Distributed and Parallel Databases (2020) 38:287–333
https://doi.org/10.1007/s10619-019-07267-w

Efficient and non-blocking agreement protocols

Suyash Gupta[1] · Mohammad Sadoghi[1]

Published online: 13 April 2019

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 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 EasyCommit protocol which leverages the best of both worlds (2PC and 3PC), that is non-blocking (like 3PC) and requires two phases (like 2PC). EasyCommit achieves these goals by ensuring two key observations: (i) first transmit and then commit, and (ii) message redundancy. We present the design of the EasyCommit 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. To cater the needs of geographically large scale distributed systems we also design a topology-aware agreement protocol (Geo-scale EasyCommit) that is non-blocking, safe, live and outperforms 3PC protocol.

Keywords Agreement · Node failures · Geo-scale

✉ Suyash Gupta
sugupta@ucdavis.edu

Mohammad Sadoghi
msadoghi@ucdavis.edu

1 University of California, Davis, Davis, CA, USA

1 Introduction

Large scale distributed databases have been designed and deployed for handling commercial and cloud-based applications [7,17,19,39,47,51,54,63–65,67,75,77]. 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 applications require the transactions to adhere to ACID [27] 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 [49,55,62] that the distributed systems undergo node failures. Recent failures [23,52,74] 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 [11] and linearizability [37] ensures system correctness. However, these properties have a causal effect on the underlying parameters such as latency and availability, that is, a need for strong consistency causes a reduction in system availability.

There have been works that try to increase the database availability [5,6]. However, 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 [7,17,39,43,57,75]. Such a pragmatic shift has necessitated the use of agreement protocols such as Two-Phase Commit [26]. Commit protocols help in achieving the twin requirements of consistency and reliability in case of partitioned distributed databases. Prior research [14,39,58,73,75] 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 in 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 (execute the termination protocol) to preserve database correctness.

One of the earliest and most popular commitment protocol is the *two-phase commit* [26] (henceforth referred as 2PC) protocol. Figure 1 presents the state diagram [55,70] 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 tran-

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

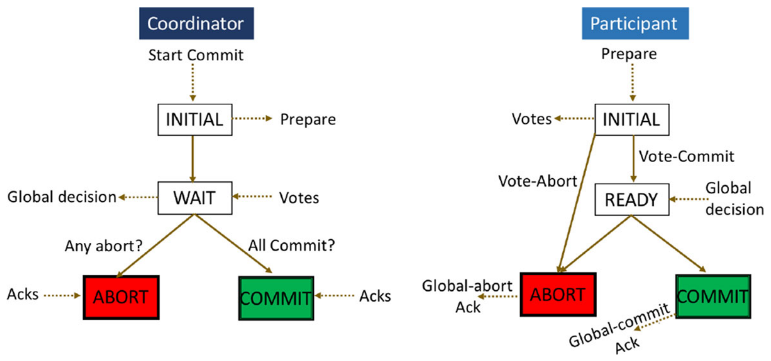


Fig. 1 Two-phase commit protocol

sitions 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 [55,69]. 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 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* [68,70]—henceforth referred as 3PC protocol. The 3PC protocol introduces an additional `PRE-COMMIT` state between the `READY` and `COMMIT` states, which ensures 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 *EasyCommit* (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 EasyCommit 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. EasyCommit protocol introduces message redundancy by ensuring that each participating node forwards the global decision to all the other participants (including the coordinator).

Prior works [3,50] have illustrated the wide-scale application of geographically large scale systems. Such systems, adhering to the philosophy of partitioned databases,

[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.

require complex agreement protocols that are both non-blocking and *topology-aware*. The key ingredient to these algorithms is their ability to take advantage to the geo-scale topology and present efficient results. It is important to understand that a simple geo-scale system consists of several clusters, and the communication across each clusters may be limited to few nodes. Hence, the design of 3PC (and even EC) may not reap benefit as it requires communication across all the participating nodes. This motivates us to learn from the design principles of EasyCommit protocol and construct a novel *topology-aware* agreement protocol for the geo-scale systems – *Geo-scale EasyCommit* (GEC). We now list down our contributions.

- We present the design of a new two-phase commit protocol (EasyCommit) and show it is non-blocking under node-failures.
- We design an associated termination protocol, to be initiated by the active nodes, on failure of the coordinating node and/or participating nodes.
- We re-use the EasyCommit principles and present a novel agreement protocol that caters to the needs of geographically large scale systems.
- We extend ExpoDB [33] framework to implement the EC protocol and its geo-scale variant. Our implementation can be used seamlessly with various concurrency control algorithms by replacing 2PC protocol with EC (and Geo-scale EasyCommit) to achieve efficient systems.
- We present a detailed evaluation of the EC protocol against the 2PC and 3PC protocol over two different OLTP benchmark suites: YCSB [16] and TPC-C [18], and scale the system upto 64 nodes, on the Microsoft Azure cloud.
- We also present an interesting evaluation of Geo-scale EasyCommit protocol against the 2PC and 3PC protocols when run across *four* geographically distant locations (across three continents). Our evaluation necessitates the need of an efficient and non-blocking geo-scale system.

The outline for rest of the paper is as follows: in Sect. 2, we motivate the need for EC protocol. In Sect. 3, we present design of the EC protocol. In Sect. 4, we present a discussion on assumptions associated with the design of commit protocols. In Sect. 5, we present the design of Geo-scale EasyCommit protocol. In Sect. 7, we present the implementations of various commit protocols. In Sect. 8, we evaluate the performance of EC and GEC protocols against the 2PC and 3PC protocols. In Sect. 11, we present the related work and conclude this work in Sect. 12.

2 Motivation and background

The state diagram representation for the two-phase commit protocol is presented in Fig. 1. In 2PC protocol, the coordinator and participating nodes require at most two transitions to traverse from `INITIAL` state to the `COMMIT` or `ABORT` states. We use Fig. 2 to present the interaction between the coordinator and the participants, on a linear time scale. The 2PC protocol starts with the coordinator node transmitting a *Prepare* message to each of the cohorts[3] and adding a *begin_commit* entry in its

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

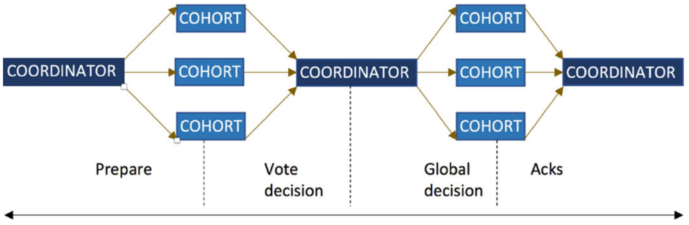


Fig. 2 Time span of 2PC protocol

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, the coordinator 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.

The 2PC protocol has been proved to be blocking [55,69] 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 [68,70].

Figure 3 presents the state transition diagram for the coordinator and cohort executing the three-phase commit protocol, while Fig. 4 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 then 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

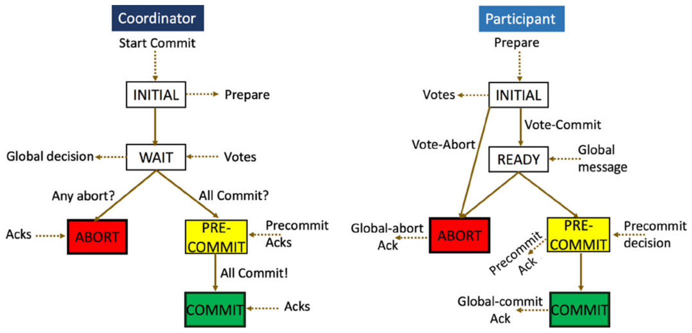


Fig. 3 Three-phase commit protocol

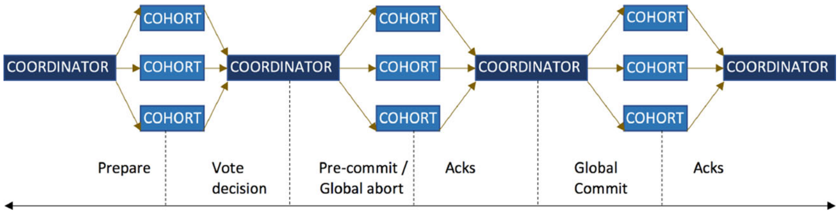


Fig. 4 Time span of 3PC protocol

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 [68]'s 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 to 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

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

be easily observed that the non-blocking characteristic of the 3PC protocol comes at an additional cost, an extra round of handshaking.

3 Easy commit

We now present the *EasyCommit* (*EC*) protocol. We assume the standard requirements for an agreement protocol [26,55,69], that is, nodes can fail, but messages cannot be delayed or lost. It has been shown [12,55] that no agreement protocol can handle message loss, delay or network partitions. Hence we assume existence of a reliable network that ensures message delivery. EC is a two-phase protocol, but unlike 2PC it exhibits non-blocking behavior, in the presence of node failures. The EC protocol achieves these goals through two key insights: (i) first transmit and then commit, and (ii) message redundancy. EC ensures that each participating node forwards the global decision to all the other participants. To ensure non-blocking behavior, EC protocol also requires each node (coordinator and participants) to delay commit until it transmits the global decision to all the other nodes. 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 Figs. 5, 6 and 7 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

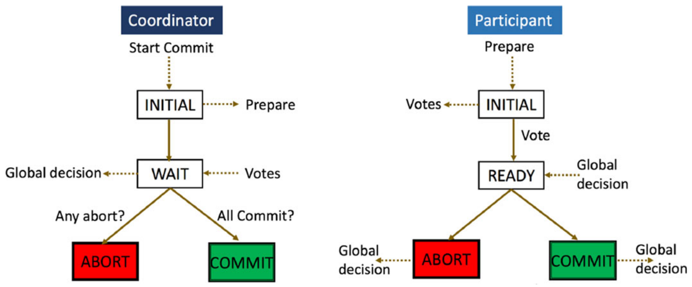


Fig. 5 EasyCommit protocol

Fig. 6 Coordinator's algorithm

```
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
```

