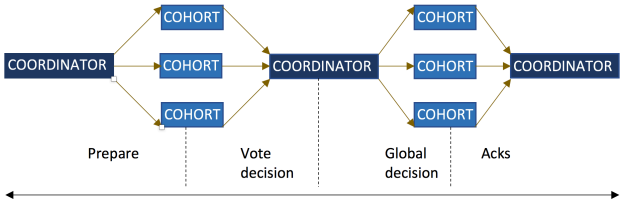
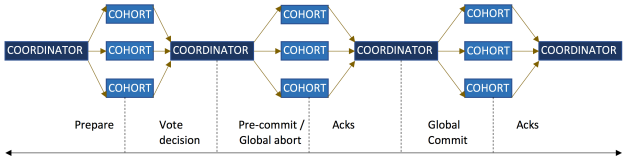


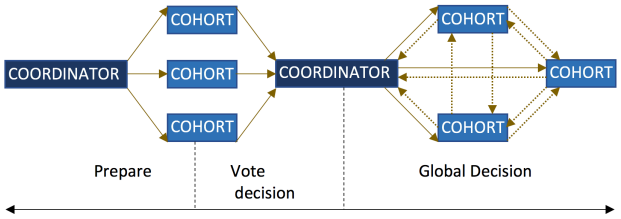
Figure 2: Three-Phase Commit Protocol



(a) Time span of 2PC Protocol



(b) Time span of 3PC Protocol



(c) Time span of EC Protocol

Figure 3: Commit Protocols Linearly Spanned

to each of the cohorts [3] and adding a *begin_commit* entry in its log. When a cohort receives the *Prepare* message, it adds a *ready* entry in its log, sends its decision (*Vote-commit* or *Vote-abort*) to the coordinator. If a cohort decides to abort the transaction then it independently moves to the ABORT state, else it transits to the READY state. The coordinator waits for the decision from all the cohorts. On receiving all the responses, it analyzes all the votes. If there is a *Vote-abort* decision, then the coordinator adds an *abort* entry in the log, transmits the *Global-Abort* message to all the cohorts and moves to the ABORT state. If all the votes are to commit, then the coordinator transmits the *Global-Commit* message to all the cohorts, and moves to COMMIT state, after adding a *commit* entry to log. The cohorts on receiving the coordinator decision move to the ABORT or COMMIT state, and add the *abort* or *commit* entry to the log, respectively. Finally, the cohorts acknowledge the global decision, which allows the coordinator to mark the completion of commit protocol.

[3]The term cohort refers to a participating node in the transaction commit process. We use these terms interchangeably.

The 2PC protocol has been proved to be blocking [39, 51] under multiple node failures. To illustrate this behavior let us consider a simple distributed database system with a coordinator C and three participants X, Y and Z. Now assume a snapshot of the system when C received *Vote-commit* from all the participants, and hence, it decides to send *Global-Commit* message to all the participants. However, say C fails after transmitting *Global-Commit* message to X, but before sending messages to Y and Z. The participant X on receiving the *Global-Commit* message, commits the transaction. Now, assume X fails after committing the transaction. On the other hand, nodes Y and Z would *timeout* due to no response from the coordinator, and would be blocked indefinitely, as they require node X to reach an agreement. They cannot make progress, as neither they have knowledge of the global decision nor they know the state of node X before failure. This situation can be prevented with the help of the three-phase commit protocol [50, 52].

Figure 2 presents the state transition diagram for the coordinator and cohort executing the three-phase commit protocol, while figure 3b expands the 3PC protocol on the linear time scale. In the first phase, the coordinator and the cohorts, perform the same set of actions as in the 2PC protocol. Once the coordinator checks all the votes, it decides whether to abort or commit the transaction. If the decision is to abort, the remaining set of actions performed by the coordinator (and the cohorts) are similar to the 2PC protocol. However, if the coordinator decides to commit the transaction, then it first transmits a *Prepare-to-Commit* message, and adds a *pre-commit* entry to the log. The cohorts on receiving the *Prepare-to-Commit* message, move to the `PRE-COMMIT` state, add a corresponding *pre-commit* entry to the log, and acknowledge the message reception to the coordinator. The coordinator then sends a *Global-Commit* message to all the cohorts, and the remaining set of actions are similar to the 2PC protocol.

The key difference between the 2PC and 3PC protocol is the `PRE-COMMIT` state, which makes the latter non-blocking. The design of 3PC protocol is based on the Skeen's [50] design of a non-blocking commit. In his work Skeen laid down two fundamental properties for the design of a non-blocking commit protocol: (i) no state should be adjacent to both the `ABORT` and `COMMIT` states, and (ii) no non-committable[4] state should be adjacent to the `COMMIT` state. These requirements motivated Skeen to introduce the notion of a new committable state (`PRE-COMMIT`) to the 2PC state transition diagram.

The existence of `PRE-COMMIT` state makes the 3PC protocol non-blocking. The aforementioned multi-node failure case does not indefinitely block the nodes Y and Z which are waiting in the `READY` state. The nodes Y and Z can make safe progress (by aborting the transaction) as they are assured that the node X could not have committed the transaction. Such a behavior is implied by the principle that no two nodes could be more than one state transition apart. The node X is guaranteed to be in one of the following states: `INITAL`, `READY`, `PRE-COMMIT` and `ABORT`, at the time of failure. This indicates that node X could not have committed the transaction, as nodes Y and Z are still in the `READY` state (It is important note that in the 3PC protocol the coordinator sends the *Global-Commit* message after it transmits the *Prepare-to-commit* message to all the nodes.). Interestingly, if either of nodes Y or Z are in the `PRE-COMMIT` state then they can actually commit the transaction. However, it can be easily observed that the non-blocking characteristic of the 3PC protocol comes at an additional cost, an extra round of handshaking.

[4] `INITAL`, `READY` and `WAIT` states are considered as non-committable states.

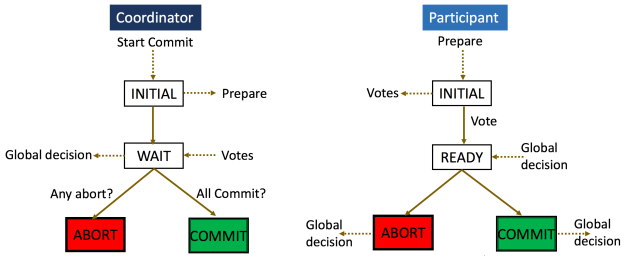


Figure 4: Easy Commit Protocol

3 EASY COMMIT

We now present the *Easy Commit* (*EC*) protocol. EC is a two-phase protocol, but unlike 2PC it exhibits non-blocking behavior. The EC protocol achieves these goals through two key insights: (i) first transmit and then commit, and (ii) message redundancy. In the second phase, Easy Commit ensures that each participating node forwards the coordinating node's decision to all the other participants. To ensure non-blocking behavior, EC protocol also requires each node (coordinator and participants) to transmit the global decision to all the other nodes, before they commit. Hence, the commit step subsumes message transmission to all the nodes.

3.1 Commitment Protocol

We present the EC protocol state transition diagram, and the coordinator and participant algorithms in Figure 4 and Figure 5, respectively. The EC protocol is initiated by the coordinator node. It sends the *Prepare* message to each of the cohorts and moves to the `READY` state. When a cohort receives the *Prepare* message, it sends its decision to the coordinator, and moves to the `READY` state. On receiving the responses from each of the cohorts, the coordinator first transmits the global decision to all the participants, and then commits (or aborts) the transaction. Each of the cohorts, on receiving a response from the coordinator, first forward the global decision to all the participants (and the coordinator), and then commit (or abort) the transaction locally.

