Unifying Consensus and Atomic Commitment for Effective Cloud Data Management

Sujaya Maiyya* Faisal Nawab† Divyakant Agrawal* Amr El Abbadi*
*UC Santa Barbara, †UC Santa Cruz
*{sujaya_maiyya, divyagrawal, elabbadi}@ucsb.edu, †{fnawab@ucsc.edu}

ABSTRACT

Data storage in the Cloud needs to be scalable and fault-tolerant. Atomic commitment protocols such as Two Phase Commit (2PC) provide ACID guarantees for transactional access to sharded data and help in achieving scalability. Whereas consensus protocols such as Paxos consistently replicate data across different servers and provide fault tolerance. Cloud based datacenters today typically treat the problems of scalability and fault-tolerance disjointedly. In this work, we propose a unification of these two different paradigms into one framework called Consensus and Commitment (C&C) framework. The C&C framework can model existing and well known data management protocols as well as propose new ones. We demonstrate the advantages of the C&C framework by developing a new atomic commitment protocol, Paxos Atomic Commit (PAC), which integrates commitment with recovery in a Paxos-like manner. We also instantiate commit protocols from the C&C framework catered to different Cloud data management techniques. In particular, we propose a novel protocol, Generalized PAC (G-PAC) that integrates atomic commitment and fault tolerance in a cloud paradigm involving both sharding and replication of data. We compare the performance of G-PAC with a Spanner-like protocol, where 2PC is used at the logical data level and Paxos is used for consistent replication of logical data. The experimental results highlight the benefits of combining consensus along with commitment into a single integrated protocol.

PVLDB Reference Format:
Sujaya Maiyya, Faisal Nawab, Divyakant Agrawal, Amr El Abbadi. Unifying Consensus and Atomic Commitment for Effective Cloud Data Management. *PVLDB*, 12(5): 611-623, 2019.
DOI: https://doi.org/10.14778/3303753.3303765

1. INTRODUCTION

The emergent and persistent need for big data management and processing in the cloud has elicited substantial interest in scalable, fault-tolerant data management protocols. Scalability is usually achieved by partitioning or sharding the data across multiple servers. Fault-tolerance is achieved by replicating data on different servers, often, geographically distributed, to ensure recovery from catastrophic failures. Both the management of partitioned data as well as replicated data has been extensively studied for decades in the database and the distributed systems communities. In spite of many proposals to support relaxed notions of consistency across different partitions or replicas, the common wisdom for general purpose applications is to support strong consistency through atomic transactions [21, 9, 20, 27]. The gold standard for executing distributed transactions is two-phase commit (2PC). But 2PC is a blocking protocol even in the presence of mere site failures [9, 27, 20], which led to the three phase commit protocol (3PC) [27] that is non-blocking in the presence of crash failures. But the general version of 3PC is still blocking in the presence of partitioning failures [28].

On the other hand, the distributed systems and cloud computing community have fully embraced Paxos [16, 2] as an efficient asynchronous solution to support full state machine replication across different nodes. Paxos is a leader-based consensus protocol that tolerates crash failures and network partitions as long as majority of the nodes are accessible. As with all consensus protocols, Paxos cannot guarantee termination, in all executions, due to the FLP Impossibility result [6]. The rise of the Cloud paradigm has resulted in the emergence of various protocols that manage partitioned, replicated data sets [5, 22, 14, 8]. Most of these protocols use variants of 2PC for the atomic commitment of transactions and Paxos to support replication of both the data objects as well as the commitment decisions.

Given the need for scalable fault-tolerant data management, and the complex landscape of different protocols, their properties, assumptions as well as their similarities and subtle differences, there is a clear need for a unifying framework that unites and explains this plethora of commitment and consensus protocols. In this paper, we propose such a unifying framework: the Consensus and Commitment (C&C) framework. Starting with 2PC and Paxos, we propose a standard state machine model to unify the problems of commitment and consensus.

The unifying framework, C&C, makes several contributions. First, it demonstrates in an easy and intuitive manner that 2PC, Paxos and many other large scale data management protocols are in fact different instantiations of the same high level framework. C&C provides a framework to understand these different protocols by highlighting how

Proceedings of the VLDB Endowment, Vol. 12, No. 5
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3303753.3303765

they differ in the way each implements various phases of the framework. Furthermore, by proving the correctness of the framework, the correctness of the derived protocols is straightforward. The framework is thus both pedagogical as well as instructive.

Second, using the framework, we derive protocols that are either variants of existing protocols or completely novel, with interesting and significant performance characteristics. In particular, we derive several data management protocols for diverse cloud settings. Paxos Atomic Commitment (PAC) is a distributed atomic commitment protocol managing sharded but non-replicated data. PAC, which is a variant of 3PC, integrates crash recovery and normal operations seamlessly in a simple Paxos-like manner. We then derive Replicated-PAC (R-PAC) for fully replicated cloud data management, which is similar to Replicated Commit [22], and demonstrate that protocols like Google's Spanner [5] as well as Gray and Lamport's Paxos Commit [8] are also instantiations of the C&C framework. Finally we propose G-PAC, a novel protocol for sharded replicated architectures, which is similar to other recently proposed hybrid protocols, Janus [25] and TAPIR [32]. G-PAC integrates transaction commitment with the replication of data and reduces transaction commit latencies by avoiding the unnecessary layering of the different functionalities of commitment and consensus.

Many prior works have observed the similarities between the commitment and consensus problems. At the theoretical end, Guerraoui [10], Hadzilacos [11] and Charron-Bost [3] have investigated the relationship between the atomic commitment and consensus problems providing useful insights into the similarities and differences between them. Gray and Lamport [9] observe the similarities and then derive a hybrid protocol. In contrast, the main contribution of this paper is to *encapsulate* commitment and consensus in a *generic framework*, and then to derive diverse protocols to demonstrate the power of the abstractions presented in the framework. Many of these derived protocols are generalizations of existing protocols, however, some of them are novel in their own right and provide contrasting characteristics that are particularly relevant in modern cloud computing settings.