Fig. 7 Participant'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
```

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 Sect. 3.2).

In Fig. 8, we also present the linear time scale model for the EasyCommit 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.

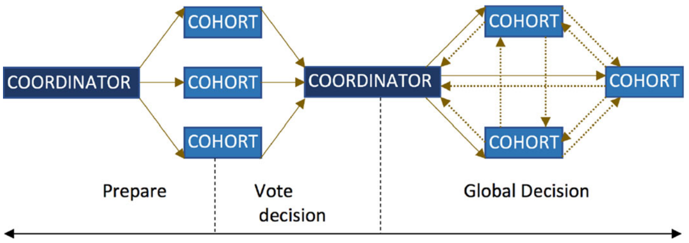


Fig. 8 Time span of EC protocol

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 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, it concludes loss of communication with the sender node, which in our case corresponds to failure of the sender. 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 makes progress. We illustrate the termination protocol by stating the actions taken by the coordinator and 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 (*global-abort-decision-reached*), next 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 initiates communication with other active cohorts to reach a common decision.
C. **Cohort timeout in `READY` state**—If the cohort timeouts in this state, then 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 (*global-abort-decision-reached*), next transmits the *Global-abort* message to all active participants and then aborts the transaction.

To prove correctness of EC protocol, Fig. 9 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 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*.

Table 1 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 properties of the EasyCommit protocol.

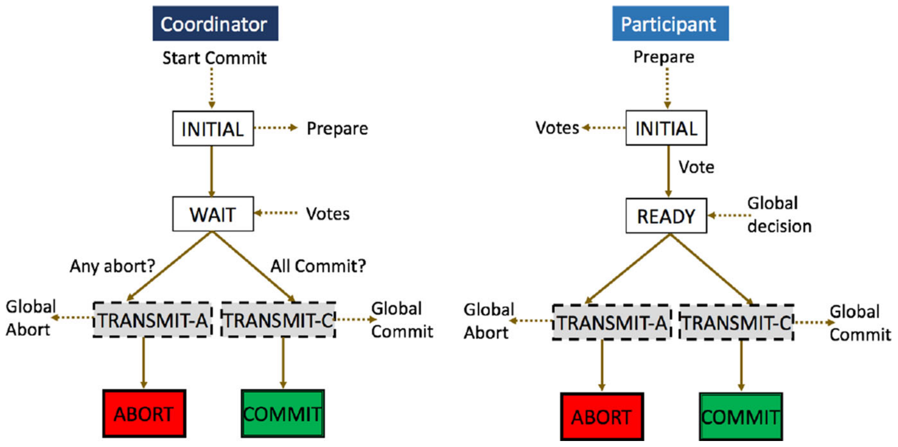


Fig. 9 Logical expansion of EasyCommit protocol

Table 1 Coexistent states in EC protocol (T-A refers to TRANSMIT-A and T-C refers to TRANSMIT-C)

	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

Theorem 1 *EasyCommit 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` first 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 2 *EasyCommit 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, it has a higher message complexity than 2PC. EC protocol's message complexity is $O(n^2)$, while the message complexity for 2PC is $O(n)$.

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 Sect. 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, loss and network partition

We now analyze the characteristics of 2PC, 3PC and EC protocols, 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 [12,55]. Prior works [31,55] 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. Assume that C receives *Vote-commit* message from all the cohorts. 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. Similarly, 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 [55]. Hence, no commit protocol is safe (or non-blocking) under message loss [12]. 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.

We can illustrate the impact of network partition on EC protocol by re-modeling our motivational example. In the motivational example, we assume that the coordinator C receives a *Vote-commit* message from all the participants X, Y and Z. Let us assume that the network partitions into two groups C, X and Y, Z. In EC protocol, we assume that if a node is able to send the global decision to all other nodes, then it can freely commit the transaction. In case of network partition, such an assumption is not valid. The coordinator C sends a *Global-commit* message to all the participants and assumes that Y and Z have failed (as they are unreachable). Hence coordinator proceeds with committing the transaction as it knows the global decision. Similarly, node X would commit the transaction. On the other hand, node Y and Z would eventually timeout, as they would not receive any message from the coordinator. They will follow case (C) of the termination protocol, select a new leader among themselves and abort the transaction. Thus, we conclude that EC is unsafe under message loss. It is easy to draw out similar examples that show both 2PC and 3PC are unsafe under network partitions.

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 it was executing prior to its failure, without any help from other active participants. Interestingly, the 2PC and 3PC protocols support only partial independent recovery [12,55].

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 scenarios:

(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 Geo-scale EasyCommit

In this section we design the EasyCommit protocol with regards to the geographically large-scale distributed database systems. A geographically large-scale (a.k.a

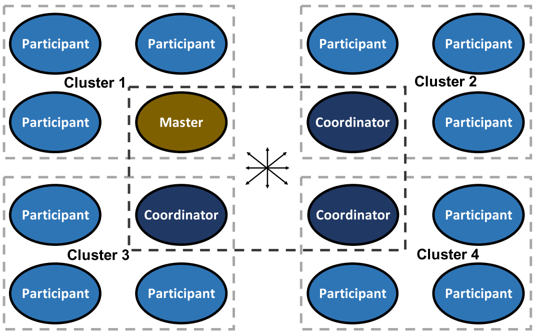


Fig. 10 Geo-scale system with four clusters

geo-scale) system consists of multiple *clusters* of nodes, where each cluster is located at a geographically different location. Each cluster is structured akin to a distributed system with one node acting as the coordinator and rest of the cluster nodes acting as the participants.

Traditional agreement protocols do not directly cater to the needs of these systems due to existence of high communication costs. Moreover, an efficient geo-scale agreement protocol should take into consideration the proximity of nodes within (or outside) a cluster. Thus, arises the need for a *two-level* agreement protocol. It is important to understand that a *two-level* agreement protocol does not necessarily imply execution of one agreement protocol atop another. For instance, for a geo-scale system, neither is the design of a two-level 3PC protocol intuitive, nor simply appending two 3PC protocols guarantees correctness (non-blocking property).

Figure 10 illustrates a simple geo-scale system consisting of four clusters. Each cluster consists of one *local* coordinator and three participants. The local coordinators are responsible for facilitating agreement within their clusters. There also exists a *global* coordinator (henceforth referred as *master*) for ensuring agreement between the geographically distributed clusters. Note that the master may also act as the *local* coordinator for its own cluster.

5.1 Motivation for Geo-scale EasyCommit

A desirable geo-scale agreement protocol should be safe and should benefit from the cluster topology. We know that 2PC protocol is blocking. Hence it suffices to show that the trivial extensions of 3PC protocol are unsafe for geo-scale systems. A simple design implies a two level execution, that is, each cluster runs a 3PC protocol and all the local coordinators execute another 3PC protocol among themselves. Within each cluster the local coordinator is responsible for the safe execution of 3PC protocol, while the master manages the 3PC protocol run among the local coordinators. When a transaction is ready to be committed, the master requests all the coordinators to provide their decisions. These coordinators, in turn, request their cluster nodes to vote and transmit the agreed decision to the master. The master then follows the 3PC sends a *Prepare-to-commit* message and waits for acknowledgments. The coordinators also follow the similar protocol and forward the *Prepare-to-commit* acknowledgments to

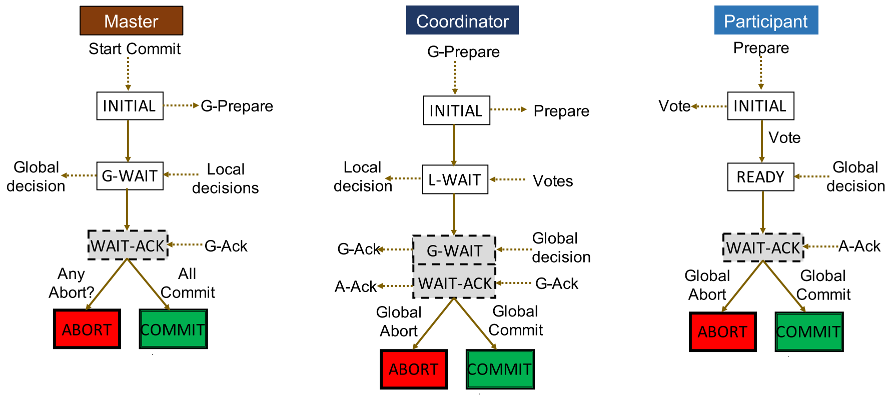


Fig. 11 Geo-scale EasyCommit protocol with state diagrams for master coordinator, local coordinator and participant

the master. Finally, the master sends a *Global-commit* message to all the coordinators, which they broadcast in their clusters.

Although, the aforementioned protocol is neat, it can be shown to be blocking. Consider a case where the master transmits the *Prepare-to-commit* message to all but one local coordinators (cluster 4 coordinator failed). Hence it *timeouts* (waiting for acknowledgment) and transmits *Global-commit* to all the local coordinators. These local coordinators would transmit the decision within their clusters. Furthermore, assume all the clusters, except cluster 4, have failed. Meanwhile, the participants in cluster 4 would *timeout*, select a new leader, reach a common decision and try to communicate with other clusters. However, as all the clusters are dead they are unsure of the global decision and are blocked. The key idea is that when only one cluster is alive and the top level communication is restricted among the coordinators, then system can block. A simple solution is to have system-wide communication (execute original 3PC algorithm), but such a solution is not scalable (communication expensive) when nodes are at geographically large distances.

5.2 Geo-scale EC commitment protocol

Figure 11 presents the state diagram for Geo-scale EasyCommit protocol (henceforth referred as GEC). We refer to the configuration identical to Fig. 10 for the ensuing discussion. Figures 12, 13 and 14 present the algorithm to be executed at the master node, coordinators and the participants. The geo-scale EasyCommit protocol also employs the same twin principles: (i) first transmit then commit, and (ii) message redundancy. These principles allow the protocol to attain safety and liveness. Additionally, we restructure the `WAIT` state. GEC introduces a new `G-WAIT` state at the master and coordinator nodes. Furthermore, it includes a `L-WAIT` state at the coordinators. GEC state machine also requires a `WAIT-ACK` state across all the nodes. The rendered state diagram encompasses a set of *half* states: (i) `WAIT-ACK` state at the master, and (ii) `READY` and `WAIT-ACK` states at a participant. These states are

```
 1: Send G-Prepare to all coordinators;
 2: Add begin_commit to log;
 3: Wait for (Local-commit or Local-abort) from all coordinators;
 4: if timeout then
 5:     Run Termination Protocol;
 6: end if
 7: if All messages are Local-commit then
 8:     Add global-commit-decision-reached in log;
 9:     Send Global-commit to all participants;
10: else
11:     Add global-abort-decision-reached in log;
12:     Send Global-abort to all participants;
13: end if
14: Wait for G-Ack from all coordinators;
15: if timeout then
16:     Run Termination Protocol;
17: end if
18: if Decision was Global-commit then
19:     Commit the transaction;
20:     Add transaction-commit to log;
21: else
22:     Abort the transaction;
23:     Add transaction-abort to log;
24: end if
```

Fig. 12 Master node's algorithm

referred to as *half* states as a node on these states never transmits any message. GEC also supports a *concurrent* state at the coordinator. This concurrent state arises from the merge of `G-WAIT` and `WAIT-ACK` state at the coordinator.

These changes could lead to an inadvertent interpretation that GEC requires up to *four* phases. However, the existence of a *concurrent* state permits a coordinator to receive messages of multiple types. Note that the master node can also act as the local coordinator. This implies that it would undergo two concurrent state machines, which could be trivially managed by employing a multi-threaded system.

GEC state machine uses the new `WAIT-ACK` state to achieve non-blocking guarantee. This state allows each node to deterministically commit (or abort) the transaction, if the system undergoes only node failures. The GEC agreement protocol starts when the transaction execution is completed. The master node sends out the *G-Prepare* message to all the coordinators and moves to the `G-WAIT` state. Each coordinator on receiving the *G-Prepare* message transmits a *Prepare* message to its cluster participants and moves to the `L-WAIT` state. When a participant receives a *Prepare* message, it decides to *Vote-commit* or *Vote-abort* the transaction. It transmits its decision to the coordinator and moves to the `READY` state. If a coordinator receives at least one *Vote-abort* message, it sends a *Local-abort* message to the master, otherwise it transmits a *Local-commit* message. Next, the coordinator moves to the `G-WAIT` state and waits for the global decision from the master. The master node aggregates all the responses from the coordinators and generates the global decision. It sends the *Global-commit* decision to all the coordinators, if it received all the *Local-commit* responses, otherwise it transmits the *Global-abort* decision. Ensuing this transmission, the master