We introduce multiple entries to the log to facilitate recovery during node failures. Note: the EC protocol allows the coordinator to commit as soon as it has communicated the global decision to all the other nodes. This implies that the coordinator need not wait for the acknowledgments. When a node `timeouts`, while waiting for a message, it executes the *termination protocol*. Some of the noteworthy observations are:

I. A participant node cannot make a direct transition from the `INITIAL` state to the `ABORT` state.
II. The cohorts, irrespective of the global decision, always forward it to every participant.
III. The cohorts need not wait for message from the coordinator, if they receive global decision from other participants.
IV. There exists some hidden states (a.k.a `TRANSMIT-A` and `TRANSMIT-C`), only after which a node aborts or commits the transaction (cf. discussed in Section 3.2).

In Figure 3c, we also present the linear time scale model for the Easy Commit protocol. Here, in the second phase, we use solid lines to represent the global decision from the coordinator to the cohorts, and the dotted lines to represent message forwarding.

Send *Prepare* to all participants;
Add *begin_commit* to log;
Wait for (*Vote-commit* or *Vote-abort*) from all participants;
if `timeout` **then**
  Run **Termination Protocol**;
end if
if All messages are *Vote-commit* **then**
  Add *global-commit-decision-reached* in log;
  Send *Global-commit* to all participants;
  `Commit` the transaction;
  Add *transaction-commit* to log;
else
  Add *global-abort-decision-reached* in log;
  Send *Global-abort* to all participants;
  `Abort` the transaction;
  Add *transaction-abort* to log;
end if

(a) Coordinator's algorithm

Wait for *Prepare* from the coordinator;
if `timeout` **then**
  Run **Termination Protocol**;
end if
Send decision (*Vote-commit* or *Vote-abort*) to coordinator;
Add *ready* to log;
Wait for message from coordinator;
if `timeout` **then**
  Run **Termination Protocol**;
end if
if Coordinator decision is *Global-commit* **then**
  Add *global-commit-received* in log;
  Forward *Global-commit* to all nodes;
  `Commit` the transaction;
  Add *transaction-commit* to log;
else
  Add *global-abort-received* in log;
  Forward *Global-abort* to all nodes;
  `Abort` the transaction;
  Add *transaction-abort* to log;
end if

(b) Participant's algorithm

Figure 5: Easy Commit Algorithm.

3.2 Termination Protocol

We now consider the correctness of the EC algorithm under *node-failures*. We want to ensure that the EC protocol exhibits both liveness and safety properties. A commit protocol is said to be *safe* if there isn't any instant during the execution of the system under consideration when two or more nodes are in conflicting states (that is one node is in `COMMIT` state while other is in `ABORT`). A protocol is said to respect *liveness* if its execution causes none of the nodes to block.

During the execution of a commit protocol, each node waits for a message for a specific amount of time before it *timeouts*. When a node timeouts then it concludes loss of communication with the sender node, which in our case corresponds to failure of the sender node. A node is assumed to be blocked if it is unable to make progress on timeout. In case of such node failures, the active nodes execute the termination protocol to ensure system

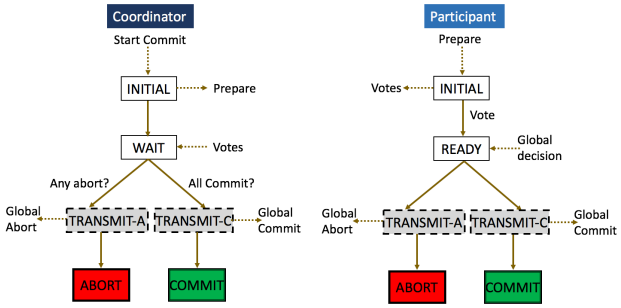


Figure 6: Logical expansion of Easy Commit Protocol.

makes progress. We illustrate the termination protocol by stating the actions taken by the coordinator and the participating nodes on timeout. The coordinator can timeout only in the `WAIT` state, while the cohorts can timeout in `INITIAL` and `READY` states.

A. **Coordinator Timeout in `WAIT` state** – If the coordinator timeouts in this state, then it implies that the coordinator didn't receive the vote from one of the cohorts. Hence, the coordinator first adds a log entry for the *global-abort-decision-reached*, then transmits the *Global-abort* message to all the active participants, and finally aborts the transaction.
B. **Cohort Timeout in `INITIAL` State** – If the cohort timeouts in this state, then it implies that it didn't receive the *Prepare* message from the coordinator. Hence, this cohort communicates with other active cohorts to reach a common decision.
C. **Cohort Timeout in `READY` State** – If the cohort timeouts in this state, it implies that it didn't receive a *Global-Commit* (or *Global-Abort*) message from any node. Hence, it would consult the active participants to reach a decision common to all the participants.

Leader Election: In last two cases we force the cohorts to perform transactional commit or abort based on an agreement. This agreement requires selection of a new leader (or coordinator). The target of this leader is to ensure that all the active participants, follow the same decision that is commit (or abort) the transaction. The selected leader can be in the `INITIAL` or the `WAIT` state. It consults all the nodes if any of them has received a copy of the global decision. If none of the nodes know the global decision, then the leader first adds a log entry for the *global-abort-decision-reached*, then transmits the *Global-abort* message to all the active participants, and finally aborts the transaction.

To prove correctness of EC protocol, Figure 6 expands the state transition diagram. We introduces two intermediate hidden states (a.k.a `TRANSMIT-A` and `TRANSMIT-C`). All the nodes are oblivious to these states, and the purpose of these states is to ensure message redundancy in the network. As a consequence, we can categorize the states of the EC protocol under five heads:

- `UNDECIDED` – The state before reception of global decision (that is `INITIAL`, `READY` and `WAIT` states).
- `TRANSMIT-A` – The state on receiving the global abort.
- `TRANSMIT-C` – The state on receiving the global commit.
- `ABORT` – The state after transmitting *Global-Abort*.
- `COMMIT` – The state after transmitting *Global-Commit*.

Figure 7 illustrate whether two states can co-exist (Y) or they conflict (N). We derive this table on the basis of our observations: I - IV and cases A - C. We now have sufficient tools to prove the liveness and safety property of EC protocol.

	UNDECIDED	T-A	T-C	ABORT	COMMIT
UNDECIDED	Y	Y	Y	N	N
T-A	Y	Y	N	Y	N
T-C	Y	N	Y	N	Y
ABORT	N	Y	N	Y	N
COMMIT	N	N	Y	N	Y

Figure 7: Coexistent states in EC protocol (T-A refers to TRANSMIT-A and T-C refers to TRANSMIT-C).

Theorem 3.1. *Easy Commit protocol is safe, that is in the presence of only node failures, for a specific transaction, two nodes cannot be in both Aborted and Committed states, at any instant.*

Proof. Let us assume the case that two nodes p and q are in the conflicting states (say p voted to abort the transaction and q voted to commit). This would imply that one of them received *Global-Commit* message while the other received *Global-Abort*. From (II) and (III) we can deduce that p and q should transmit the global decision to each other, but as they are in different states, it implies a contradiction. Also, from (I) we have the guarantee that p could not have directly transited to the ABORT state. This implies p and q would have received message from some other node. But, then they should have received the same global decision.

Hence, we assume that either of the nodes p or q moved to a conflicting state and then failed. But, this violates property (IV) which states that a node needs to transmit its decision to all the other nodes before it can commit or abort the transaction. Also, once either of p or q fails, the rest of the system follows termination protocol (cases (A) to (C)), and reaches a safe state. It is important to see that the termination protocol is re-entrant. □

Theorem 3.2. *Easy Commit protocol is live that is in the presence of only node failures, it does not block.*

Proof. The proof for this theorem is a corollary of Theorem 3.1. The termination protocol cases (A) to (C) provide the guarantee that the nodes do not block and can make progress, in case of a node failure. □

3.3 Comparison with 2PC Protocol

We now draw out comparisons between the the 2PC and EC protocols. Although, EC protocol is non-blocking, but still 2PC protocol has a lower message complexity. EC protocol's message complexity is $O(n^2)$, while the message complexity for 2PC $O(n)$.