The paper is developed in a pedagogical manner. In Section 2, we explain 2PC and Paxos and lay the background for the framework. In Section 3 we propose the C&C unifying framework. This is followed in Section 4 by the derivation of a novel fault-tolerant, non-blocking commit protocol, PAC, in a sharding-only environment. In Section 5, we propose R-PAC for atomically committing transaction across fully-replicated data. Section 6 introduces G-PAC for managing data in a hybrid case of sharding and replication. In Section 7 we experimentally evaluate the performance of G-PAC and compare it with Spanner-like commit protocol. We discuss the related work in Section 8 and Section 9 concludes the paper, followed by Appendix in Section 10.

2. BACKGROUND

In this section, we provide an overview of 2PC and Paxos as representatives of consensus and atomic commit protocols, respectively. Our goal is to develop a unified framework for a multitude of protocols used in the cloud. In this section, we provide a simple state-machine representation of 2PC and Paxos. These state-machine representations are essential in our framework development later in the paper.

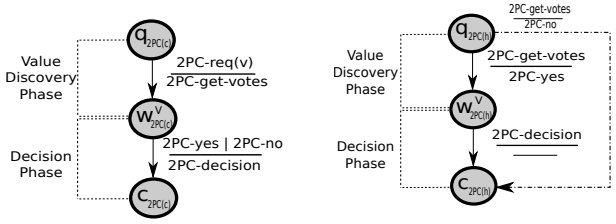
Algorithm 1 Given a set of *response values* sent by the cohorts, the coordinator chooses one value for the transaction based on the following conditions.

Possible values are: V = {2PC-yes, 2PC-no}.
Value Discovery: Method $\mathcal{V}$

```
1: Input: response values ⊂ V
2: if response values from all cohorts then
3:     if 2PC-no ∈ response values then
4:         value ← abort
5:     else
6:         value ← commit
7:     end if
8: else
9:     if Timer T times out then
10:    \* If a cohort crashed *\
11:        value ← abort
12:    end if
13: end if
```

2.1 Two Phase Commit

Two-Phase Commit (2PC) [9, 20] is an atomic commitment protocol. An atomic commitment protocol coordinates between data shards whether to commit or abort a transaction. Commitment protocols typically consist of a *coordinator* and a set of *cohorts*. The coordinator drives the commitment of a transaction. The cohorts vote on whether they agree to commit or decide to abort. We use the notion of a method $\mathcal{V}$ that takes as input all the votes of individual cohorts and decides on a final value for the transaction. Method $\mathcal{V}$ is represented in Algorithm 1.



(a) Coordinator **(b)** Cohort

Figure 1: State machine representation of 2PC.

We now present a state machine that represents 2PC. We show two distinct state machines, one for the coordinator that triggers the commitment protocol, and another for each cohort responding to the requests from the coordinator. In each state machine, a state is represented as a circle. An arrow from a state s^i to s^j with the label $\frac{e_{i,j}}{a_{i,j}}$ denotes that a transition from s^i to s^j is triggered by an event $e_{i,j}$ and causes an action $a_{i,j}$. Typically, events are received messages from clients or other nodes but may also be internal events such as timeouts or user-induced events. Actions are the messages that are generated in response to the transition, but may also denote state changes such as updating a variable. We use notations q as a starting state, followed by one or more waiting states, denoted by w and the final commit (representing both commit and abort) state, c.

Figure 1 shows the state machine representation of 2PC. In this work, we represent both transaction commit and

Algorithm 2 Given a set of responses sent by the acceptors, the leader decides whether to transit from one state to another based on the conditions explained below.

Possible responses: {pax-prepared, pax-accept}.

Leader election: Method $\mathcal{M}$

1: $Q_M \leftarrow$ *majority quorum*
2: **if** pax-prepared *messages from* Q_M **then**
3: return *true*
4: **else**
5: return *false*
6: **end if**

Replication: Method $\mathcal{R}$

1: $Q_M \leftarrow$ *majority quorum*
2: **if** pax-accept *messages from* Q_M **then**
3: return *true*
4: **else**
5: return *false*
6: **end if**

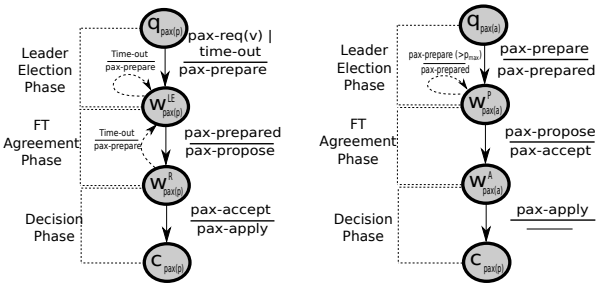
transaction abort as one final state, instead of having two different final states for commit and abort decisions as represented in [27]. The coordinator (Figure 1a) begins at the initial state $q_{2PC(C)}$ (the subscript $2PC(C)$ denotes a 2PC coordinator state). When a client request, 2PC-req[1], to end the transaction arrives, the coordinator sends 2PC-get-votes messages to all cohorts and enters a waiting state, $w^V_{2PC(C)}$. Once all cohorts respond, the responses are sent to method $\mathcal{V}$ represented in Algorithm 1 and a decision is made. The coordinator propagates 2PC-decision messages to all cohorts and reaches the final state $c_{2PC(C)}$. Although 2PC is proposed for an asynchronous system, in practice, if the coordinator does not hear back the value from a cohort after a time $\mathcal{T}$, the cohort is considered to be failed and the method $\mathcal{V}$ returns *abort*.