```
1: Wait for G-Prepare from the master;
2: if timeout then
3:     Run Termination Protocol;
4: end if
5: Send Prepare to all cluster participants;
6: Add begin_commit to log;
7: Wait for (Vote-commit or Vote-abort) from all cluster participants;
8: if timeout then
9:     Run Termination Protocol;
10: end if
11: if All messages are Vote-commit then
12:     Add local-commit-decision-reached to log;
13:     Send Local-commit to master;
14: else
15:     Add local-abort-decision-reached to log;
16:     Send Local-abort to master;
17: end if
18: Wait for (Global-commit or Global-abort) from master;
19: if timeout then
20:     Run Termination Protocol;
21: end if
22: if Received Global-commit then
23:     Add global-commit-decision-reached to log;
24:     Forward Global-commit to all cluster participants;
25:     Send G-Ack to master and all coordinators;
26: else
27:     Add Global-abort-decision-reached to log;
28:     Forward Global-abort to all cluster participants;
29:     Send G-Ack to master and all coordinators;
30: end if
31: Wait for G-Ack from all coordinators;
32: if timeout then
33:     Run Termination Protocol;
34: end if
35: Forward an A-Ack to all cluster participants;
36: if Decision was Global-commit then
37:     Commit the transaction;
38:     Add transaction-commit to log;
39: else
40:     Abort the transaction;
41:     Add transaction-abort to log;
42: end if
```

Fig. 13 Local coordinator's algorithm

node moves to WAIT-ACK state and waits for acknowledgment messages (*G-Ack*) from all the coordinators.

When a coordinator receives the global decision, it forwards the global decision to its cluster participants. Once the coordinator has sent the global decision, it creates an acknowledgment message (*G-Ack*) and transmits the same to all the coordinators (including the master). Next, the coordinator transits to the WAIT-ACK state and waits for the *G-Ack* messages from other coordinators. Each participant on receiving the global decision moves to the WAIT-ACK state and waits for an *aggregated* acknowledgment message from its coordinator. When a coordinator receives all the required *G-Ack* messages, it aggregates them into one message (*A-Ack*), transmits that