To illustrate the non-blocking property of EC protocol, we now tackle the motivational example of multiple failures. For the sake of completeness we restate the example here. Let us assume a distributed system with coordinator C and participants X, Y and Z. We also assume that C decides to transmit *Global-Commit* message to all the nodes, and fails just after transmitting message to the participant X. Say, the node X also fails after receiving the message from C. Thus, nodes Y and Z neither received messages from C nor from node X. In this setting, the nodes Y and Z would eventually timeout, and run the termination protocol. From case (C) of termination protocol, it is evident that the nodes Y and Z would select a new leader among themselves, and would safely transit to the ABORT state.

3.4 Comparison with 3PC protocol

Although, EC protocol looks similar to 3PC protocol, but it is a stricter and an efficient variant to 3PC protocol. It introduces the notion of a set of intermediate hidden states: TRANSMIT-A and TRANSMIT-C, which can be superimposed on the ABORT and COMMIT states, respectively. Also, in the EC protocol, the nodes do not expect any acknowledgements. So unlike the 3PC protocol, there are no inputs to the TRANSMIT-A, TRANSMIT-C, ABORT and COMMIT states. However, EC protocol has a higher message complexity than 3PC, which has a message complexity of $O(n)$.

4 DISCUSSION

Until now, all our discussion assumed existence of only node failures. In Section 3 we prove that EC protocol is non-blocking under node failures. We now discuss the behavior of the 2PC, 3PC and EC protocols under communication failures that is message delay and message loss. Later in this section we also study the degree to which these protocols support independent recovery.

4.1 Message Delay and Loss

We now analyze the characteristics of 2PC, 3PC and EC protocol, under unexpected delays in message transmission. Message delays represent an unprecedented lag in the communication network. The presence of message delays can cause a node to timeout and act as if a node failure has occurred. This node may receive a message pertaining transaction commitment or abort, after the decision has been made. It is interesting to note that 2PC and 3PC protocols are not safe under message delays [7, 39]. Prior works [26, 39] have shown that it is impossible to design a non-blocking commitment protocol for unbounded asynchronous networks with even a single failure.

We illustrate the nature of 3PC protocol under message delay, as it is trivial to show that 2PC protocol is unsafe under message delays. The 3PC protocol state diagram does not provide any intuition about the transitions that two nodes should perform when both of them are active but unable to communicate. In fact, partial communication or unprecedented delay in communication can easily hamper the database consistency.

Let us consider a simple configuration with a coordinator C and the participants X, Y and Z. Now assume that C receives *Vote-commit* message from all the cohorts, and, hence it decides to send the *Prepare-to-Commit* message to all the cohorts. However, it is possible that the system starts facing unanticipated delays on all the communication links with C at one end. We can also assume that the paths to node X are also facing severe delays. In such a situation, the coordinator would proceed to *globally commit* the transaction (as it has moved to the PRE-COMMIT state), while the nodes X, Y and Z would abort the transaction (as from their perspective the system has undergone multiple failures). This implies that the 3PC termination protocol is not sound under message delays, and similarly we can show that EC protocol is unsafe under message delays.

This situation can aggravate if the network undergoes message loss. Interestingly, message loss has been deemed to be true representation of the network partitioning [39]. Hence, no commit protocol is safe (or non-blocking) under message loss [7]. If the system is suffering from message loss then the participating nodes (and coordinator) would *timeout*, and would run the associated terminating protocol that could make nodes transit to conflicting states. Thus, we also conclude that 2PC, 3PC and EC protocols are unsafe under message loss.

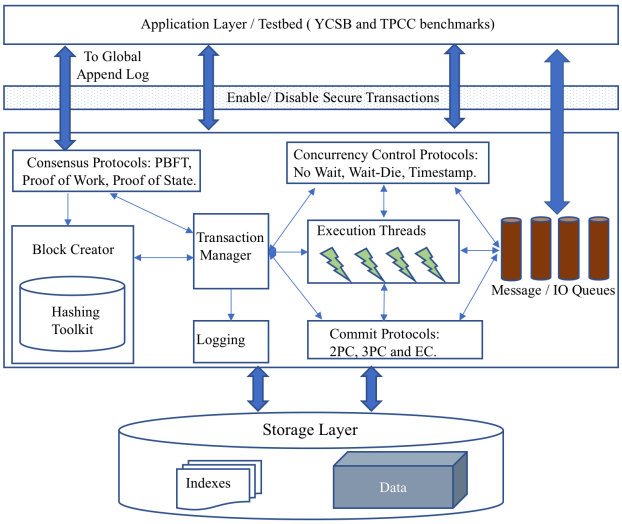


Figure 8: ExpoDB Framework - executed at each server process, hosted on a cloud node. Each server process receives a set of messages (from clients and other servers), and uses multiple threads to interact with various distributed database components.

4.2 Independent Recovery

Independent recovery is one of the desired properties from the nodes in a distributed system. An independent recovery protocol lays down a set of rules that help a failed node to terminate (commit or abort) the transaction which was executing at the time of its failure, without any help from other active participants. It is interesting to note that 2PC and 3PC protocols only support partial independent recovery [7, 39].

It is easy to present a case where the 3PC protocol lacks independent recovery. Consider a cohort in the `READY` state that votes to commit the transaction, and fails. On recovery this node needs to consult with the other nodes about the fate of the last transaction. This node cannot independently commit (or abort) the transaction, as it does not know the global decision, which could have been either commit or abort.

EC protocol supports independent recovery in following cases:

(i) If a cohort fails before transmitting its vote, then on recovery it can simply abort the transaction.
(ii) If the coordinator fails before transmitting the global decision then it aborts the transaction on recovery.
(iii) If either coordinator or participant fail after transmitting the global decision and writing the log, then on recovery they can use this entry to reach the consistent state.

5 EASY COMMIT IMPLEMENTATION

We now present a discussion on our implementation of the Easy Commit (EC) protocol. We have implemented EC protocol in the ExpoDB platform [45]. ExpoDB is an in-memory, distributed transactional platform that incorporates and extends the Deneva [28] testbed. ExpoDB also offers secure transactional capability, and presents a flexible framework to study distributed ledger—blockchain [8, 43].

5.1 Architectural Overview

ExpoDB includes a lightweight layer for testing distributed protocols and design strategy. Figure 8 presents the block diagram representation of the ExpoDB framework. It supports a client-server architecture, where each client or server process is hosted on one of the cloud nodes. To maintain inherent characteristics of a distributed system, we opt for a shared nothing architecture. Each partition is mapped to one server node.

A transaction is expressed as a stored procedure that contains both program logic and database queries, which read or modify the records. The clients and server processes communicate with each other using TCP/IP sockets. In practice, the client and server processes are hosted on different cloud nodes, and we maintain an equal number of client and server cloud instances.

Each client creates one or more transactions, and sends the transaction execution request to the server process. The server process in turn executes the transaction by accessing the local data and runs the transaction until further execution requires access to remote data. The server process then communicates with other server processes that have access to remote data (remote partitions), to continue the execution. Once, these processes return the result, the server process continues execution till completion. Next, it takes a decision to commit or abort the transaction (that is, executes the associated *commit protocol*).

In case a transaction has to be aborted then the coordinating server sends messages to the remote servers to rollback the changes. Such a transaction is resumed after an exponential back-off time. On successful completion of a transaction, the coordinating server process sends an acknowledgment to the client process, and performs necessary garbage collection.

5.2 Design of 2PC and 3PC

2PC: The 2PC protocol starts after the completion of the transaction execution. The read-only transactions and single partition transactions do not make use of the commit protocol. Hence, the commit protocol comes into play when the transaction is multi-partition and performs updates to the data-storage. The coordinating server sends a *Prepare* message to all the participating servers, and waits for their response. The participating servers respond with the *Vote-commit* message[5]. On receiving the *Vote-commit* message the coordinating server starts the final phase, and transmits the *Global-Commit* message to all the participants. Each participant on receiving the *Global-Commit* message commits the transaction, releases the local transactional resources, and responds with an acknowledgment for the coordinator. The coordinator waits on a counter for response from each participant and then commits the transaction, sends a response to the client node, and releases the associated transactional data-structures.