Figure 1b shows the state machine representation for a cohort. Initially, the cohort is in the initial state, $q_{2PC(h)}$ (the subscript $2PC(h)$ denotes a 2PC cohort state). If it receives a 2PC-get-votes message from a coordinator, it responds with a yes or no vote. A no vote is a unilateral decision, and therefore the cohort moves to the final state immediately with an abort decision. A yes vote will move the cohort to a waiting state, $w^V_{2PC(h)}$. In both cases, the cohort responds with its vote to the coordinator. While in $w^V_{2PC(h)}$ state, the cohort waits until it receives a 2PC-decision message from the coordinator and it moves to the final decision state of $c_{2PC(h)}$. 2PC can be blocking when there are crashes but we will not discuss it in this work.

2.2 Paxos Consensus Protocol

Paxos [16] is a consensus protocol that is often used to support fault-tolerance through replication. Consensus is the problem of reaching agreement on a single value between a set of nodes. To achieve this, Paxos adopts a leader-based approach. Figure 2 presents the state machine representation of Paxos: one for the process aspiring to be the leader, called a *proposer*, and another for each process receiving requests from the proposer, called an *acceptor*.

[1]The different protocols we discuss use similar terminology for messages with different meanings. To avoid confusion, we will use a prefix to denote the corresponding protocol.



(a) Proposer **(b)** Acceptor

Figure 2: State machine representation of Paxos.

Consider the proposer state machine in Figure 2a. The proposer is with an initial state $q_{pax(p)}$ (the subscript $pax(p)$ denotes a Paxos proposer state). A user request to execute value v, pax-req(v), triggers the Leader Election phase. pax-prepare messages are sent with the proposal number (initially 0) to at least a majority of acceptors. The proposer is then in state $w^{LE}_{pax(p)}$ waiting for pax-prepared messages. The condition to transition from $w^{LE}_{pax(p)}$ state to the next state is given by method $\mathcal{M}$ in Algorithm 2: once a majority of acceptors, denoted by *majority quorum* Q_M, respond with pax-prepared messages, the proposer moves to the Replication phase (state $w^R_{pax(p)}$), sending pax-propose messages to at least a majority of acceptors and waiting to receive enough pax-accept messages. To decide the completion of replication phase, the leader uses method $\mathcal{R}$ described in Algorithm 2, which requires a majority of pax-accept messages. The proposer then moves to the final commit state, denoting that the proposed value has been chosen, and pax-apply messages are sent to acceptors, notifying them of the outcome.

An unsuccessful Leader Election or Replication phase may be caused by a majority of acceptors not responding with pax-prepared or pax-accept messages. In these cases, a timeout is triggered (in either state $w^{LE}_{pax(p)}$ or $w^R_{pax(p)}$). In such an event, a new unique proposal number is picked that is larger than all proposal numbers received by the proposer. This process continues until the proposer successfully commits the value v.

Now, consider the acceptor state machine in Figure 2b. Initially, the acceptor is in an initial state $q_{pax(a)}$ (the subscript $pax(a)$ denotes a Paxos acceptor state). The acceptor maintains the highest promised proposal number, denoted p_{max}. An acceptor may receive a pax-prepare message which triggers responding with a pax-prepared message if the received ballot is greater than p_{max}.. After responding, the acceptor moves to state $w^P_{pax(a)}$ waiting for the next message from the leader. If the acceptor receives a pax-accept message with a proposal number that is greater or equal to p_{max}, then it moves to state $w^A_{pax(a)}$. Finally, the acceptor may receive a pax-apply message with the chosen value.

Note that, for presentation purposes, we omit reactions to events that do not change the state of the process. An example of such reactions is an acceptor responding to a pax-prepare or a pax-propose messages in the commit state. In case of a leader failure while executing Paxos, an acceptor will detect the failure using a timeout. This acceptor now tries to become the new leader, thus following the states shown in Figure 2a.

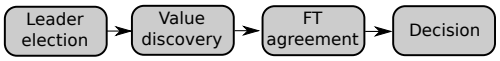


Figure 3: A high-level sequence of tasks that is generalized based on Paxos and 2PC.

3. UNIFYING CONSENSUS AND COMMITMENT

In this section, we present a Consensus and Commitment (C&C) framework that unifies Paxos and 2PC. This general framework can then be instantiated to describe a wide range of data management protocols, some of them well known, and others novel in their own right. We start by exploring a high level abstraction of Paxos and 2PC that will aid in developing the framework, and then derive the general C&C framework.

3.1 Abstracting Paxos and 2PC

In this section we deconstruct Paxos and 2PC into their basic tasks. Through this deconstruction we identify the tasks performed by both protocols that lead to the construction of the C&C framework.

Both consensus and atomic commit protocols aim at ensuring that *one* outcome is agreed upon in a distributed environment *while tolerating failures*. However, the conditions for achieving this agreement is different in the two cases. The basic phase transitions for Paxos are: leader election, followed by fault-tolerant replication of the value and finally the dissemination of the decision made by leader. For 2PC, considering a failure free situation, a predetermined coordinator requests the value to decide on from all cohorts, makes the decision based on the obtained values and disseminates the decision to all cohorts. We combine the above phases of the two protocols and derive a high level overview of the unified framework shown in Figure 3. Each of the phases shown in Figure 3 is described in detail explaining each phase, its significance and its derivation.

Phases of the C&C Framework

• ***Leader Election***: A normal operation in Paxos encompasses a leader election phase. 2PC, on the other hand, assumes a predesignated leader or a coordinator, and does not include a leader election process as part of normal operation. However, if the coordinator fails while committing a transaction, one way to terminate the transaction is by electing one of the live cohorts as a coordinator which tries to collect the states from other live nodes and attempts to terminate the transaction.