```
1: Wait for *Prepare* from the coordinator;
2: if timeout then
3:     Run Termination Protocol;
4: end if
5: Send decision (*Vote-commit* or *Vote-abort*) to coordinator;
6: Add *ready* to log;
7: Wait for message from coordinator;
8: if timeout then
9:     Run Termination Protocol;
10: end if
11: if Decision received is *Global-commit* then
12:     Add *global-commit-received* in log;
13: else
14:     Add *global-abort-received* in log;
15: end if
16: Wait for *A-Ack* from the coordinator;
17: if timeout then
18:     Run Termination Protocol;
19: end if
20: if Decision was *Global-commit* then
21:     Commit the transaction;
22:     Add *transaction-commit* to log;
23: else
24:     Abort the transaction;
25:     Add *transaction-abort* to log;
26: end if
```

Fig. 14 Participant's algorithm

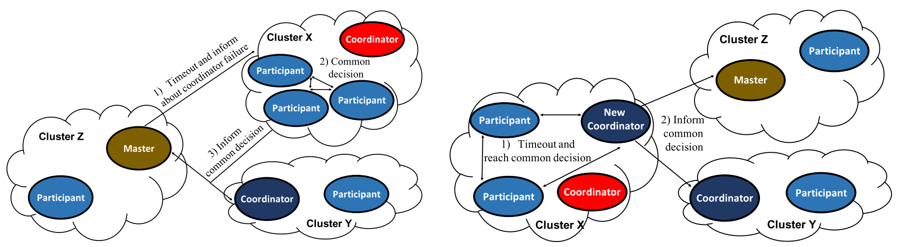
message to all its cluster participants and decides to commit (or abort) the transaction. Finally, the participants on receiving the *A-Ack* message, follow the global decision and commit (or abort) the transaction.

It is important to understand that a coordinator can receive *G-Ack* from some node before it receives global decision from the master. Hence, the coordinator keeps track of number of *G-Ack* messages it has received. This implies that the coordinator can switch between the `G-WAIT` and `WAIT-ACK` states. Thus, at coordinator, these states exist concurrently.

The preceding discussion permits following observations:

I. No node has a direct transition from `INITIAL` state to `ABORT` state.
II. Each coordinator after successfully transmitting the global decision in its cluster, transmits a *G-Ack* message to all other coordinators.
III. Each participant commits (or aborts) only after receiving an *A-Ack*.
IV. Each coordinator commits after sending an *A-Ack* to its participants.

Although (I) states that no node can have a direct transition from INITIAL state to ABORT state, it does not prohibit a coordinator from sending an *early* global decision. This implies that a coordinator which receives at least one *Vote-abort* message from its cluster participants, can send them the *Global-abort* decision. This will allow the cluster participants to move to `WAIT-ACK` states, without any wait. Note that this behavior of the coordinator is *safe* as the nodes need to wait for the *A-Ack* message, before they implement the global decision. Further, as one cluster has an *Vote-abort* message, so the transaction would be eventually aborted.



(a) Cluster Consultation with cluster X. **(b)** Reverse Cluster Consultation at cluster X.

Fig. 15 Cluster consultation initiated by the primary, and Reverse cluster consultation initiated by the participants, on failure of coordinator of Cluster X

5.3 Geo-scale EC termination protocol

We now present the GEC *termination* protocol that allows geo-scale systems, undergoing node failures, guarantee both safety and liveness. To ensure liveness we require each node to wait on a timer and *timeout* on expiry of its timer. A system undergoing an agreement protocol is live if the nodes are able to progress and not block. A geo-scale system is referred as safe, if at no instant its nodes are in conflicting states (refer Sect. 3).

Cluster consultation

GEC termination protocol requires communication among the nodes of different clusters, to attain twin guarantees of safety and liveness. Figure 15 lists down the steps taken by various nodes in case of a coordinator failure. In Fig. 15a we introduce the notion of Cluster consultation, which allows the master node to communicate with the participants of a cluster. This process occurs when the coordinator of a cluster fails, which in turn causes the master to *timeout*. Hence the master apprises the associated cluster about the failed coordinator and requests them to attain a common decision. Figure 15a presents the three steps in the process of Cluster consultation. On the other hand, if the participants detect the failure of their coordinator, prior to a message from the master, then they reach a common ground and initiate *Reverse Cluster consultation* (communication with the master and/or coordinators). *Reverse Cluster consultation* would lead to election of a new coordinator as illustrated in Fig. 15b.

Master timeout

A. **In** `G-WAIT` **state** If the master timeouts in this state, then it implies that the master did not receive the *Local-commit* or *Local-abort* message from one of the coordinators. Hence, the master would add a log entry (*global-abort-decision-reached*), transmit the *Global-abort* message to all other coordinators and initiate *Cluster consultation*.
B. **In** `WAIT-ACK` **state** If the master timeouts in this state, then it implies that the master did not receive a *G-Ack* from one of the coordinators. Hence, the master initiates *Cluster consultation* and then follows the global decision.

Coordinator timeout

C. **In** `INITIAL` **state** If the coordinator timeouts in this state, then it implies that the coordinator did not receive the *G-Prepare* message from the coordinator. Hence, the coordinator communicates with other active coordinators to reach a common decision.
D. **In** `L-WAIT` **state** If the coordinator timeouts in this state, then it implies that the coordinator did not receive the vote from one of the participants. Hence, the coordinator adds a log entry (*local-abort-decision-reached*) and transmits the *Local-abort* message to the master.
E. **In** `G-WAIT` **state** If the coordinator timeouts in this state, then it implies that the coordinator did not receive the global decision from the master. Hence, the coordinator interacts with other active coordinators to reach a common decision.
F. **In** `WAIT-ACK` **state** If the coordinator timeouts in this state, then it implies that the coordinator did not receive the *G-Ack* message from one of the coordinators. Hence, the coordinator communicates with the master node. If the master has the required *G-Ack* message, then it forwards the same, otherwise the master proceeds similarly to case (B).

Participant timeout

G. **In** `INITIAL` **state** If the participant timeouts in this state, then it implies that it did not receive the *Prepare* message from the coordinator. Hence, the participant consults other active participants, for agreement.
H. **In** `READY` **state** If the participant timeouts in this state, then it implies that it did not receive the global decision from the coordinator. Hence, the participant interacts with other active participants.
I. **In** `WAIT-ACK` **state** If the participant timeouts in this state, then it implies that it did not receive the *A-Ack* message from the coordinator. Hence, the participant decides to consult other active participants.

Master election

When the coordinator timeouts while waiting for a response from the master (cases C and E), it interacts with other coordinators to reach a common decision. Such a situation arises because the master node is experiencing a failure and has stopped responding to other nodes. Hence it is necessary to designate one of the active coordinators as the *new* master node. The election of the new master is akin to *leader election* in the EC protocol. The new master, communicates with the cluster of the failed master and requests them to select a new representative for the cluster. Next, the new master attempts to move the system to safe state. If the new master is in `INITIAL` or `L-WAIT` states, then it decides to globally abort the transaction. If the new master is in `G-WAIT` state, then it first checks whether any other coordinator previously received a global decision. If there exists a previous global decision then the new leader follows that decision, otherwise it proceeds to abort the transaction. If the new leader is in WAIT-ACK state, then evidently it knows the global decision and simply ensures that every other node is also in the same state.

The new master also needs to communicate with the participant nodes in the cluster of the old master node. The new master requests these nodes to select a new coordinator and asks them to announce the global decision on the fate of the transaction, if they are aware of any such decision. Note that this process is similar to the Cluster consultation.

Coordinator election

The participant nodes of a cluster initiate election of a new coordinator when they detect failure of their current coordinator. The new coordinator helps them to reach an agreement and initiates the *Reverse Cluster consultation*. If the new coordinator is in the INITIAL state, then it transmits *Local-abort* as the decision of the cluster, to the master. If the new coordinator is in READY state, then it consults other participants to check if anyone received the global decision. In case none of the active participants are aware of the global decision, the new coordinator performs *Reverse Cluster consultation*. If the new coordinator is in WAIT-ACK state, then it proceeds with *Reverse Cluster consultation*. Note that either using the *Cluster consultation* or *Reverse Cluster consultation* the new coordinator can make the system reach a stable state. If *Cluster consultation* occurs prior to election of new coordinator, then the new coordinator knows the correct state to proceed. Otherwise, the new coordinator either conveys its state to the master or acquires the information about the global state.

Cluster failure

In geo-scale systems, although node failures are common, *cluster failures* can also occur. We refer to a cluster failure as a state when either the whole cluster or a majority of cluster nodes (including the coordinator) have failed. An agreement protocol should guarantee both safety and liveness properties, in the presence of a cluster failure. As 3PC protocol is independent of the geo-scale topology, so it is both safe and live under a cluster failure.

GEC is also safe and live in the presence of a cluster failure. The master node would detect such a failure, when it attempts to perform Cluster consultation. If the master detects a cluster failure during the G-WAIT state, then it aborts the transaction and sends the other coordinators a *Global-abort* message. Similarly, if the master is in *WAIT-ACK* state and it discovers a cluster failure, then it proceeds with the global decision.

When a coordinator in the INITIAL state discovers a master failure, then it communicates with other clusters and tries to select a new master node. In case this coordinator discovers that all the clusters, except its own have failed, it appoints itself as the master and follows the termination protocol. Further, it is possible that the coordinator *timeouts* in the G-WAIT state and detects failure of all the clusters. If such is the case, then the coordinator at least has a guarantee that the other nodes could not have committed the transaction as it never sent a G-ACK to other coordinators. Hence it appoints itself as the master and aborts the transaction.

5.4 GEC correctness: safety and liveness

The EasyCommit extensions for the geo-scale systems are intended towards achieving a safe and scalable agreement protocol. Hence, we need to illustrate that GEC is non-blocking in scenarios considered earlier in this section.

For the sake of completeness, we revisit the scenario. We assume existence of four clusters, coordinators and a master node, akin to Fig. 10. The master node asks all the coordinators to send a decision and they reply with their local decisions (say *Local-commit*). The master sends the global decision to all the coordinators and its participants and dies. All but one coordinator (cluster 4) receives the global decision and forward it to their cluster participants. We assume that coordinator for cluster 4 could not receive the global decision as it failed. This implies that participant nodes of cluster 4 are not aware of the global decision. Furthermore, consider that except for cluster 4, all the nodes in every other cluster have failed. The nodes of cluster 4 can still make progress. Cluster 4 nodes have a guarantee that no other node could have committed (or aborted) the transaction. It is important to understand that cluster 4 nodes did not receive the global decision from their coordinator. Hence, their coordinator could not have sent an *G-Ack* message, which in turn ensures that other nodes could not have committed the results. The cluster 4 nodes can independently select a new leader and progress to safe state.