3PC: To gauge the performance of the EC protocol, we also implemented the three-phase commit protocol. The 3PC protocol implementation is a straightforward extension to the 2PC protocol. We add an extra `PRE-COMMIT` phase before the final phase. On receiving, all the *Vote-commit* messages, the coordinator sends the *Precommit* message to each participant. The participating nodes acknowledge the reception of the *Precommit* message from the coordinator. The coordinating server on receiving these acknowledgments, starts the finish phase.

5.3 Easy Commit Design

We now explain the design of Easy Commit protocol in the ExpoDB framework. The first phase (that is the `INITIAL` phase) is same for both the 2PC and the EC protocol. In the EC protocol,

[5]Without node failures, any transaction that reaches the prepare phase is assumed to successfully commit.

once the coordinator receives the *Vote-commit* message from all the nodes, it first sends the *Global-commit* message to each of the participating processes, and then commits the transaction. Next it, responds to the client with the transaction completion notification. When the participating nodes receive the *Global-Commit* message from the coordinator, they forward the *Global-Commit* message to all the other nodes (including the coordinator), and then commit the transaction.

Although, in the EC protocol the coordinator has a faster response rate to the client, but its throughput takes a slight dip due to additional, implementation enforced wait. It can be noted that we have not performed any cleanup tasks (such as releasing the transactional resources) yet. The cleanup of the transactional resources is performed once it is ensured that neither of those resources would be ever used, nor any messages associated with the transaction would be further received. Hence, we have to force all the nodes (both the coordinator and the participants) to poll the message queue, and wait till they have received the messages from each other node. Once all the messages are received, each node performs the cleanup.

To implement EC protocol we had to extend the message being transmitted with a new field which identifies all the participants of the transaction. This array contains the `Id` for each participant, and is updated by the coordinator (as only the coordinator has information about all the partitions) and transmitted as part of the *Global-Commit* message.

6 EVAUATION

In this section, we present a comprehensive evaluation of our novel Easy Commit protocol against 2PC and 3PC. As discussed in Section 5, we use the ExpoDB framework for implementing the EC protocol. For our experimentation, we adopt the evaluation scheme of Harding et al. [28].

To evaluate various commit protocols, we deploy the ExpoDB framework on the Microsoft Azure cloud. For running the client and server processes, we use upto 64 `Standard_D8S_V3` instances, deployed in the US East region. Each `Standard_D8S_V3` instance consists of 8 virtual CPU cores and 32GB of memory. For our experiments, we ensure a one-to-one mapping between the server (or client) process and the hosting `Standard_D8S_V3` instance. On each server process, we allowed creation of 4 worker threads, each of which were attached to a dedicated core, and 8 I/O threads. At each server node, a load of 10000 open client connections is applied. For each experiment, we first initiated a warmup phase for 60 seconds, followed by 60 seconds of execution. The measured throughput does not include the transactions completed during warmup phase. If a transaction gets aborted then it is restarted again, only after a fixed time. To attenuate the noise in our readings, we average our results over three runs.

To evaluate the commit protocols, we use the `NO_WAIT` concurrency control algorithm. We use the `NO_WAIT` algorithm as: (i) it is the simplest algorithm, amongst all the concurrency control algorithms present in the ExpoDB framework, and (ii) has been proved to achieve high system throughput. It has to be noted that the use of underlying concurrency control algorithm is orthogonal to our approach. We present the design of a new commit protocol, and hence other concurrency control algorithms (except Calvin) available in the ExpoDB framework, can also employ EC protocol during the commit phase. We present a discussion on the different concurrency control algorithms, later in this section.

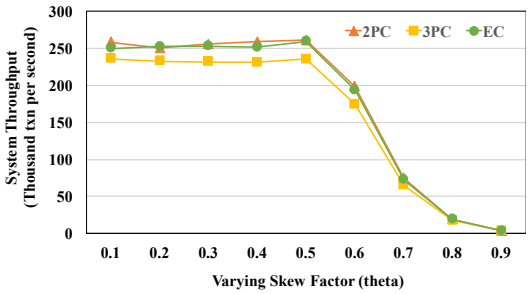


Figure 9: System throughput (transactions per second) on varying the skew factor (`theta`) for the 2PC, 3PC and EC protocols. These experiments run the YCSB benchmark. Number of server nodes are set to 16 and partitions per transaction are set to 2.

In `NO_WAIT` protocol, a transaction requesting access to a locked record is aborted. On aborting the transaction, all the locks held with this transaction are released, which allows other transactions waiting on these locks to progress. `NO_WAIT` algorithm prevents deadlock by aborting transactions in case of conflicts, and hence, has high abort rate. The simple design of `NO_WAIT` algorithm, and its ability to achieve high system throughput [28] motivated us to use it for concurrency control.

6.1 Benchmark Workloads

We test our experiments on two different benchmark suites: YCSB [10] and TPC-C [12]. We use YCSB benchmark to evaluate EC protocol on characteristics interesting to the OLTP database designers (Section 6.2 to Section 6.5), and use TPC-C to gauge the performance of EC protocol from the perspective of a real world application (Section 6.6 and Section 6.7).

YCSB – The Yahoo! Cloud Serving Benchmark consists of 11 columns (including a primary key) and 100B random characters. In our experiments we used a YCSB table of size 16 million records per partition. Hence, the size of our database was 16 GB per node. For all our experiments we ensured that each YCSB transaction accessed 10 records (we mention changes to this scheme explicitly). Each access to YCSB data followed the Zipfian distribution. Zipfian distribution tunes the access to hot records through the *skew factor* (`theta`). When `theta` is set to 0.1, the resulting distribution is uniform, while the `theta` value 0.9 corresponds to extremely skewed distribution. In our evaluation using YCSB data, we only executed multi-partition transactions, as single partition transactions do not require use of commit algorithms.

TPC-C – The TPC-C benchmark helps to evaluate system performance by modeling an application for warehouse order processing. It consists of a read-only, item table that is replicated at each server node while rest of the tables are partitioned using the warehouse ID. ExpoDB supports *Payment* and *NewOrder* transactions, which constitute 88% of the workload. Each transaction of *Payment* type accesses at most 2 partitions. These transaction first update the payment amounts for the local warehouse and district, and then update the customer data. The probability that a customer belongs to a remote warehouse is 0.15. In case of transactions of type *NewOrder*, first the transaction reads the local warehouse and district records and then modifies the district record. Next, it modifies item entries in the stock table. Only, 10% *NewOrder* transactions are multi-partition , as only 1% of the updates require remote access.

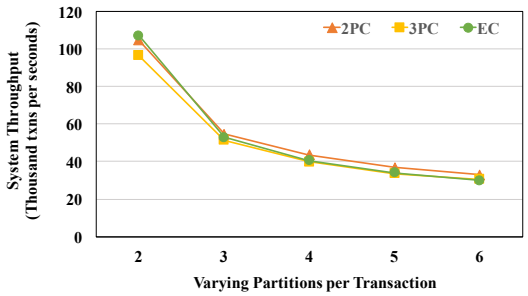


Figure 10: System throughput (transactions per second) on varying the number of partitions per transactions for the commit protocols. These experiments use YCSB benchmark. The number of server nodes are set to 16 and `theta` is set to 0.6.

6.2 Varying Skew factor (Theta)

We evaluate the system throughput by tuning the skew factor (theta), available in YCSB benchmarks, from 0.1 to 0.9. Figure 9 presents the statistics when the number of partitions per transaction are set to 2. In this experiment, we use 16 server nodes to analyze the effects induced by the three commit protocols.

A key takeaway from this plot is that, for `theta` $\leq$ 0.7 the system throughputs for EC and 2PC protocols are better than the system throughput for the 3PC protocol. On increasing the theta further the transactional access becomes highly skewed. This results in an increased contention between the transactions as they try to access (read or write) the same record. Hence, there is a significant reduction in the system throughput across various commit protocols. Thus, it can be observed that the magnitude of difference in the system throughputs for 2PC, 3PC and EC protocol is relatively insignificant. It is important to note that on highly skewed data, the gains due to the choice of underlying commit protocols are overshadowed by other system overheads (such as cleanup, transaction management and so on).