• ***Value Discovery***: Both Paxos and 2PC are used for reaching *agreement* in a distributed system. In Paxos, agreement is on arbitrary values provided by a client, while in 2PC agreement is on the outcome of a transaction. The decision in 2PC relies on the state of the cohorts and hence requires communication with the cohorts. This typically constitutes the first phase of 2PC. Whereas in Paxos, although it is agnostic to the process of deriving the value and the chosen value is independent of the current state of acceptors, it does incorporate value discovery during re-election. The response to a leader election message inherently includes a previously accepted value and the new leader should choose that value, in order to ensure the safety of a previously decided value.

• ***Fault-Tolerant Agreement***: Fault tolerance is a key feature that has to be ensured by all atomic commitment and consensus protocols. In the most naive approach, 2PC provides fault tolerance by persisting the decision to a log on the hard disk and recovering from the logs after a crash [19]. In Paxos, the role of the replication phase is essentially to guarantee fault tolerance by ensuring that the value chosen by the leader is persistent even in the event of leader failure. As explained in Section 2.2, the value proposed by the leader will be stored in at least a majority of the nodes, thus ensuring fault-tolerance of the agreed upon value.

• ***Decision***: In Paxos, once the leader decides on the proposed value, it propagates the decision *asynchronously* to all the participants who *learn* the decision. Similarly in 2PC, once a decision is made, the coordinator disseminates the decision to all the cohorts. Essentially, in both protocols a value is decided by the leader based on specific conditions and that value (after made fault tolerant) is broadcast to all the remaining nodes in the system.

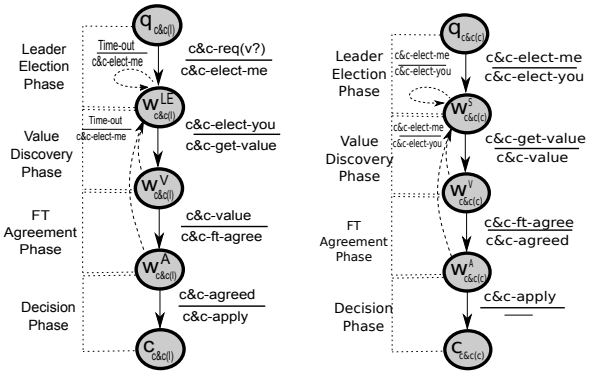
Given the task abstraction of the C&C framework, we can see that a Paxos instantiation of the framework, in normal operation, will lead to Leader Election, Fault-tolerant (FT) Agreement and Decision phases but will skip the additional Value Discovery phase. On the other hand, a 2PC instantiation of the C&C framework, in normal operation, will become a sequence of Value Discovery and Decision phase, avoiding an explicit Leader Election phase and FT-Agreement phase.

Although we highlighted the high-level similarities in the two protocols, there are subtle differences between the two problems of consensus and commitment. For example: the difference in the number of involved participants in both protocols: Paxos only needs a majority of nodes to be alive for a decision to be made whereas 2PC needs votes from all the participants to decide on the final value. Such subtleties in different protocols can be captured by specific protocol instantiations of the generic framework.

3.2 The C&C Framework

The Consensus and Commitment (C&C) framework aims to provide a general framework that represents both consensus and atomic commitment protocols. In Section 3.1, we started by unifying the two key protocols, Paxos and 2PC, and developed a high level abstraction of the unified framework. Now we expand the precise states in the C&C framework and the transitions across different states. Since our framework takes a centralized approach, each participating node in the system either behaves as a *leader* or a *cohort*. Figure 4 shows the state machines for a leader and a cohort in the framework. As mentioned earlier, an arrow from states s^i to s^j with the label $\frac{e_{i,j}}{a_{i,j}}$ denotes a transition from s^i to s^j. This transition is triggered by an event $e_{i,j}$ and the transition causes an action $a_{i,j}$. One important point to keep in mind is that each event $e_{i,j}$ and its corresponding action $a_{i,j}$ can have different meaning in different protocol instantiations.

We first define the typical behavior of a leader node in the C&C framework. As shown in Figure 4a, the leader node starts in the initial state $q_{c\&c(l)}$ (l indicates leader). A client sends c&c-req(v?) request to node $\mathcal{L}$. Depending on the protocol being instantiated, a client request may or may not contain the value v on which to reach agreement. In commitment protocols, the client request will be void



(a) Leader **(b)** Cohort

Figure 4: State machines of the C&C framework.

of the value. The leader $\mathcal{L}$ increments its ballot number b and starts *Leader Election* by sending c&c-elect-me messages containing b to all the cohorts. The leader waits in the $w^{LE}_{c\&c(l)}$ state for a majority of the cohorts to reply with c&c-elect-you messages. We model the above event as the method $\mathcal{M}$ as explained in Algorithm 2 which will return true when a majority of the cohorts vote for the contending leader. Once $\mathcal{L}$ receives votes from a majority, it starts *Value Discovery*. The c&c-elect-you message can contain previously accepted values by the cohorts, in which case $\mathcal{L}$ chooses one of these values. Otherwise, $\mathcal{L}$ sends c&c-get-value to all participants and transitions to the wait state $w^{V}_{c\&c(l)}$. The leader now waits to receive c&c-value messages from *all* the participants and since value discovery is derived from 2PC, C&C uses the method $\mathcal{V}$, explained in Algorithm 1, to decide on a value based on the c&c-value replies. Method $\mathcal{V}$ can be overridden depending on the requirements of the instantiating protocol (as will be shown in Section 4). The leader makes the chosen value fault-tolerant by starting *FT-Agreement* and sending out c&c-ft-agree messages to all nodes. $\mathcal{L}$ waits in the $w^{A}_{c\&c(l)}$ state until method $\mathcal{R}$ in Algorithm 2 (with c&c-agreed messages as input) returns *true*. In $\mathcal{R}$ we use majority quorum but an instantiated protocol can use any other quorum as long as the quorum in method $\mathcal{R}$ **intersects** with the quorum used in method $\mathcal{M}$. This follows from the generalizations proposed in [12, 24]. The leader $\mathcal{L}$ finally propagates the decision by sending c&c-apply messages and reaches the final state $c_{c\&c(l)}$.