To prove the twin guarantees of safety and liveness for GEC it is easy to generate a representation similar to Table 1. The modified table will replace the hidden states `TRANSMIT-A` and `TRANSMIT-C` with `WAIT-ACK`. It is important to understand that `WAIT-ACK` is the state when nodes know the global decision. Hence, it is not an `UNDECIDED` state.

Theorem 3 *Geo-scale EasyCommit 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 that two nodes `p` and `q`, in different clusters, are in conflicting states (say `p` voted to abort the transaction and `q` voted to commit). This implies that one of them received *Global-abort* message, while other received the *Global-commit* message. This indicates that the master sent conflicting global decisions, which is a contradiction.

Another possibility is that due to master failure, at least one of the coordinators did not receive the message from the master and transmitted conflicting decision to its cluster. Such an assumption is again a contradiction, as cases (C) and (E) require coordinators to communicate with each other.

It is possible that the coordinator of `p`'s cluster has failed and the active nodes decide to move to a conflicting state. This is in contradiction with cases (G) to (I) as active participants need to elect a new coordinator and initiate *Reverse Cluster consultation*. If during *Reverse Cluster consultation* they find master to have failed, then they would initiate election of new master. Furthermore, if `p`'s cluster is unaware of the global decision then rules (I) to (IV) safeguard them from transitioning to a state in conflict with `q`. Note: `p`'s failed coordinator could not have transmitted an *G-Ack*. However, if `p`'s cluster knows the global decision and is waiting for an *A-Ack* then they can simply perform *Reverse Cluster consultation*. At this stage, it is also possible that every cluster except p's cluster has also failed. Still, the nodes can safely follow the global decision. □

Theorem 4 *Geo-scale EasyCommit protocol is live that is in the presence of only node failures, it does not block.*

Proof The proof for liveness property follows directly from the proof for Theorem 3. The termination protocol cases (A) to (I) provide sufficient guarantee that active nodes continue making progress under node failures. □

6 Geo-scale EasyCommit discussion

We now move beyond the crash failures and discuss the behavior of GEC in presence of message delays and network partitioning. Later in this section, we also analyze the impact of replication on GEC.

6.1 Message delay and loss

In Sect. 4.1, we show that none of the agreement protocols are safe under message delays, loss or network partitioning. Further, prior works [12,55] have shown that agreement protocols can only handle node failures. In fact it is hard for a node to distinguish between a message loss, message delay, network partitioning and node failure. Hence it is easy to design a case where GEC may block due to a message delay or network partitioning.

Message delays and loss are comparatively milder to manage, if they are restricted to some of the network paths. Such an assumption is valid as often message delay and loss are caused by network congestion and may only affect paths between some of the nodes. We now illustrate some of cases where GEC remains safe, if only some paths are experiencing message delay or loss.

- *Case 1* If a coordinator has already received a *Vote-abort* message from one of its participant, then even if the votes from some participant (say P) gets indefinitely delayed (or lost), the coordinator can simply send a *Local-abort* to the master. Further, on receiving the *Global-abort* decision from the primary, if the coordinator has not yet received a reply from the participant P, then it considers P failed, transmits the global decision to other participants and sends a *G-Ack* message to other coordinators. Whenever, the coordinator hears back from the participant P, it would forward P the global decision and move P to the correct global state.
- *Case 2* If a master node receives at least one *Vote-abort* (or *Local-abort*) message, then it can transmit the *Global-abort* decision to all the coordinators and its cluster participants. The master can consider the non-responding coordinator as failed and can initiate Cluster consultation with the participants of the associated cluster. Note that during these Cluster consultation master may realize that only the messages are getting delayed (or lost) and the coordinator is still non-faulty.

Table 2 Comparison of costs associated with different agreement protocols

	2PC	3PC	EC	GEC
Phases	2	3	2	3 (2 concurrent states)
Total messages	$4 \times (n-1)$	$6 \times (n-1)$	$n \times (n+1) - 2$	$m \times (m + 4 \times p - 2) - 3$
Intra-cluster	$4 \times (p-1)$	$6 \times (p-1)$	$(n+2) \times (p-1)$	$4 \times p \times (m-1)$
Inter-cluster	$4 \times p \times (m-1)$	$6 \times p \times (m-1)$	$(n+2) \times (n-p)$	$m \times (m-2) - 3$

We assume a system having m clusters and p nodes per cluster ($n = m \times p$)

- *Case 3* If the master did not receive a *G-Ack* message from some coordinator C, then it would initiate Cluster consultation, where, similar to the above case it would find that the messages are getting delayed or lost.
- *Case 4* If the coordinator timeouts in the `WAIT-ACK` because it did not receive a *G-Ack* message from some coordinator, then it can proceed as stated in Termination protocol case (F). If the messages from the master are also delayed (or lost), then the coordinator tries to communicate with other cluster coordinators. In case it receives no responses, it can simply ask its participants to follow the global decision.

Note that above cases are standalone and cannot be combined with cases where other nodes are in a different state and are also facing message delay or loss. For instance, when one coordinator is in *Case 4* and another coordinator is undergoing a timeout (due to message loss or delay) in *G-Wait* state, then the system can reach an unsafe state (if the global decision was *Global-Commit*).

6.2 Network partition

A network partition is a harder problem than message delay and loss. It isolates some nodes from rest of the nodes, such that no communication path exists among the nodes to carry the network traffic. This implies if a network partition occurs between two clusters then neither the two clusters can communicate nor can they determine if the other cluster has undergone a failure.

GEC allows the system to reach safe state in *Cases 1 and 2*, as the *Global-abort* becomes a common decision for the transaction. Similarly, if all the nodes have received the global decision and following which a network partition occurs, then the GEC protocol still remains safe for this transaction. Another case where the system will remain safe is when the system partitions before any coordinator could receive the *G-Prepare* message from the master. Here, each cluster would try to select a new master node (or appoint itself as the new master) and would abort this transaction. On the other hand, *Case 3* can easily be shown to lead the replicas in unsafe (different) states, if some cluster received the global decision, while another cluster partitions away from rest of the system, without receiving the global decision.

Note that following the network partition, nodes neither would be able to complete client transactions nor would be able to reach a global decision. This behavior results from the nature of a partitioned database, where each node only stores some part of the distributed database.

6.3 Cost and application of GEC

We now present a theoretical analysis of the costs associated with the GEC protocol. Table 2 compares the number of phases, total messages, intra-cluster messages and inter-cluster messages required by the four commit protocols. We assume a system having m clusters, where each cluster consists of p nodes. Thus, total nodes in the system are n, where $n = m \times p$.

We observe that the 2PC protocol requires less phases than the other two protocols, while although GEC requires three phases, one of the phases consists of two concurrent states. Further, all the protocols have a message complexity which is linear in the number of nodes (n). Additionally, GEC's message complexity has a component that is quadratic in the number of clusters m. This quadratic component comes from the communication among the coordinators.

Table 2 also shows that GEC requires the least amount of inter-cluster communication. This suggests that if the database is spread across several clusters, lying across geographically distant locations, then the GEC protocol could have performance benefit over the other two protocols. However, GEC faces intra-cluster overheads, as it has linear message complexity.

These metrics allow us to analyze the relevance of GEC against other protocols. In real systems, the number of nodes per cluster p is greater than or equal to the number of clusters m, $p \geq m$. Hence GEC would outperform the 3PC protocol. Further, if m is sufficiently large and the clusters are geographically apart, then GEC could achieve performance as good as 2PC. However, GEC may prove to be expensive, in case majority of the clusters are located in nearby regions, that is, the communication cost among the clusters is negligible. Thus, GEC derives its usefulness from the number of clusters and the geographical distance between the clusters.

7 EasyCommit implementation

We now present a discussion on our implementation of the EasyCommit protocol. We have implemented EC protocol in the ExpoDB platform [34,60]. ExpoDB is an in-memory, distributed transactional platform that incorporates and extends the Deneva [35] testbed. ExpoDB also offers secure transactional capability, and presents a flexible framework to study distributed ledger–blockchain [33].

7.1 Architectural overview

ExpoDB includes a lightweight layer for testing distributed protocols and design strategy. Figure 16 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.

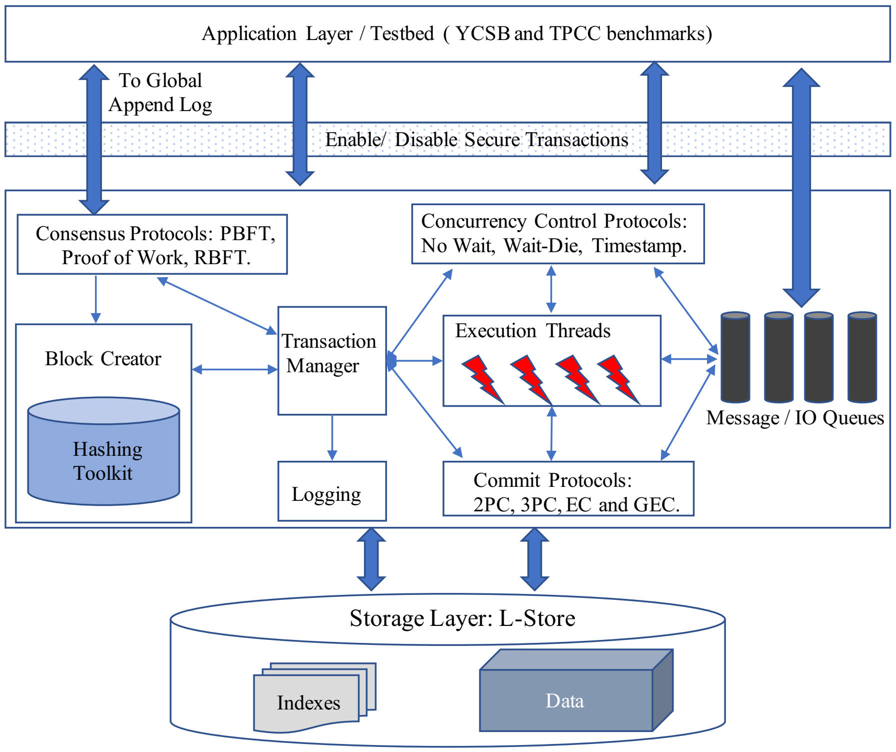

Application Layer / Testbed (YCSB and TPCC benchmarks)
To Global Append Log
Enable/ Disable Secure Transactions
Consensus Protocols: PBFT, Proof of Work, RBFT.
Concurrency Control Protocols: No Wait, Wait-Die, Timestamp.
Block Creator
Hashing Toolkit
Transaction Manager
Execution Threads
Message / IO Queues
Logging
Commit Protocols: 2PC, 3PC, EC and GEC.
Storage Layer: L-Store
Indexes
Data

Fig. 16 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

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 these transactions to a server process. The server process in turn executes these 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). 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.

7.2 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 3PC 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 *Prepare-to-commit* message to each participant. The participating nodes acknowledge the reception of the *Prepare-to-commit* message from the coordinator. The coordinating server on receiving these acknowledgments, starts the finish phase.

7.3 EasyCommit design

We now explain the design of EasyCommit 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, 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

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

has information about all the partitions) and transmitted as part of the *Global-commit* message.

7.4 Geo-scale EasyCommit design

To extend EasyCommit design to geographically large distributed systems, we adapt Geo-scale EasyCommit algorithm into a topology-aware implementation. We ensure that during the execution of GEC protocol none of the cluster members, except the coordinator communicate outside the clusters. Our implementation allows each cluster node to statically compute the identifiers of other nodes in the cluster. This requirement is met by informing each node about the cluster size, during initial system setup.

The master node needs to apprise each coordinator about identity of other coordinators. It is important to note that two transactions may not have the same master. We dedicate the node receiving the transaction as the master node for that transaction. This node can statically compute the coordinators for its transaction (using *modulo* operation). These coordinators acknowledge their elevated *status* once they receive a *G-Prepare* message. Similarly, coordinators request remaining nodes in the cluster to act as participants.