In the YCSB benchmark, for `theta` $\leq$ 0.5 the data access is uniform across the nodes, which implies that the client transactions access data on various partitions – low contention. Hence, each server node achieves nearly the same throughput. It can be observed that for all the three commit protocols the throughput is nearly constant (not same). We attribute the delta difference in the throughputs of the EC and 2PC protocols to the system induced overheads, network communication latency, and resource contention between the threads (for access to cpu and cache).

6.3 Varying Partitions per Transaction

We now measure the system throughput achieved by the three commit protocols on varying the number of partitions per transactions from 2 to 6. Figure 10 presents the throughput achieved on the YCSB benchmark, when `theta` is fixed to 0.6, and number of server nodes are set to 16. The number of operations accessed by each transaction are set to 16, and the transaction read-write ratio is maintained at 1 : 1.

It can be observed that on increasing the number of partitions per transaction there is a dip in the system throughput, across all of the commit protocols. On moving from 2 to 4 partitions there is an approximate decrease of 55%, while the reduction is system performance is around 25% from 4 partitions to 6 partitions, for the three commit protocol. As the number of partitions per transaction increase, the number of messages being exchanged in each round increases linearly for 2PC and 3PC, and quadratically for EC. Also, an increase in partitions imply the transactional resources are held longer across multiple sites, which leads to throughput degradation for all the protocols. Note: in practice, the number of partitions per transaction are not more than four [12].

6.4 Varying Server Nodes

We study the effect of varying the number of server nodes (from 2 nodes to 32 nodes) on the system throughput and latency, for the 2PC, 3PC and EC protocols. In Figure 11 we set the number of partitions per transaction to 2 and plot graphs for the low contention (`theta` = 0.1), medium contention (`theta` = 0.6) and high contention (`theta` = 0.7). In these experiments, we increase size of YCSB table in accordance to the the increase in number of server nodes.

In Figure 11, we use the plots on the left to study the system throughput on varying the number of server nodes. It can be observed that as the contention (or skew factor) increases the system throughput decreases, and such a reduction is sharply evident on moving from `theta` = 0.6 to `theta` = 0.7. Another interesting observation is that the system throughput attained by the EC protocol is significantly greater than the throughput attained under 3PC protocol. The gains in system throughput are due to reduction of an extra phase which compensates for the extra messages communicated during the EC protocol.

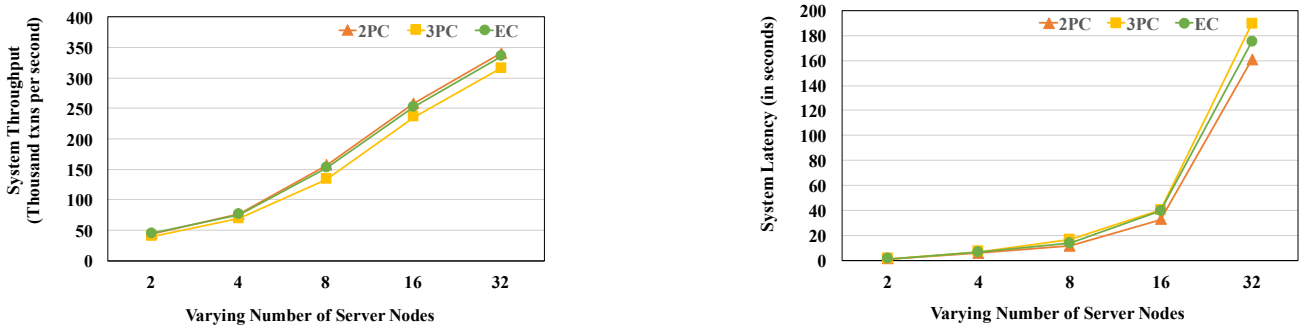
In comparison to the 2PC protocol the system throughput under EC protocol is marginally lower at low contention and medium contention, and relatively same at high contention. These gains are the result of zero acknowledgment messages required by the coordinating node, in the commit phase, which helps EC protocol perform nearly as efficient as the 2PC protocol. This helps us to conclude that a database system using EC is as scalable as its counterpart employing 2PC.

6.4.1 Latency. In Figure 11, we use the plots on the right, to shows the 99 percentile system latency when one of the three commit protocols are employed by the system. We again vary the number of server nodes from 2 to 32. The 99 percentile latency is measured from the first commit to the final commit of a transaction. On increasing the number of server nodes there is a steep increase in latency for each commit protocol. The high latency values for 3PC protocol can be easily cited to the extra phase of communication.

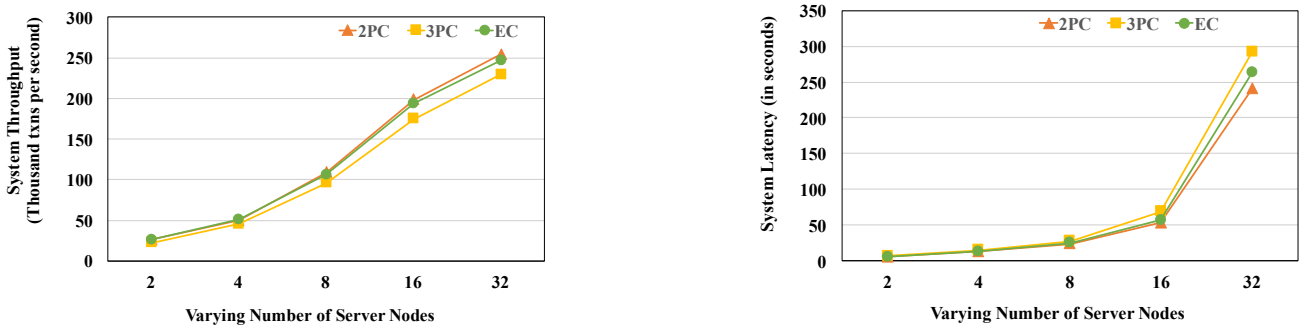
6.4.2 Proportion of time consumed by various components: Figure 12 presents the time spent on various components of the distributed database system. We show the time distribution for the different degree of contention (*theta*). We categorize these measures under seven different heads.

Useful Work is the time spent by worker threads doing computation for read and write operations. **Txn Manager** is the time spent in maintaining transaction associated resources. **Index** is the time spent in transaction indexing. **Abort** is the time spent in cleaning up aborted transactions. **Idle** is the time worker thread spends when not performing any task. **Commit** is the time spent in executing the commit protocol. **Overhead** represents the time to fetch transaction table, transaction cleanup and releasing transaction table.

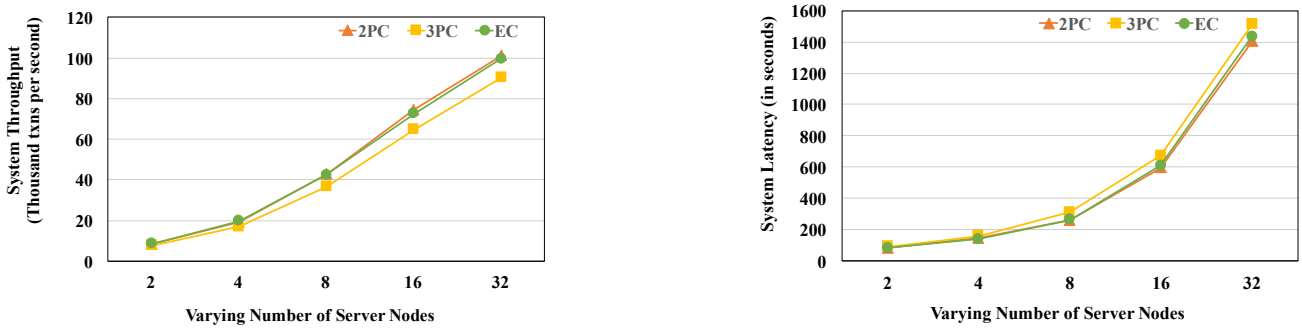
The key intuition from these plots is that as the contention (`theta`) increases there is an increase in time spent in abort. At low contention as most of the transactions are read-only, so the time spent in commit phase is least, and as contention increase, commit phase plays an important role in achieving high throughput from databases. Also, it can be observed at medium and high contention, worker threads executing 3PC protocol



(a) Low contention – (`theta = 0.1`).