Now we consider the typical behavior of a cohort in the C&C framework, as shown in Figure 4b. The cohort $\mathcal{C}$ starts in an initial state $q_{c\&c(c)}$ (c stands for cohort). After receiving a c&c-elect-me message from the leader $\mathcal{L}$, the cohort responds with c&c-elect-you upon verifying if ballot b sent by the leader is the largest ballot seen by $\mathcal{C}$. The c&c-elect-you response can also contain any value previously accepted by $\mathcal{C}$, if any. $\mathcal{C}$ then moves to the $w^{LE}_{c\&c(c)}$ state and waits for the new leader to trigger the next action. Typically, the cohort receives a c&c-get-value request from the leader. Each cohort independently chooses a value and then replies with a c&c-value message to $\mathcal{L}$. In atomic-commitment-like protocols, the value will be either *commit* or *abort* of the ongoing transaction. The cohort then waits in the $w^{V}_{c\&c(c)}$ state to hear back the value chosen by the leader. Upon receiving c&c-ft-agree, the cohort stores the value sent by leader and acknowledges its receipt to the leader by sending c&c-agreed message, and moving to $w^{A}_{c\&c(c)}$ state. Once fault-tolerance is achieved, the leader sends c&c-apply and the cohort applies the decided value and moves to the final state $c_{c\&c(c)}$.

A protocol instantiated from the framework can have either all the state transitions presented in Figure 4 or a subset of the states. Specific protocols can also merge two or more states for optimization (PAC undertakes this approach).

Safety in the C&C framework:

Any consensus or commitment protocol derived from the C&C framework should provide safety. **The safety condition states that a value once decided, will never be changed.** A protocol instantiated from the C&C framework will guarantee an overlap in the majority quorum used for Leader Election and the majority quorum used in Fault-Tolerant Agreement. This allows the new leader to learn any previously decided value, if any. We provide a detailed Safety Proof in the Appendix section 10 and show that the C&C framework is safe, and thus, we argue that any protocol instantiated from the framework is also safe.

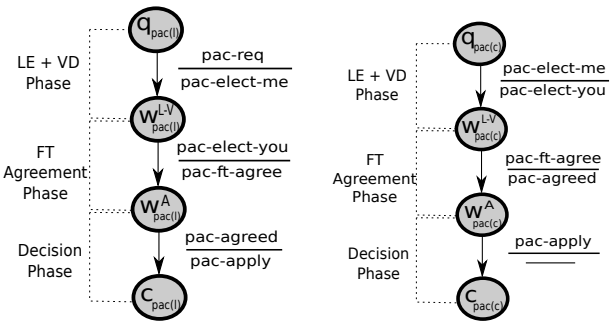
4. SHARDING-ONLY IN THE CLOUD

We now consider different data management techniques used in the cloud and derive commitment protocols in each scenario using the unified model. This section deals with the sharding only scenario where the data is sharded and there is no replication. When the data is partitioned, transactions can access data from more than one partition. To provide transactional atomicity, distributed atomic commitment protocols such as 2PC [9] and 3PC [27] are used. Since crash failures are frequent and 2PC can be blocking, 3PC was proposed as a non-blocking commitment protocol under crash failures. 3PC is traditionally presented using two modes of operation: 1) Normal mode (without any failures), 2) Termination mode (to terminate the ongoing transaction when a crash occurs) [1, 27]. 3PC is nonblocking if a majority (or a quorum) of sites are connected. However, Keidar and Dolev [13] show that 3PC may still suffer from blocking after a series of failures even if a quorum is connected. They develop a protocol, E3PC, that guarantees any majority of sites to terminate irrespective of the failure pattern. However, E3PC still requires both normal and failure mode operations. Inspired by the simplicity of Paxos to integrate the failure-free and crash- recovery cases in a single mode of operation, we use the C&C framework to derive an integrated atomic commitment protocol, PAC, which, similar to E3PC, is guaranteed to terminate as long as a majority of sites are connected irrespective of the failure pattern. The protocol presented in this section and the subsequent ones assume asynchronous networks.

4.1 System Model

Transactions accessing various shards consist of read and/or write operations on the data objects stored in one or more shards. The term *node* abstractly represents either a process or a server responsible for a subset of data. A key point to note here is that the protocol developed in this section (and the subsequent ones) is oblivious to the underlying concurrency control (CC) mechanism. We can use a pessimistic CC such as Two Phase Locking [9] or an optimistic CC technique [15]. The commit protocol is derived such that each data shard has a transaction manager and when the client

requests to end a transaction, the underlying transaction manager for each data shard decides to either commit or abort the transaction based on the chosen CC mechanism. Hence, the isolation and serializability guarantees depend on the deployed CC, and is orthogonal to the work presented here, which is focused on the atomic commitment of a *single* transaction, as is traditional with atomic commitment protocols.



(a) Leader **(b)** Cohort

Figure 5: State m/c representation of the PAC protocol.

Table 1: State variables for each process p in PAC.

BallotNum	initially $< 0, p >$
InitVal	*commit* or *abort*
AcceptNum	initially $< 0, p >$
AcceptVal	initially Null
Decision	initially False

4.2 Protocol

We now derive the **Paxos Atomic Commitment** (PAC) protocol from the C&C framework. Each committing transaction executes a single instance of PAC, *i.e.*, if there are several concurrent transactions committing at the same time, multiple concurrent PAC instances would be executing *independently*. In PAC, each node involved in a transaction T maintains the variables shown in Table 1, with their initial values, where p is the process id.