A key consideration in our implementation is the design of the master node. GEC algorithm states requirement of a master and a set of coordinators. This approach allows us to have a separate node demarcated as the coordinator in the cluster accommodating the master. However, we opt for an efficient design scheme where only one node performs the tasks of both the master and coordinator. This change requires us integrate the master and coordinator algorithms in a manner that prevents redundancy. For instance, once the master nodes transmits the *G-Prepare* message to other coordinators, it initiates the task of transmitting *Prepare* to its cluster participants. Similarly, the master tracks the incoming votes from its participants and the local decisions from other coordinators. Once the master has received all the votes, it transmits the global decision, to its cluster and the coordinators.

A key takeaway from our discussion in Sect. 5 was existence of intermediate states. We claim that these states could easily be merged with other states and do not introduce additional load. Our GEC implementation helps us to validate this claim. We allow each coordinator to transmit the *G-Ack* message as soon as its participants receive the global decision. Moreover, a coordinator could receive the *G-Ack* message from another coordinator prior to the global decision. Hence, the coordinator can track the number of *G-Ack* messages it has received and may piggyback *A-Ack* message to its participants along with the global decision from the master.

8 Evauation

In this section, we present a comprehensive evaluation of our novel EasyCommit protocol against 2PC and 3PC. As discussed in Sect. 7, we use the ExpoDB framework for implementing the EC protocol. For our experimentation, we adopt the evaluation scheme of Harding et al. [35].

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. For our evaluation, we adhere to a static partitioning scheme, where we assume that the data items are partitioned across several servers. Note that our commitment algorithm does not depend on the scheme employed to partition the database. In our experiments, we do show the effect of data skew on the performance of our agreement protocol.

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 Harding et al. [35] motivated us to use it for concurrency control.

8.1 Benchmark workloads

We test our experiments on two different benchmark suites: YCSB [16] and TPC-C [18]. We use YCSB benchmark to evaluate EC protocol on characteristics interesting to the OLTP database designers (Sects. 8.2–8.5) and use TPC-C to gauge the performance of EC protocol on a real world benchmark (Sects. 8.6 and 8.7).

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

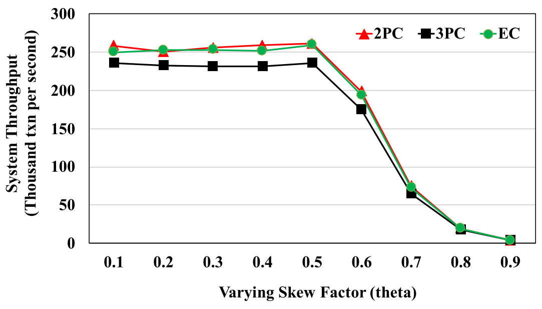


Fig. 17 System throughput (transactions per second) on varying the skew factor (`theta`) for the 2PC, 3PC and EC protocols. These experiments run the YCSB benchmark. Number of server nodes are set to 16 and partitions per transaction are set to 2

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.

8.2 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 17 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 –

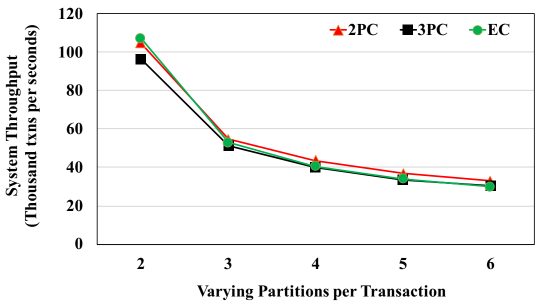


Fig. 18 System throughput (transactions per second) on varying the number of partitions per transactions for the commit protocols. These experiments use YCSB benchmark. The number of server nodes are set to 16 and `theta` is set to 0.6

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). However, EC is able to commite up to 20K more transactions than 3PC per second, and attains throughput gains of 12%.

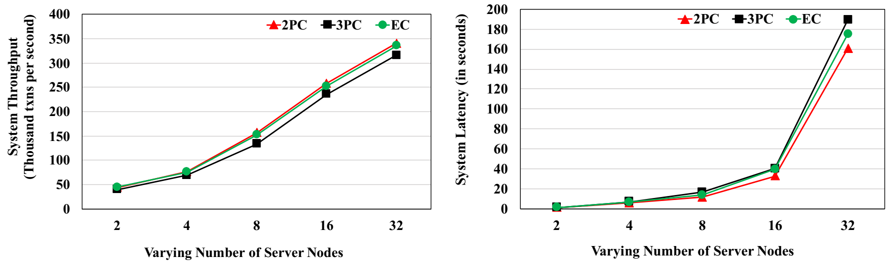
8.3 Varying partitions per transaction

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 18 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 $1:1$.

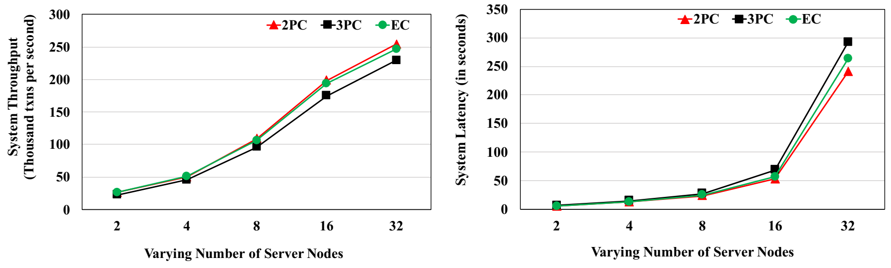
It can be observed that on increasing the number of partitions per transaction there is a dip in the system throughput, across all of the commit protocols. On moving from 2 to 4 partitions there is an approximate decrease of 55%, while the reduction is system performance is around 25% from 4 partitions 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 [18]. Further, EC is commits up to 10,500 more transactions per second than 3PC, and attains up to 11% more throughput that 3PC.

8.4 Varying server nodes

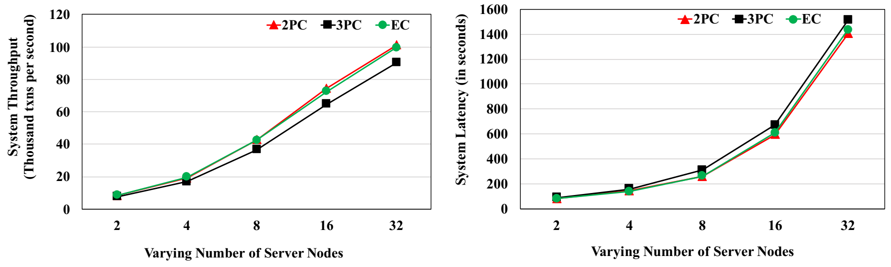
We study the effect of varying the number of server nodes (from 2 nodes to 32 nodes) on the system throughput and latency, for the 2PC, 3PC and EC protocols. In Fig. 19 we set the number of partitions per transaction to 2 and plot graphs for the low contention ($\texttt{theta} = 0.1$), medium contention ($\texttt{theta} = 0.6$) and high contention ($\texttt{theta} = 0.7$). In these experiments, we increase size of YCSB table in accordance to the the increase in number of server nodes.



(a) Low contention – (`theta` = 0.1).



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

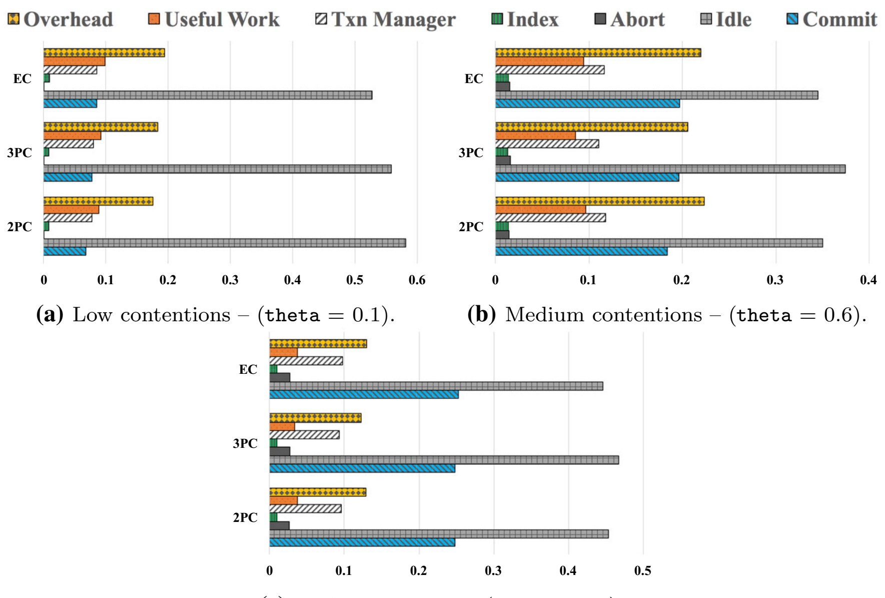


(c) High contention – (`theta` = 0.7).

Fig. 19 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

In Fig. 19, 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



(a) Low contentions – (`theta` = 0.1). **(b)** Medium contentions – (`theta` = 0.6). **(c)** High contentions – (`theta` = 0.7).

Fig. 20 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

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. At `theta` = 0.1, EC commits up to 19900 more transactions per second than 3PC and attains up to 12.5% more throughput than 3PC. At `theta` = 0.6, EC commits up to 17200 more transactions than 3PC and attains up to 15.8% more throughput than 3PC. Finally, at `theta` = 0.7 EC commits up to 9200 more transactions than 3PC and attains up to 15.7% more throughput than 3PC.

8.4.1 Latency

In Fig. 19, 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.

8.4.2 Proportion of time consumed by various components

Figure 20 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.

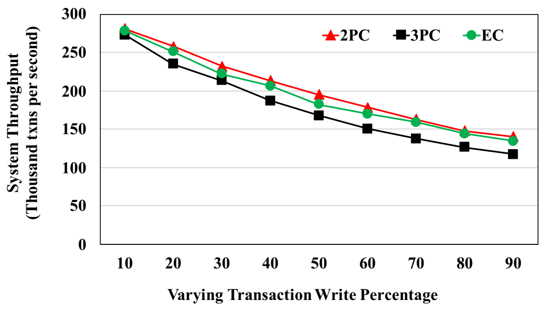


Fig. 21 System throughput (transactions per second) on varying the transaction write percentage for the 2PC, 3PC and EC protocols. These experiments use YCSB benchmark, and set the number of server nodes to 16 and partitions per transactions to 2

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 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.

8.5 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 (refer Fig. 21) 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

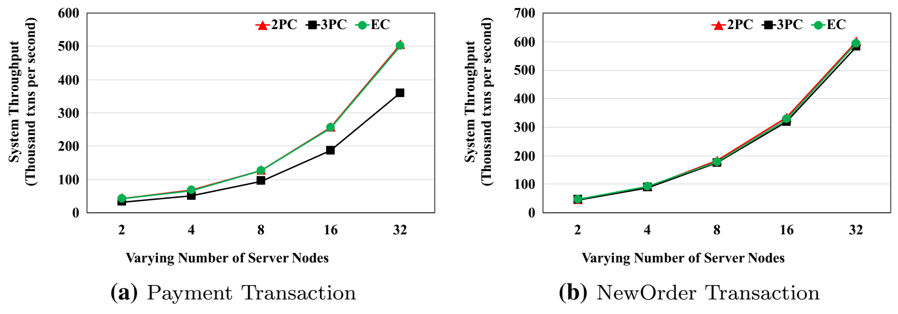


Fig. 22 System throughput on varying the number of server nodes, on the TPC-C benchmark. The number of warehouses per server are set to 128

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. To summarize EC protocol commits up to 17700 more transactions per second than 3PC and achieves up to 14.5% more throughput than 3PC.

8.6 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 22 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 sets 128 warehouses per server, and, hence a multi-partition can access two co-located partitions (that is on a single server).

Figure 22a 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 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. EC protocol commits up to 140K more transactions that 3PC and achieves up to 39% more throughput than 3PC protocol.

Figure 22b 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

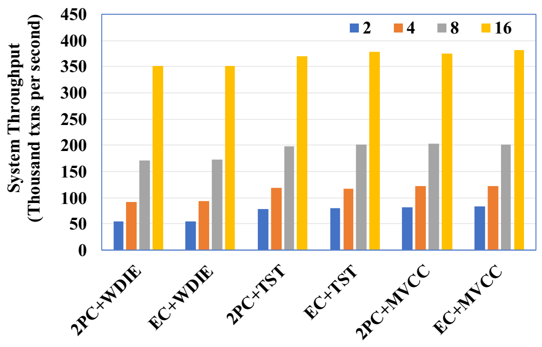


Fig. 23 System throughput achieved by three different concurrency control algorithms. For experimentation, we use the TPC-C *Payment* transaction, and vary the number of server nodes to 16. The number of warehouses per server are set to 128. Here WDIE and TST refer to `WAIT-DIE` and `TIMESTAMP`, respectively

higher throughput. Also, as there are only 10% multi-partition transactions, so all the protocols achieve nearly the same performance.

8.7 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 23 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` [8], `TIMESTAMP` [8] and `MVCC` [9]. 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* [75]. *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 [35] 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 24 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

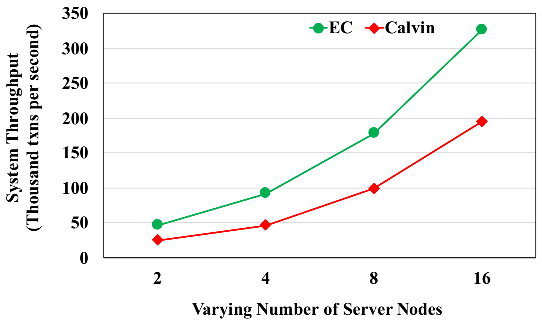


Fig. 24 Comparison of throughput achieved by the system executing Calvin versus the system implementing the combination of No-Wait + *EC* protocol. In this experiment we use the TPC-C *Neworder* transaction, and vary the number of server nodes to 16. The number of warehouses per server are set to 128

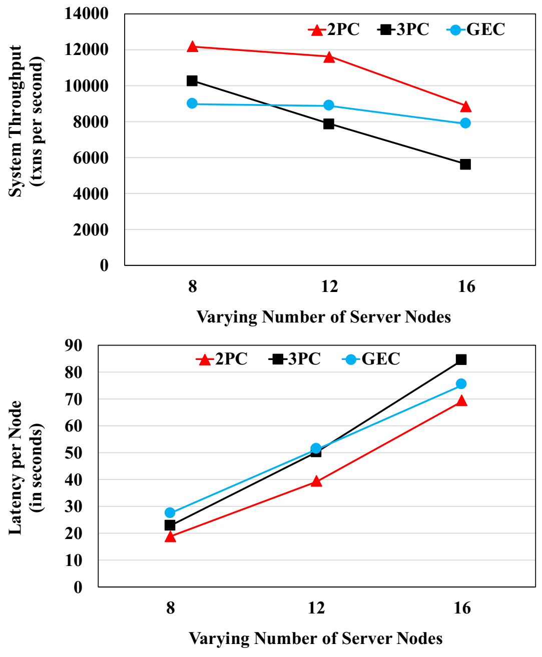


Fig. 25 System throughput and latency per node on varying the number of server nodes, across four regions, where each region consists of equal number of nodes. For these experiments we use YCSB benchmark and set both partitions per transaction and requests per transaction equal to total number of nodes in each run (Color figure online)

the order number in their districts. Hence, the deterministic protocols such as Calvin suffer performance degradation. In this graph, EC protocol commits up to 275K more transactions per second than Calvin and achieves a throughput gain of 87% with respect to Calvin.

8.8 Geo-scale EasyCommit

We now present an evaluation of our GEC algorithm against 2PC and 3PC algorithms. In Sect. 5 we present the need for a topology-aware algorithm that performs efficiently in the presence of geographically large clusters. Geo-scale systems require existence of algorithms that can reduce the latency and do not require communication between all the nodes, across clusters. Figure 25 illustrates the performance achieved by GEC and validates our aforementioned claim through interesting comparisons against 3PC and 2PC protocols.

In these figures we run experiments across four geographically distant regions: US East (Ohio), US West (N. California), EU (Ireland) and Asia Pacific (Mumbai). We use Amazon AWS to run these experiments and establish Virtual Private Network to facilitate these experiments. We use `m5x2large` nodes for servers and `t2x2large` nodes for clients. In each run, we ensure there is a one-to-one mapping between the server and the client, that is we have equal number of client and server nodes. We set the number of partitions per transaction and requests per transaction equal to number of servers. We vary the number of server nodes in each region from 2 to 4.

Figure 25 illustrates that on increasing the number of server nodes the system throughput decreases across all the three protocols. This phenomena can be easily attributed to: (i) increase in number of partitions per transaction, and (ii) increase in number of requests per transaction. This implies that as the number of server nodes increase, each transaction needs to complete more requests and each request refers to a different partition. An increase in number of server nodes also leads to faster degradation in the performance of 3PC and 2PC protocols, in comparison to GEC protocol. This behavior arises as traditional agreement protocols are oblivious to the underlying cluster topology. This in turn causes existence of a single master that communicates with all the nodes. Interestingly, the throughput of GEC is poor when the cluster size is small, as it performs more work in comparison to 3PC protocol. On increasing the number of nodes per cluster, the throughput reduction is less for GEC (a proof that it is *topology-aware*). Moreover, GEC performs significantly better than 3PC and nearly as good as 2PC protocol. GEC attains up to 40% more throughput than 3PC protocol.

Figure 25 also presents the statistics for 99 percentile latency incurred at each server node. It is easy to gauge that the latency per node increases significantly, on increasing the number of server nodes. When the number of nodes per region is small, the GEC protocol suffers from high latency, as it requires higher communication. However, on increasing the number of nodes per cluster, GEC protocol outperforms 3PC and has latency closer to 2PC protocol. It is our assertion that a further increase in the number of nodes per cluster should bridge the gap between 2PC and GEC throughput and latency values. It is important to understand that GEC is non-blocking and attains performance of the order of 2PC protocol.