(b) Medium contention – (`theta = 0.6`).



(c) High contention – (`theta = 0.7`).

Figure 11: System Throughput (transactions per second) and System Latency (in seconds), on varying the number of server nodes for the 2PC, 3PC and EC protocols. The measured latency is the 99-percentile latency, that is, latency from the first start to final commit of a transaction. For these experiments we use the YCSB benchmarks and set the number of partitions per transaction to 2.

are idle for the maximum time and perform the least amount of useful work, which indicates a decrease in system throughput under 3PC protocol due to an extra phase of communication.

6.5 Varying Transaction Write Percentage

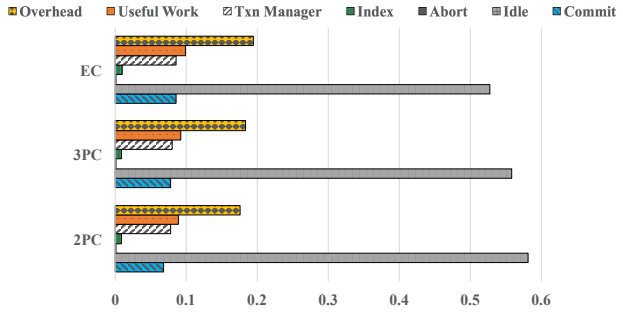
We now vary the transactional write percentage, and draw out comparisons between the system throughput achieved by the ExpoDB when employing one of the three commit protocols. These experiments are based on YCSB benchmark, and vary the percentage of write operations accessed by each transaction from 10 to 90. We set the skew factor to 0.6, number of server nodes to 16 and partitions per transaction to 2.

It can be seen that when only 10% of the operations are write then all the protocols achieve nearly the same system throughput. This is because most of the requests sent by the client consists of read-only transactions, and under read only transactions, the commit protocols are not executed. However, as the write percentage increases the gap between the system throughput achieved by 3PC protocol and the other two commit protocols increases. This indicates that 3PC protocol performs poorly when the underlying application consists of write intensive transactions.

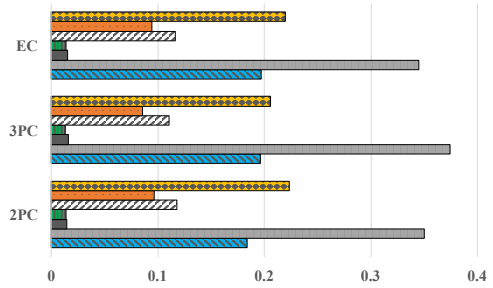
In comparison to the 2PC protocol, EC protocol undergoes marginal reduction in throughput. As the number of write operations increase, the number of transactions undergoing the commit protocol also increase. We have already seen that under EC protocol (i) the amount of message communication is higher than the 2PC protocol, and (ii) each node needs to wait for additional *wait-time* before releasing the transactional resources. Some of these held resources include locks on data items, and it is easy to surmise that under EC protocol locks are held longer than the 2PC protocol. The increase in duration of locks being held also leads to an increased abort rate, which is another important factor for reduced system throughput.

6.6 Scalability of TPC-C Benchmarks

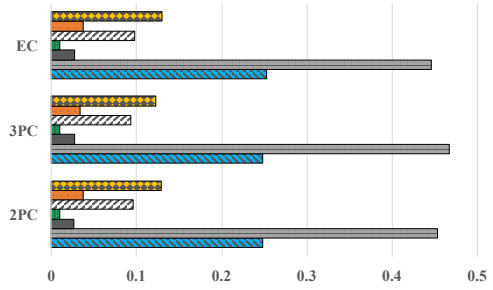
We now gauge the performance of the EC protocol with respect to a real-world application, that is using TPC-C benchmark. Figure 14 presents the characteristics of the 2PC, 3PC and EC protocols, under TPC-C benchmark, on varying the number of server nodes. It has to be noted that a major chunk of TPC-C transactions are single-partition, while most of the multi-partition transactions access only two partitions. Our evaluation scheme



(a) Low contentions – (theta = 0.1).



(b) Medium contentions – (theta = 0.6).



(c) High contentions – (theta = 0.7).

Figure 12: Percentage of time spent by various database components, on executing the YCSB benchmark. We set the number of server nodes to 16 and partitions per transaction to 2.

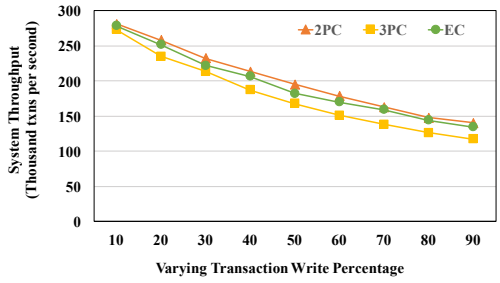
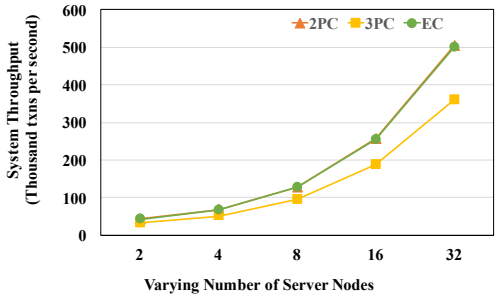


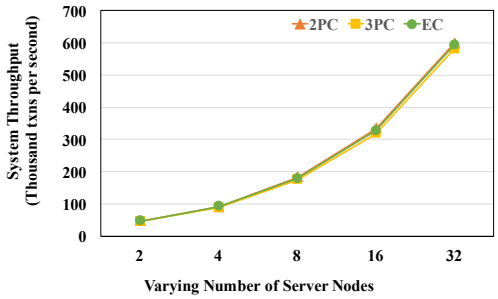
Figure 13: System throughput (transactions per second) on varying the transaction write percentage for the 2PC, 3PC and EC protocols. These experiments use YCSB benchmark, and set the number of server nodes to 16 and partitions per transactions to 2.

sets 128 warehouses per server, and, hence a multi-partition can access two co-located partitions (that is on a single server).

Figure 14a represents the scalability of the *Payment* transactions for the three commit protocols. It is evident from this plot that as the number of server nodes increase, the system throughput increases for each commit protocol. However, there



(a) Payment Transaction



(b) NewOrder Transaction

Figure 14: System throughput on varying the number of server nodes, on the TPC-C benchmark. The number of warehouses per server are set to 128.

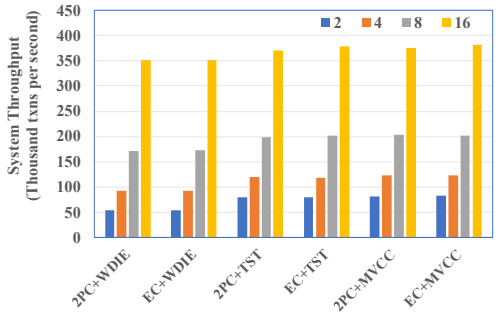


Figure 15: System throughput achieved by three different concurrency control algorithms. For experimentation, we use the TPC-C *Payment* transaction, and vary the number of server nodes to 16. The number of warehouses per server are set to 128. Here WDIE and TST refer to WAIT-DIE and TIMESTAMP, respectively.

is a performance bottleneck in case of 3PC protocol. In case of payment transactions as updates are performed at the home warehouse, which requires exclusive access, so there is an increase in abort rate for the underlying concurrency control algorithm (in our case NO_WAIT). Now, as 3PC protocol requires an additional phase to commit the transaction, hence there is an increase in the abort rate. Interestingly, the throughput achieved by the EC protocol is approximately equal to the system throughput under 2PC protocol.