Figure 5 shows the state transition diagram for both the leader and a cohort in PAC. Abstractly, PAC follows the four phases of the C&C framework: Leader Election, Value Discovery, Fault-tolerant Agreement and the Decision Phases. However, as an optimization, Leader Election and Value Discovery are merged together. Furthermore, in the C&C framework, Leader Election needs response from a *majority* while Value Discovery requires response from *all* the nodes. Hence, the optimized merged phase in PAC needs to receive responses from *all* nodes in the initial round, while any subsequent (after failure) Leader Election may only need to receive response from a *majority*.

Unlike 3PC, in PAC, the external client can send an *end -transaction(T)* request to any shard accessed by T. Let $\mathcal{L}$ be the node that receives the request from the client. $\mathcal{L}$, starting with an initial ballot, tries to become the leader to atomically commit T. Note that in failure-free mode, there are no contending leaders unless the client does not hear from $\mathcal{L}$ for long-enough time and sends the *end_transaction-(T)* request to another shard in T. $\mathcal{L}$ increments its ballot number and sends pac-elect-me messages and moves to the $W^{L-V}_{pac(l)}$ state (l represents leader). A cohort, $\mathcal{C}$, starts in state $q_{pac(c)}$ (c for cohort). After receiving the election message, $\mathcal{C}$ responds with a pac-elect-you message only if $\mathcal{L}$'s ballot number is the highest that $\mathcal{C}$ has received. Based on the CC execution, each node, including $\mathcal{L}$, sets its *InitVal* with a decision *i.e.*, either commit or abort the transaction. The pac-elect-you response contains *InitVal*, any previously accepted value *AcceptVal*, and the corresponding ballot number *AcceptNum*, along with the *Decision* variable (the initial values are defined in Table 1).

Algorithm 3 Given the pac-elect-you replies from the cohorts, the leader chooses a value for the transaction based on the conditions presented here.

Leader Election + Value Discovery: Method $\mathcal{V}$

The replies contain variables defined in Table 1.

```
1: if received response from a MAJORITY then
2:     if at least ONE response with Decision=True then
3:         AcceptVal ← AcceptVal of that response
4:         Decision ← True
5:     else if at least one response with AcceptVal=commit
6:         \* Decision is True for none.*\ then
7:         AcceptVal ← commit
8:     else if received response from ALL cohorts then
9:     \* The normal operation case *\
10:        if all InitVal = commit then
11:            AcceptVal ← commit
12:        else
13:            AcceptVal ← abort
14:        end if
15:    else
16:        AcceptVal ← abort
17:    end if
18: else transaction is blocked
19: end if
```

The transition conditions for $\mathcal{L}$ to move from $W^{L-V}_{pac(l)}$ to $W^{A}_{pac(l)}$ are shown in method $\mathcal{V}$ in Algorithm 3. Once $\mathcal{L}$ receives pac-elect-you messages from a majority of cohorts, it is elected as leader, based on the Leader Election phase of the C&C framework. If none of the responses had *Decision* value true or *AcceptVal* value set, then this corresponds to Value Discovery phase where $\mathcal{L}$ has to wait till it hears pac-elect-you from ALL the cohorts to check if any cohort has decided to abort the transaction. If any *one* cohort replies with *InitVal* as abort, $\mathcal{L}$ chooses abort as *AcceptVal*, and commit otherwise. We will describe crash recovery later. The leader then propagates pac-ft-agree message with the chosen *AcceptVal* to all the cohorts and starts the fault-tolerant agreement phase Each cohort upon receiving pac-ft-agree message validates the ballot number and updates the local *AcceptVal* to the value chosen by the leader. It then responds to the leader with pac-ft-agreed message, thus moving to $W^{A}_{pac(c)}$ state.

The leader waits to hear back pac-ft-agreed message from only a majority, as explained in method $\mathcal{R}$ in Algorithm 2 but with pac-ft-agreed messages as input. After hearing back from a majority, the leader sets *Decision* to True, informs the client of the transaction decision and asynchronously sends

out pac-apply message with *Decision* as True to all cohorts, eventually reaching the final state $c_{pac(l)}$. A cohort $\mathcal{C}$ can receive pac-apply message when it is in either $W^{L-V}_{pac(c)}$ or $W^{A}_{pac(c)}$ states, upon which it will update its local *Decision* and *AcceptVal* variables and applies the changes made by the transaction to the data shard that $\mathcal{C}$ is responsible for.

In case of leader failure while executing PAC, the recovery is similar to Paxos (Section 2.2). A cohort will detect the failure using a timeout and sends out pac-elect-me to all the live nodes and will become the new leader upon receiving a majority of pac-elect-you. The value to be selected by the leader depends on the obtained pac-elect-you messages, as described in method $\mathcal{V}$ in Algorithm 3. If any node, say $\mathcal{N}$ replies with *Decision* as True, this implies that the previous leader had made the decision and propagated it to at least one node before crashing; so the new leader will choose the value sent by $\mathcal{N}$. If none of the replies has *Decision* as True but at least one of them has *AcceptVal* as commit, this implies that the previous leader obtained replies from all and had chosen commit and sent out pac-ft-agree messages. Hence, the new leader will choose commit. In all the other cases, abort is chosen. If the new leader does not get a majority of pac-elect-you, then the protocol is blocked. The subsequent phases of replication and decision follow the states shown in Figure 5.

5. REPLICATION-ONLY IN THE CLOUD

In this section, we explore a data management technique that deals with fully replicated data *i.e.*, the data is fully replicated across, potentially different, data-centers. Using the unified C&C framework, we derive a commit protocol, similar to PAC, called Replicated-PAC (R-PAC).