Messages Communicated Figure 26 illustrates the number of messages transmitted or received per second (in thousands), on an average, at each server node. These results are derived from the experiment of Fig. 25. In our system, the size of each message is in the range 128 bytes to 152 bytes. As the number of nodes per region increases, there is a subsequent increase in the number of messages transmitted (or received) at each server. As 2PC achieves higher throughput than both GEC and 3PC, so it transmits higher number of messages. Similarly, the throughput of GEC improves in comparison to other protocols, which leads to a larger set of messages transmitted (or received) by GEC. Further, when each region has four server nodes, then from Fig. 25 we learn that throughput of GEC is quite close to throughput for 2PC. Hence GEC transmits nearly the same number of messages. Note that as GEC has a higher message complexity

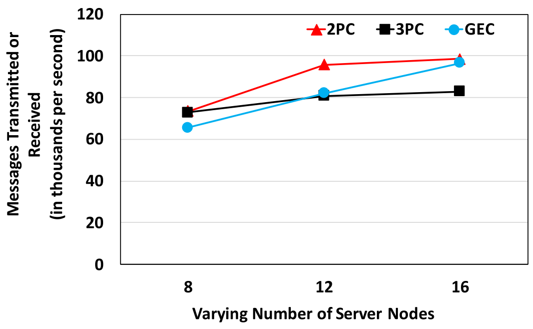


Fig. 26 Messages transmitted or received at any server node, on an average. Here, each region consists of equal number of nodes. We use YCSB benchmark and set both partitions per transactions and requests per transaction equal to total number of nodes in each run

Table 3 System throughput on increasing number of nodes per region

Nodes per region	4	5
2PC	9725.07	8984.80
3PC	8845.07	7664.05
GEC	9187.95	8282.70

These experiments were run across three regions and both partitions per transaction and requests per transaction were equal to total number of nodes in each run

than 2PC, so a further increase in the number of server nodes could cause GEC to incur more messages.

Varying nodes per region We use Table 3 to affirm our above insights across *three* regions, by varying the number of nodes per region. GEC protocol attains higher throughput than 3PC protocol even on a smaller setup consisting of just three clusters. This proves that GEC protocol is useful across setups of different topologies.

9 Optimizations

In earlier sections, we presented a theoretical proof and an evaluation of EasyCommit 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 Sect. 7.3). 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.

10 Replication

Until now our discussion surrounded a system that employs a partitioned distributed database and requires the use of a commit protocol for guaranteeing its correctness [1]. Prior works [10,21,25,72] have employed replication to tolerate a greater set of faults that can handled by a partitioned database, at the cost of redundancy. Further, replication also necessitates the use of a replica management protocol [1,40] to bring all the replicas at the same state.

We envision the design of our EasyCommit protocol quite close to the Paxos [40] algorithm, which is capable to handling crash failures. Paxos is a two-phase algorithm, where a leader presents its proposal to a set of acceptors. Each acceptor, on receiving a proposal, informs the leader about the latest proposal that it has received. If the leader detects its proposal in the response from a majority of acceptors, then it requests the acceptors to accept its proposal. A leader marks its proposal accepted, when it has heard back from a majority of acceptors.

Akin to Paxos, EC is also a two-phase protocol, which can be easily extended to a replicated system. In EC the role of acceptors can be played by the participants. Each participant also sends its vote (similar to latest proposal) to the coordinator (leader). Based on the votes of the participants, the coordinator sends a global decision to all participants, which is again transmitted by all the participants to each other. Hence EC protocol can be seamlessly shown to work with either replicated databases or partitioned databases.

Akin to EC, our Geo-scale EasyCommit protocol can also work with geographically replicated databases. GEC utilizes the notion of cluster, which has been adopted by prior works [4,17], to reduce latency between geographically distant locations. Further, GEC limits the communication across clusters to coordinator nodes. This property can be useful as general fault-tolerant protocols [40,46] are not aware of cluster topology, which could cause expensive communication costs. Finally, our notion of Cluster consultation can help to remove the dependency on the coordinator (leader), at each cluster, in case the coordinator crashes. Note that fault-tolerant protocols do not require agreement among all the replicas to work correctly. These protocols only suggest that a majority of replicas should always be non-faulty. As our agreement protocols (EC and GEC) require agreement among all the nodes, so their extension to replicated system would allow them to handle a larger set of failures. A comprehensive study of replication and application of our agreement protocols to fault-tolerant settings is a subject of exciting future work.

11 Related work

The literature presents several interesting works [2,28,71] 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 [13,24,30,36,38,41,48,56,66] have been proposed that aim at improving its performance. Presumed-commit and presumed-abort [48] 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 [30] 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 [24,56] helps to reduce the commit overhead by committing a batch of transactions together. Samaras et al. [66] 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. [41] 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 EasyCommit protocol to achieve further performance.

Boutros and Desai [13] 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 et al. [36] 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. [38] 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 [61] modify the 3PC protocol by introducing the notion of backup sites. With the help of backup sites they are able to achieve higher throughput, but their approach blocks in case of multiple failures. EasyCommit is non-blocking and does not require any backup sites.

There also have been works [20,32] that provide better performance bounds than the 3PC protocol if the number of failures are sufficiently less than the participants. EasyCommit does not bound the number of failures, and is nearly as efficient as 2PC.

Gray and Lamport [29] developed an interesting non-blocking version of 2PC protocol using Paxos [40]. 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 $O(n^2)$. EasyCommit 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 designated acceptor nodes.

Percolator [59] presents a distributed database that facilitates incremental processing. It allows random access to data and resolves transaction conflict by locking the associated data. It also employs the two-phase commit protocol to ensure consistency. Percolator uses 2PC protocol to protect itself against machine failures. This also acts as a good application for commit protocols (GEC or EC) to provide non-blocking commitment.

Prior works [15,17,44,76] employ a replicated database that partitions the data across several datacenters. These works employ Paxos [40] which provides a fault-tolerant consensus. In this work, we present EC and GEC protocols that are designed to provide safety and liveness properties to non-replicated systems. The deployment of these protocols to a replicated service is orthogonal to our current work and acts an exciting future research. We now present some more details on these existing replicated databases.

Spanner [17] provides an interesting solution to resolve the wide area communication and handles a large number of client requests. It employs a large distributed database that spans several locations. Spanner replicates its data across a set of machines. Each data is assigned to a *Paxos group*, which maintains the data and facilitates fault-tolerant access to the data by the client. However, when a client request is essentially a distributed transaction that requires access to data spanning multiple paxos groups, then Spanner employs 2PC protocol. We believe Spanner can employ our protocol GEC, which not only provides efficient access to data spanning across geographical locations, but also is non-blocking.

Replicated Commit [44] modifies the principle behind Spanner, to attain a database that facilitates efficient transaction logging and commit. Replicated Commit suggests achieving agreement on the transactional result (that is, employing 2PC), prior to starting a Paxos based consensus. This helps the system limit the extra inter-datacenter latency, which are prevalent in Spanner. We believe this also acts an interesting application for our EC protocol as it can provide non-blocking guarantees to the system employing Replicated Commit by replacing the 2PC protocol.

MaaT [45] presents a distributed transaction processing system that renders optimistic concurrency control (OCC). MaaT's redesign allows OCC to employ the 2PC protocol without requiring any locks on any data-item during the commitment phase. Further, MaaT's optimistic concurrency control allows it to reduce the number of aborted transactions. MaaT's use of 2PC acts as another relevant use case for our EC protocol. As EC does not rely on any locking mechanism and is compatible with different concurrency control algorithms, so it can easily integrate with MaaT's architecture.

Leap [42] aims at designing a distributed transaction processing system that is devoid of any commitment protocol, such as 2PC. Leap removes the need of 2PC by converting each distributed transaction into a local transaction. To perform this task, Leap uses the principles of data migration and locality. When a client requires access to some data from a node, such that the data lies on a different node (remote), then the data is copied from the remote node to the local node. Hence the local node is now responsible for managing the data, as each data is managed by only one node. Although Leap seems exciting, it can face load imbalance if a burst of client requests are targeted at a specific node. Further, Leap expects data locality, without which a large amount of time would be spent in transmitting data from one node to another. This issue can exacerbate if the data size increases.

CockroachDB [15] is a strongly consistent replicated distributed database that extends the principles of Spanner. CockroachDB employs Raft Ongaro and Ousterhout [53], which is an extension to Paxos, to guarantee efficient fault-tolerance across replicas. It also employs distributed transactions, which are fulfilled by a leader site, by querying the other sites. Once the leader site performs the required operation, it informs other sites to commit the transactions. This protocol is similar to 2PC protocol, which implies that it can serve as an application for GEC protocol.

TiDB [76] is an analytical and transactional processing database that is aimed at allowing clients performing faster analysis and querying. Its design share principles with both Spanner and CockroachDB. It also replicates data across a set of nodes, which are maintained using Raft. To ensure consistency of distributed transactions, it also employs MVCC Bernstein and Goodman [9] algorithm (Raft also employs a variant to MVCC).

FaunaDB [22] adheres to the principle of deterministic databases, by requiring the full knowledge of the transaction before executing the transaction. This allows FaunaDB to generate deterministic plans, which help it to process client queries efficiently, even at geo-scale level. Hence it does not require the use of a commit protocol to ensure consistency. However, this also limits the type of client transactions that FaunaDB can handle.

This work builds on top of our previous work [34]. In this paper we present the design of a novel agreement protocol (Geo-scale EasyCommit) for the geographically distant systems. We prove that GEC is non-blocking and upholds the key requirements of safety and liveness. We also present the associated *termination* protocol that allows GEC to work effortlessly under node failures. We implement GEC on ExpoDB [33] and present an interesting evaluation of GEC on *four* geographically distant locations, across three continents. Our results show that GEC outperforms 3PC and performs nearly as good as 2PC protocol.

12 Conclusions