Figure 14b depicts the system throughput on executing TPC-C *NewOrder* transactions. The performance bottleneck is reduced for these transactions as there only 10 districts per warehouse, and hence, the commit protocols achieve comparatively higher throughput. Also, as there are only 10% multi-partition transactions, so all the protocols achieve nearly the same performance.

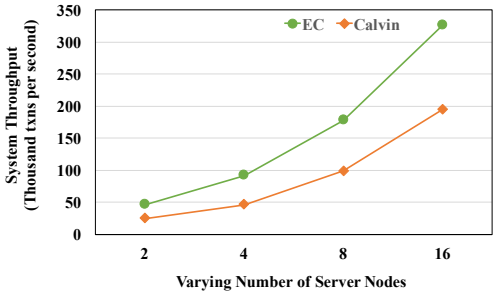


Figure 16: Comparison of throughput achieved by the system executing Calvin versus the system implementing the combination of No-Wait+EC protocol. In this experiment we use the TPC-C *Neworder* transaction, and vary the number of server nodes to 16. The number of warehouses per server are set to 128.

6.7 Concurrency Control

The presence of read/write data conflicts between transactional accesses necessitates the use of concurrency control algorithms by the database management system. The ExpoDB framework implements multiple state-of-the-art concurrency control algorithms. Although, in this work, we use NO_WAIT concurrency control algorithm, but EC protocol can be easily integrated to work alongside other concurrency control algorithms.

Figure 15 measures the system throughput for three different concurrency control algorithms. We use TPC-C *Payment* transactions for these experiments, and increase the number of server nodes upto 16. We also set the number of warehouses per server to 128. We compare the performance of EC protocol against the 2PC protocol, when the underlying concurrency control algorithm is WAIT-DIE [4], TIMESTAMP [4] and MVCC [5]. It is evident from these experiments that the EC protocol is able to achieve as high efficiency as the 2PC protocol, irrespective of the mechanism used for ensuring concurrency control.

We also analyze our commit protocol against an interesting deterministic concurrency control algorithm – *Calvin* [56]. *Calvin* is a deterministic algorithm that requires the prior knowledge of the read/write sets of the transaction before its execution. When the transaction's read/write sets are not known, at prior, then *Calvin* causes some transactions to execute twice. Interestingly, in the second pass, if some records modify then the transaction is aborted and restarted again. Hence, prior works [28] have shown *Calvin* to perform poorly in such settings. Another strong critic against *Calvin* is that in case of failures, it requires a replica node that executes the same set of operations as the node responding to client query. This implies that Calvin is not suitable under failures for use with partitioned databases. Also, the requirement for replica node, reduces the system throughput.

Figure 16 presents a comparison of NO_WAIT algorithm (employing EC protocol) and Calvin. For this experiment we use the TPC-C *Neworder* transactions, and vary the number of server nodes from 2 to 16. These transactions are required to update the order number in their districts. Hence, the deterministic protocols such as Calvin suffer performance degradation.

7 OPTIMIZATIONS

In earlier sections, we presented a theoretical proof and an evaluation of Easy Commit protocol, which proved its relevance in the space of existing commit protocols. We now discuss some optimizations for the EC protocol.

An optimized version of the EC protocol would allow achieving further gains in comparison to both the 2PC and 3PC protocols. A simple approach is to reduce the number of messages transmitted in the second phase. In the optimized protocol, each node only forwards messages to those nodes from which it has not received a *Global-Commit* or *Global-Abort* message. Another simple optimization is to ensure early cleanup, that is reduction of implementation enforced wait (refer Section 5.3). To achieve this, each node would maintain a lookup table, where an entry for each transaction is added, on receiving the first *Global-Commit* or *Global-Abort* message. The remaining messages, addressed to the same transaction, would be matched in the table and deleted. We would also need to periodically, flush some of the entries of the table, to reclaim memory. Interestingly, such an optimization would allow implementing a variant of EC protocol that does not require any "implicit" acknowledgments. Note a similar limited variant for 3PC protocol can be constructed where the coordinator does not wait for acknowledgments after sending the *Prepare-to-Commit* messages, and directly transmits *Global-Commit* message to all the cohorts. Our proposed optimized version is comparable to this 3PC variant.

8 RELATED WORK

The literature presents several interesting works [1, 23, 53] that suggest the use of *one phase commit* protocol. These works are strictly targeted at achieving performance, rather than consistency. Clearly, none of these works satisfy the non-blocking requirement, expected of a commit protocol.

Several variants to the 2PC protocol [20, 25, 29, 31, 33, 36, 40, 44, 49] have been proposed that aim at improving its performance. Presumed-commit and presumed-abort [36] work by reducing a single round of message transmission between the coordinator and the participants, when the transaction is to be committed or aborted, respectively. Gray and Reuter [25] present a series of optimizations for enhancing the 2PC protocol such as lazy commit, read-only commit and balancing the load by coordinator transfer. Group commit [20, 40] helps to reduce the commit overhead by committing a batch of transactions together. Samaras et al. [49] design several interesting optimizations to improve the performance of 2PC protocol. They present heuristics to reduce the overhead of logging, network contention and resource conflicts. Compared to all of these works, we present EC protocol, which is not only efficient, but also satisfies the non-blocking property.

Levy et al. [33] present an optimistic 2PC protocol that releases the locks held by a transaction once all the nodes agree to commit. In case a node decides to abort the transaction then to prevent violation of database atomicity, compensating transactions are issued to rollback the changes. Although their approach does not guarantee non-blocking behavior, but we believe the idea of optimistic resource release can be integrated with Easy Commit protocol to achieve further performance.

Boutros and Desai [44] present another variant to 2PC protocol which forces each node to send an additional message in case of a communication failure between the coordinator and the participant. Their approach is only susceptible to the cases where there is a message loss. However, their work does not resolve blocking under site failures and can be integrated with our work to achieve further resilience during message loss.

Haritsa [29] et al. improve the performance of the 2PC protocol, in the context of *real-time* distributed systems. Their protocol permits a conflicting transaction to access the non-committed data. This can lead to cascading aborts, and is not suitable for use with the traditional distributed databases. Our technique, on the other hand, is independent of the underlying concurrency control mechanism, and does not cause any special aborts.

Jiménez-Peris et al. [31] also allow their system to optimistically fetch the uncommitted data, thereby rendering the 2PC performance. However, their protocol is tailored for usage alongside strict two-phase locking, and assumes existence of an additional replica of each process. Our technique is not tailored to any specific concurrency control mechanism, and neither assumes existence of any extra process. Also, we believe these heuristics can be used alongside EC protocol, to render further benefits.

Reddy and Kitsuregawa [58] modify the 3PC protocol by introducing the notion of backup sites. With the help of backup sites they are able to achieve better performance than 3PC, but their approach blocks in case of multiple failures. Easy Commit is non-blocking and does not require any backup sites.

There also have been works [13, 27] that provide better performance bounds than the 3PC protocol if the number of failures are sufficiently less than the participants. Easy Commit does not bound the number of failures, and is nearly as efficient as 2PC.

Gray and Lamport [24] developed an interesting non-blocking version of 2PC protocol using Paxos [32]. Their approach shows that 2PC protocol is a variant of general consensus protocol. However, to ensure non-blocking property they require use of an extra set of acceptor nodes and in the worst case it can be shown that the number of messages transmitted in their approach is $O(n^2)$. Easy Commit is a hybrid between 2PC and 3PC protocol, which is nearly as efficient as former and non-blocking as latter. It also does not require the paxos consensus algorithm, and hence, no additional requirement of acceptor nodes.