5.1 System Model

In a traditional state machine replication (SMR) system, the client proposes a value and all replicas try to reach agreement on that value. In a fully replicated data management system, each node in the system maintains an identical copy of the data. Clients perform transactional accesses on the data. Here, the abstraction of a node can represent a datacenter or a single server; but all entities denoted as nodes handle identical data. At transaction commitment, each node independently decides whether to commit or to abort the transaction. The transactions can span multiple data objects in each node and any updates on the data by a transaction will be executed atomically. Since the data is fully replicated, R-PAC is comparable to the Replicated Commit protocol [22].

Every node runs a concurrency control (CC) protocol and provides a commitment value for a transaction. If a node is an abstraction for a single machine, we can have a CC, such as 2PL, that decides if a transaction can be atomically committed or if it has to be aborted. Whereas, if a node represents a datacenter and if data is partitioned across different machines within the datacenter, the commitment value per node can be obtained in two ways. In the first approach with shared-nothing architecture, each data center has a transaction manager which internally performs a distributed atomic commitment such as 2PC or PAC (Section 4) and provide a single value for that node (datacenter). In the second approach with a shared-storage architecture, different machines can access the same storage driver and detect any concurrency violations [1]. In either architecture, each node provides a *single* value per transaction. For simplicity, we do not delve deeper into the ways of providing CC; rather we work with the abstraction that when the leader asks for a commitment value of a transaction, each *node* provides one value. The protocol presented below ensures that all the nodes either atomically commit or abort the transaction, thus maintaining a consistent view of data at all nodes.

5.2 Protocol

The commit protocol, Replicated-PAC (R-PAC), is similar to PAC except for one change: the Value Discovery method $\mathcal{V}$. In PAC (Section 4), during value discovery, the leader waits to receive pac-elect-you message from *all* cohorts. When the data is partitioned, a commit decision cannot be made until all cohorts vote because each cohort is responsible for a disjoint set of data. In the replication-only case, since all nodes maintain identical data, the leader need to only wait for replies from a *majority* of replicas that have the **same** *InitVal*. Hence, the method $\mathcal{V}$ presented in Algorithm 3 differs for R-PAC only at line 8, namely waiting for pac-elect-you messages from a *majority* rather than *all* cohorts. Since R-PAC only requires a majority of replicas to respond, it is similar to the original majority consensus protocol proposed by Thomas [29]. The rest of the replication phase, decision phase and in case of a crash, the recovery mechanism, are identical to PAC.

Depending on the CC mechanism adopted, and due to the asynchrony of the system, different replicas can end up choosing different commitment values. A key observation here is that if a majority of the replicas choose to commit and one or more replicas in the minority choose to abort the transaction, the leader forces *ALL* replicas to commit the transaction. This does not violate the isolation and serializabilty guarantees of the CC protocol, as updates will not be reflected on the data of a replica R until R receives a pac-apply message. This ensures that all replicas have consistent views of the data at the end of each transaction.

6. SHARDING + REPLICATION IN THE CLOUD

In this section, we present a general hybrid of the two previously presented data management schemes *i.e.*, data is both partitioned across different shards and each shard is replicated across different nodes. Transactions can span multiple shards, and any update of a shard will cause an update of its replicas. When the data is both sharded and replicated, solutions like Spanner and MDCC [5, 14] use a hierarchical approach by horizontally sharding the data and vertically replicating each shard onto replicas. The replication is managed by servers called *shard leaders*. The other category of solutions, such as Janus [25] and TAPIR [32] deconstruct the hierarchy of shards and replicas and atomically access data from all the involved nodes. Hence, we categorize the hybrid case of sharded and replicated data into two different classes based on the type of replication: 1) Using standard State Machine Replication (SMR) wherein the coordinator communicates only with the leaders of each shard, 2) Using PAC-like protocol wherein the coordinator of a transaction communicates with all the involved nodes.

6.1 Replication using standard SMR: Layered architecture

A popular approach for providing non-blocking behavior to a commitment protocol such as 2PC is to replicate each state of a participating node (coordinator or cohort). SMR ensures fault-tolerance by replicating each 2PC state of a shard to a set of replica nodes. This has been adopted by many systems including Spanner [5] and others [7, 22, 14]. We will refer to this approach as 2PC/SMR. 2PC/SMR shares the objective of 3PC which is to make 2PC fault-tolerant. **While SMR uses replicas to provide fault tolerance, 3PC uses participants to provide persistence of decision**. Therefore, 2PC/SMR can be considered as an alternative to commitment protocols that provides fault tolerance using as additional phase such as 3PC or PAC. 2PC/SMR follows the abstraction defined by the C&C framework. In particular, 2PC provides the Value Discovery phase of C&C and any SMR protocol, such as Paxos, provides the Fault-Tolerant Agreement phase of the framework. The Decision phase of C&C is propagated hierarchically by the coordinator to SMR leaders and then the leaders propagate the decision on to the replicas.

The system model of 2PC/SMR consists of a coordinator and a set of cohorts, each responsible for a data shard. Along with that, 2PC/SMR also introduces *SMR replicas*. SMR replicas are not involved in the 2PC protocol. Rather, they serve as backups for the coordinator and cohorts and are used to implement the FT-Agreement phase of C&C. The coordinator and cohorts each have a set of—potentially overlapping—SMR replicas, the idea originally proposed by Gray and Lamport in [8]. If a shard holder (coordinator or cohort) fails, the associated SMR replicas recover the state of the failed shard. This changes the state transitions of 2PC. At a high level, every state change in 2PC is transformed into two state changes: one to replicate the state to the associated SMR replicas and another to make the transition to the next 2PC state. For example, before a cohort in 2PC responds to 2PC-get-votes, it replicates its value onto a majority of SMR replicas. Similarly, the coordinator, after making a decision, first replicates it on a majority of SMR replicas before responding to the client or informing other cohorts.