We present a novel commit protocol—EasyCommit. Our design of EasyCommit, leverages the best of twin worlds (2PC and 3PC), it is non-blocking (like 3PC) and requires two phases (like 2PC). EasyCommit achieves these goals by ensuring two key observations: (i) first transmit and then commit, and (ii) message redundancy. We present the design of the EasyCommit protocol and prove that it guarantees both safety and liveness. We also present the associated termination protocol and state cases where EasyCommit can perform independent recovery. We learn from our EC protocol and design a novel agreement protocol (Geo-scale EasyCommit) that caters to the needs of a geographically large scale system. GEC protocol limits cost-expensive inter-cluster communication by facilitating cost-inexpensive within cluster communication. 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. We also evaluate GEC protocol on a setup scaling three continents and show that GEC is an efficient alternative to 3PC and performs nearly as good as blocking 2PC.

Acknowledgements We would like to acknowledge Thamir Qadah for the valuable discussions that helped us to design ExpoDB system. Further, we acknowledge the anonymous reviewers for their useful inputs and comments.

References

1. Abbadi, A.E., Toueg, S.: Maintaining availability in partitioned replicated databases. ACM Trans Database Syst **14**(2), 264–290 (1989). https://doi.org/10.1145/63500.63501
2. Abdallah, M., Guerraoui, R., Pucheral, P.: One-phase commit: does it make sense? ICPADS (1998)
3. Agrawal, D., El Abbadi, A., Mahmoud, H.A., Nawab, F., Salem, K.: Managing geo-replicated data in multi-datacenters. In: Proceedings of the 2013 Databases in Networked Information Systems—8th International Workshop, DNIS'13, pp. 23–43 (2013)
4. Amir, Y., Danilov, C., Dolev, D., Kirsch, J., Lane, J., Nita-Rotaru, C., Olsen, J., Zage, D.: Steward: scaling byzantine fault-tolerant replication to wide area networks. IEEE Trans. Dependable Secur. Comput. **7**(1), 80–93 (2010). https://doi.org/10.1109/TDSC.2008.53
5. Bailis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Highly available transactions: virtues and limitations. Proc VLDB Endow **7**(3), 181–192 (2013)
6. Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Scalable atomic visibility with RAMP transactions. ACM Trans Database Syst **41**(3), 15 (2016)
7. Baker, J., Bond, C., Corbett, J.C., Furman, J., Khorlin, A., Larson, J., Leon, J.M., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: providing scalable, highly available storage for interactive services. In: Proceedings of the Conference on Innovative Data system Research (CIDR), pp. 223–234 (2011)
8. Bernstein, P.A., Goodman, N.: Concurrency control in distributed database systems. ACM Comput Surv **13**(2), 185–221 (1981)
9. Bernstein, P.A., Goodman, N.: Multiversion concurrency control—theory and algorithms. ACM TODS **8**(4), 465–483 (1983)
10. Bernstein, P.A., Goodman, N.: An algorithm for concurrency control and recovery in replicated distributed databases. ACM Trans Database Syst **9**(4), 596–615 (1984). https://doi.org/10.1145/1994.2207
11. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA (1987a)
12. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley Longman Publishing Co., Boston, MA (1987b)
13. Boutros, B.S., Desai, B.C.: A two-phase commit protocol and its performance. In: IEEE, DEXA, pp. 100–105 (1996)
14. Chen, K., Zhou, Y., Cao, Y.: Online data partitioning in distributed database systems. In: Proceedings of the 18th International Conference on Extending Database Technology, OpenProceeding.org, pp. 1–12 (2015)
15. CockroachDB (2018). https://www.cockroachlabs.com/
16. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing, ACM, pp. 143–154 (2010)

17. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally-distributed database. In: 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), USENIX Association, pp. 261–264 (2012)
18. Council TPP (2010) Tpc benchmark c (revision 5.11)
19. Diaconu, C., Freedman, C., Ismert, E., Larson, P.A., Mittal, P., Stonecipher, R., Verma, N., Zwilling, M.: Hekaton: SQL Server's Memory-optimized OLTP Engine. ACM, pp. 1243–1254 (2013)
20. Dutta, P., Guerraoui, R., Pochon, B.: Fast non-blocking atomic commit: an inherent trade-off. Inf Process Lett **91**(4), 195–200 (2004)
21. El Abbadi, A., Skeen, D., Cristian, F.: An efficient, fault-tolerant protocol for replicated data management. In: Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, ACM, New York, PODS '85, pp 215–229 (1985). https://doi.org/10.1145/325405.325443
22. Freels, M.: FaunaDB: an architectural overview (2018)
23. Fung, B.: The embarrassing reason behind Amazons huge cloud computing outage this week. The Washington Post, Washington, DC (2017)
24. Gawlick, D., Kinkade, D.: Varieties of concurrency control in IMS/VS fast path. IEEE Database Eng. Bull. **8**, 3–10 (1985)
25. Gifford, D.K.: Weighted voting for replicated data. In: Proceedings of the Seventh ACM Symposium on Operating Systems Principles, ACM, New York, NY, SOSP '79, pp 150–162 (1979). https://doi.org/10.1145/800215.806583
26. Gray, J.: Notes on data base operating systems. In: Operating Systems, An Advanced Course. Springer, Berlin, pp. 393–481 (1978)
27. Gray, J.: The transaction concept: virtues and limitations (invited paper). In: VLDB, pp. 144–154 (1981)
28. Gray, J.: A Comparison of the Byzantine Agreement Problem and the Transaction Commit Problem, pp. 10–17. Springer, New York (1990)
29. Gray, J., Lamport, L.: Consens. Trans. Commit. ACM TODS **31**(1), 133–160 (2006)
30. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques, 1st edn. Morgan Kaufmann Publishers Inc., Burlington (1992)
31. Guerraoui, R.: Revisiting the Relationship Between Non-blocking Atomic Commitment and Consensus, pp. 87–100. Springer, Berlin (1995)
32. Guerraoui, R., Larrea, M., Schiper, A.: Reducing the Cost for Non-blocking in Atomic Commitment. In: IEEE Proceedings of 16th International Conference on Distributed Computing Systems, pp. 692–697 (1996)
33. Gupta, S., Sadoghi, M.: Blockchain Transaction Processing, pp. 1–11. Springer, Cham (2018a)
34. Gupta, S., Sadoghi, M.: EasyCommit: A non-blocking two-phase commit protocol. In: Proceedings of the 21st International Conference on Extending Database Technology, Open Proceedings, EDBT (2018b)
35. Harding, R., Van Aken, D., Pavlo, A., Stonebraker, M.: An evaluation of distributed concurrency control. Proc VLDB Endow **10**(5), 553–564 (2017)
36. Haritsa, J.R., Ramamritham, K., Gupta, R.: The PROMPT real-time commit protocol. IEEE TPDS **11**(2), 160–181 (2000)
37. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM TOPLAS **12**(3), 463–492 (1990)
38. Jiménez-Peris, R., Patiño Martínez, M., Alonso, G., Arévalo, S.: A low-latency non-blocking commit service. Springer, Berlin DISC'01 (2001)
39. Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S.B., Jones, E.P.C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., Abadi, D.J.: H-store: a high-performance, distributed main memory transaction processing system. PVLDB **1**, 1496–1499 (2008)
40. Lamport, L.: The part-time parliament. ACM Trans Comput Syst **16**(2), 133–169 (1998)
41. Levy, E., Korth, H.F., Silberschatz, A.: An optimistic commit protocol for distributed transaction management. In: ACM SIGMOD, ACM, pp. 88–97 (1991)
42. Lin, Q., Chang, P., Chen, G., Ooi, B.C., Tan, K.L., Wang, Z.: Towards a non-2PC transaction management in distributed database systems. In: Proceedings of the 2016 International Conference on Management of Data, ACM, New York, NY, SIGMOD '16, pp 1659–1674 (2016). https://doi.org/10.1145/2882903.2882923

43. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Stronger semantics for low-latency geo-replicated storage. In: USENIX Association, NSDI, pp. 313–328 (2013)
44. Mahmoud, H., Nawab, F., Pucher, A., Agrawal, D., El Abbadi, A.: Low-latency multi-datacenter databases using replicated commit. Proc VLDB Endow **6**(9), 661–672 (2013). https://doi.org/10.14778/2536360.2536366
45. Mahmoud, H.A., Arora, V., Nawab, F., Agrawal, D., El Abbadi, A.: MaaT: effective and scalable coordination of distributed transactions in the cloud. Proc VLDB Endow **7**(5), 329–340 (2014). https://doi.org/10.14778/2732269.2732270
46. Mao, Y., Junqueira, F.P., Marzullo, K.: Mencius: building efficient replicated state machines for WANs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, USENIX Association, pp. 369–384 (2008)
47. MemSQL (2013). http://www.memsql.com
48. Mohan, C., Lindsay, B., Obermarck, R.: Transaction management in the R* distributed database management system. ACM TODS **11**(4), 378–396 (1986)
49. Nawab, F., Sadoghi, M.: Blockplane: A global-scale byzantizing middleware. In: Proceedings of the 35th IEEE International Conference on Data Engineering, IEEE, ICDE '19 (2019)
50. Nawab, F., Arora, V., Agrawal, D., El Abbadi, A.: Minimizing commit latency of transactions in geo-replicated data stores. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ACM, SIGMOD '15, pp 1279–1294 (2015)
51. NuoDB (2010). http://www.nuodb.com
52. O'Brien, S.A.: Facebook. Instagram experience outages Saturday. CNN, GA, USA (2017)
53. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, USENIX Association, USENIX ATC'14, pp. 305–320 (2014)
54. Oracle, C.: Oracle 9i real application clusters concepts release 2 (9.2), Part Number A96597-01 (2002)
55. Ozsu, M.T., Valduriez, P.: Principles of Distributed Database Systems, 3rd edn. Springer, New York (2011)
56. Park, T., Yeom, H.Y.: A distributed group commit protocol for distributed database systems. ICPADS (1991)
57. Patterson, S., Elmore, A.J., Nawab, F., Agrawal, D., El Abbadi, A.: Serializability, not serial: concurrency control and availability in multi-datacenter datastores. Proc VLDB Endow **5**(11), (2012)
58. Pavlo, A., Curino, C., Zdonik, S.: Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In: ACM, SIGMOD '12, pp. 61–72 (2012)
59. Peng, D., Dabek, F.: Large-scale incremental processing using distributed transactions and notifications. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, USENIX Association, Berkeley, CA, OSDI'10, pp. 251–264 (2010)
60. Qadah, T.M., Sadoghi, M.: QueCC: a queue-oriented, control-free concurrency architecture. In: Proceedings of the 19th International Middleware Conference, ACM, New York, NY, Middleware '18, pp 13–25, (2018). https://doi.org/10.1145/3274808.3274810
61. Reddy, P.K., Kitsuregawa, M.: Reducing the blocking in two-phase commit protocol employing backup sites. In: IEEE, COOPIS'98, pp. 406–416 (1998)
62. Sadoghi, M., Blanas, S.: Transaction processing on modern hardware. Synth. Lect. Data Manag. **14**(2), 1–138 (2019). https://doi.org/10.2200/S00896ED1V01Y201901DTM058
63. Sadoghi, M., Ross, K.A., Canim, M., Bhattacharjee, B.: Making updates disk-I/O friendly using SSDs. Proc VLDB Endow **6**(11), 997–1008 (2013)
64. Sadoghi, M., Canim, M., Bhattacharjee, B., Nagel, F., Ross, K.A.: Reducing database locking contention through multi-version concurrency. Proc VLDB Endow **7**(13), 1331–1342 (2014)
65. Sadoghi, M., Bhattacherjee, S., Bhattacharjee, B., Canim, M.: L-Store: A real-time OLTP and OLAP system (2018). http://www.OpenProceeding.org, EDBT
66. Samaras, G., Britton, K., Citron, A., Mohan, C.: Two-phase commit optimizations in a commercial distributed environment. Distrib. Parallel Databases **3**(4), 325–360 (1995)
67. Shute, J., Vingralek, R., Samwel, B., Handy, B., Whipkey, C., Rollins, E., Oancea, M., Littleeld, K., Menestrina, D., Ellner, S., Apte, H.: F1: A distributed sql database that scales. In: VLDB (2013)
68. Skeen, D.: Nonblocking commit protocols. In: ACM, SIGMOD, pp. 133–142 (1981)
69. Skeen, D.: A quorum-based commit protocol. Tech. rep. (1982)
70. Skeen, D., Stonebraker, M.: A formal model of crash recovery in a distributed system. IEEE Trans. Softw. Eng. **9**(3), 219–228 (1983)

71. Stamos, J., Cristian, F.: A low-cost atomic commit protocol. In: Proceedings of the 9th Symposium on Reliable Distributed Systems, IEEE, pp. 10–17 (1990)
72. Stonebraker, M.: Concurrency control and consistency of multiple copies of data in distributed ingres. IEEE Trans. Softw. Eng. **SE–5**(3), 188–194 (1979). https://doi.org/10.1109/TSE.1979.234180
73. Stonebraker, M.: The case for shared nothing. Database Eng. **9**, 4–9 (1986)
74. Sulleyman, A.: Twitter down: social media app and website not working. The Independent, UK (2017)
75. Thomson, A., Diamond, T., Weng, S.C., Ren, K., Shao, P., Abadi, D.J.: Calvin: fast distributed transactions for partitioned database systems. In: SIGMOD (2012)
76. TiDB (2018). https://pingcap.com/en/
77. VoltDB (2010). https://www.voltdb.com/