9 CONCLUSIONS

We present a novel commit protocol – Easy Commit. Our design of Easy Commit, leverages the best of twin worlds (2PC and 3PC), it is non-blocking (like 3PC) and requires two phases (like 2PC). Easy Commit achieves these goals by ensuring two key observations: (i) first transmit and then commit, and (ii) message redundancy. We present the design of the Easy Commit protocol and prove that it guarantees both safety and liveness. We also present the associated termination protocol and state cases where Easy Commit can perform independent recovery. We perform a detailed evaluation of EC protocol on a 64 node cloud, and show that it is nearly as efficient as the 2PC protocol.

ACKNOWLEDGMENTS

We would like to acknowledge the Microsoft Azure Research Award, which allowed us to conduct extensive experimentation on Microsoft Azure Cloud Services.

REFERENCES

[1] M. Abdallah, R. Guerraoui, and P. Pucheral. 1998. One-Phase Commit: Does it make Sense? *(ICPADS)*.
[2] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *Proc. VLDB Endow.* 7, 3 (2013), 12.
[3] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. 2016. Scalable Atomic Visibility with RAMP Transactions. *ACM Trans. Database Syst.* 41, 3 (2016), 45.
[4] P. A. Bernstein and N. Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (1981), 185–221.
[5] P. A. Bernstein and N. Goodman. 1983. Multiversion Concurrency Control - Theory and Algorithms. *ACM TODS* 8, 4 (1983), 465–483.
[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co.
[8] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. 2015. SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 104–121.
[9] K. Chen, Y. Zhou, and Y. Cao. 2015. Online Data Partitioning in Distributed Database Systems. In *Proceedings of the 18th International Conference on Extending Database Technology*. OpenProceeding.org, 1–12.
[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 143–154.
[11] Oracle Corporation. Oracle 9i Real Application Clusters concepts Release 2 (9.2), Part Number A96597-01.
[12] Transaction Processing Performance Council. 2010. TPC Benchmark C (Revision 5.11).
[13] P. Dutta, R. Guerraoui, and B. Pochon. 2004. Fast Non-blocking Atomic Commit: An Inherent Trade-off. *Inf. Process. Lett.* 91, 4 (2004), 195–200.
[14] C. Diaconu et al. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. ACM, 1243–1254.
[15] J. Baker et al. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*. 223–234.
[16] J. C. Corbett et al. 2012. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, 261–264.
[17] J. Shute et al. 2013. F1: A Distributed SQL Database That Scales. In *VLDB*.
[18] R. Kallman et al. 2008. H-store: A High-Performance, Distributed Main Memory Transaction Processing System. *PVLDB* 1 (2008), 1496–1499.
[19] B. Fung. The embarrassing reason behind AmazonâĂŹs huge cloud computing outage this week. The Washington Post, GA, USA.
[20] Dieter Gawlick and David Kinkade. 1985. Varieties of concurrency control in IMS/VS fast path. 8 (01 1985), 3–10.
[21] J. Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*. Springer-Verlag, 393–481.
[22] J. Gray. 1981. The Transaction Concept: Virtues and Limitations (Invited Paper) *(VLDB)*. 144–154.
[23] J. Gray. 1990. *A comparison of the Byzantine Agreement problem and the Transaction Commit Problem.* Springer New York, 10–17.
[24] J. Gray and L. Lamport. 2006. Consensus on Transaction Commit. *ACM TODS* 31, 1 (2006), 133–160.
[25] J. Gray and A. Reuter. 1992. *Transaction Processing: Concepts and Techniques* (1st ed.). Morgan Kaufmann Publishers Inc.
[26] R. Guerraoui. 1995. *Revisiting the relationship between non-blocking atomic commitment and consensus.* Springer Berlin Heidelberg, 87–100.
[27] R. Guerraoui, M. Larrea, and A. Schiper. 1996. Reducing the Cost for Non-blocking in Atomic Commitment. IEEE.
[28] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *Proc. VLDB Endow.* 10, 5 (2017), 553–564.
[29] Jayant R. Haritsa, Krithi Ramamritham, and Ramesh Gupta. 2000. The PROMPT Real-Time Commit Protocol. *IEEE TPDS* 11, 2 (2000), 160–181.
[30] M. P. Herlihy and J. M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM TOPLAS* 12, 3 (1990).
[31] Ricardo Jiménez-Peris, Marta Patiño Martínez, Gustavo Alonso, and Sergio Arévalo. 2001. A Low-Latency Non-blocking Commit Service *(DISC'01)*. Springer-Verlag.
[32] L. Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169.
[33] E. Levy, H. F. Korth, and A. Silberschatz. An Optimistic Commit Protocol for Distributed Transaction Management *(ACM SIGMOD)*. ACM, 88–97.
[34] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. 2013. Stronger Semantics for Low-latency Geo-replicated Storage *(NSDI)*. USENIX Association, 313–328.
[35] MemSQL. http://www.memsql.com.
[36] C. Mohan, B. Lindsay, and R. Obermarck. 1986. Transaction Management in the R* Distributed Database Management System. *ACM TODS* 11, 4 (1986).
[37] NuoDB. http://www.nuodb.com.
[38] S. A. O'Brien. Facebook, Instagram experience outages Saturday. CNN, GA, USA.
[39] M. T. Ozsu and P. Valduriez. 2011. *Principles of Distributed Database Systems* (3rd ed.). Springer-Verlag. 428–453 pages.
[40] T. Park and H. Y. Yeom. 1991. A Distributed Group Commit Protocol for Distributed Database Systems *(ICPADS)*.
[41] S. Patterson, A. J. Elmore, F. Nawab, D. Agrawal, and A. El Abbadi. 2012. Serializability, Not Serial: Concurrency Control and Availability in Multi-datacenter Datastores. *Proc. VLDB Endow.* 5, 11 (2012).
[42] A. Pavlo, C. Curino, and S. Zdonik. 2012. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems *(SIGMOD '12)*. ACM, 61–72.
[43] M. Pilkington. 2015. Blockchain Technology: Principles and Applications. In *Research Handbook on Digital Transformations*. SSRN.
[44] B. S. Boutros and B. C. Desai. A two-phase commit protocol and its performance *(DEXA)*. IEEE, 100–105.
[45] M. Sadoghi. 2017. ExpoDB: An Exploratory Data Science Platform. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017*.
[46] M. Sadoghi, S. Bhattacherjee, B. Bhattacharjee, and M. Canim. 2018. L-Store: A Real-time OLTP and OLAP System *(EDBT)*. OpenProceeding.org.
[47] M. Sadoghi, M. Canim, B. Bhattacharjee, F. Nagel, and K. A. Ross. 2014. Reducing Database Locking Contention Through Multi-version Concurrency. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1331–1342.
[48] M. Sadoghi, K. A. Ross, M. Canim, and B. Bhattacharjee. 2013. Making Updates disk-I/O Friendly Using SSDs. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 997–1008.
[49] G. Samaras, K. Britton, A. Citron, and C. Mohan. 1995. Two-phase commit optimizations in a commercial distributed environment. *Distributed and Parallel Databases* 3, 4 (1995), 325–360.
[50] D. Skeen. 1981. Nonblocking Commit Protocols *(SIGMOD)*. ACM, 133–142.
[51] D. Skeen. 1982. *A Quorum-Based Commit Protocol.* Technical Report.
[52] D. Skeen and M. Stonebraker. 1983. A Formal Model of Crash Recovery in a Distributed System. *IEEE Trans. Softw. Eng.* 9, 3 (1983), 219–228.
[53] J.W. Stamos and F. Cristian. 1990. A Low-Cost Atomic Commit Protocol. In *Proceedings of the 9th Symposium on Reliable Distributed Systems*. IEEE, 10–17.
[54] M. Stonebraker. 1986. The Case for Shared Nothing. *Database Engineering* 9 (1986), 4–9.
[55] A. Sulleyman. Twitter Down: Social Media App And Website Not Working. The Independent, UK.
[56] A. et al. Thomson. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems *(SIGMOD)*.
[57] VoltDB. https://www.voltdb.com/.
[58] P. K. Reddy and M. Kitsuregawa. 1998. Reducing the Blocking in Two-Phase Commit Protocol Employing Backup Sites. In *COOPIS'98*. IEEE, 406–416.