The advantage of using 2PC/SMR in building a system is that if the underlying SMR replication is in place, it is easy to leverage the SMR system to derive a fault-tolerant commitment protocol using 2PC. Google's Spanner is one such example. We discuss the trade-offs in terms of number of communication rounds for a layered solution vs. flattened solution in the evaluation Section 7.

6.2 Replication using Generalized PAC: Flattened architecture

In this section, we propose a novel integrated approach for SMR in environments with both sharding and replication. Our approach is a generalized version of PAC, hence is named Generalized PAC or G-PAC. The main motivation driving our proposal is to reduce the number of wide-area communication messages. One such opportunity stems from the hierarchy that is imposed in traditional SMR systems—such as Spanner. The hierarchy of traditional SMR systems incur wide-area communication unnecessarily. This is because the 2PC (Atomic Commitment) layer and Paxos (consensus/replication) layers are operating independently from each other. Specifically, a wide-area communication message that is sent as part of the 2PC protocol can be used for Paxos (*e.g.*, leader election). We investigate this opportunity and propose G-PAC to find optimization opportunities between the Atomic Commit and Consensus layers.

The G-PAC protocol consists of three phases: an integrated Leader Election and Value Discovery phase, a Fault-Tolerant Agreement phase, followed by the Decision phase. If a transaction, T, accesses n shards and each shard is replicated in r servers, there are a total of $n * r$ servers that are involved in the transaction T. This set of servers will be referred as *participating servers*. The client chooses one of the participating servers, $\mathcal{L}$, and sends an *end_transaction(T)* request. $\mathcal{L}$ then tries to become the leader or the coordinator for transaction T. The coordinator, $\mathcal{L}$, and the cohorts follow the same state transition as shown in Figure 5 except that the contending leader sends plac-elect-me message to *all* the participating servers. The overridden Value Discovery method $\mathcal{V}$ is similar to the one presented Algorithm 3. The flattening of the architecture for sharding and replication changes the notion of *all* and *majority* cohorts that is referred in Algorithm 3 to:

- **super-set**: Given n shards, each with r replicas, super -set is a majority of replicas for each of the n shards. The value for each shard is the one chosen by a majority of replicas of that shard *i.e.*, $(\frac{r}{2}+1)$ replicas. If any shard (represented by a majority of its replicas) chooses to abort, the coordinator sets *AcceptVal* to abort.
- **super-majority**: a majority of replicas for a *majority* of shards involved in transaction T *i.e.*, $(\frac{n}{2}+1)$ shards and for each shard, a value is obtained when a majority of its replicas respond *i.e.*, $(\frac{r}{2}+1)$ replicas for each shard.

Algorithm 4 For G-PAC, given the plac-elect-you replies from the participating servers, the leader chooses a value for the transaction based on the conditions presented here.

Leader Election + Value Discovery: Method $\mathcal{V}$
The replies contain variables defined in Table 1.

```
1: if received response from a SUPER-MAJORITY then
2:     if at least ONE shard response has Decision=True
3:     then
4:         AcceptVal ← AcceptVal of that response
5:         Decision ← True
6:     else if at least one shard response has AcceptVal
7:     = commit  then \* Decision is True for none.*\
8:         AcceptVal ← commit
9:     else if received response from SUPER-SET
10:    \* The normal operation case *\ then
11:        if all InitVal = commit then
12:            AcceptVal ← commit
13:        else
14:            AcceptVal ← abort
15:        end if
16:    else
17:        AcceptVal ← abort
18:    end if
19: else transaction is blocked
20: end if
```

Method $\mathcal{V}$ with the newly defined notions of *super-set* and *super-majority*, which decides the value for the transaction, in described in Algorithm 4. We reuse the definition of replication method $\mathcal{R}$ described in Algorithm 2, where majority is replaced by *super-majority*. Note that during the integrated Leader Election and Value Discovery phase, if a node receives response from a *super-majority*, it could be elected as leader, however to proceed with Value Discovery, it needs to wait for a *super-set*, which is a more stringent condition due to the atomic commitment requirement.

7. EVALUATION

In our evaluations we compare the performance of G-PAC and 2PC/SMR and discuss the trade-offs in wide-area communication delays between the two approaches. The performance of 2PC/SMR varies widely based on the placement of the SMR leaders. When the SMR leaders are dispersed across different datacenters, the commitment of a transaction needs 4 inter-datacenter round-trip time (RTT) delays: two rounds for the two phases of 2PC and one round each for replicating each of those phases using multi-Paxos [2]. As an optimization, placing all the leaders in a single datacenter will reduce the inter-datacenter communication to 3 RTTs. G-PAC, on the other hand, always only needs 3 RTTs (one for each phase of G-PAC) to complete a transaction commitment.

The practical implications of using G-PAC is that in G-PAC, the leader should know not only the involved shards in a transaction, but also about their replicas. This requires the replicas to have additional information which either has to be stored as meta-data for each transaction or can be stored in a configuration file. If a replica is added/removed or if new shards are added, this will require a configuration change. Propagating this change can be challenging, potentially leading to additional overhead. Although this is practically challenging, many existing practical deployments deal with configuration changes using asynchronous but consistent roll-out based methods such as Raft [26] or Lamport proposals [18, 17]. Any such method can be adapted in the case of G-PAC.

Our experiments evaluate the performance of G-PAC with respect to two versions of 2PC/SMR: the most optimal one (collocated SMR leaders) and the worst-case (geographically distributed SMR leaders across different datacenters). Hence, on average, the performance of 2PC/SMR would lie in between the two cases. We compare the behavior of all the protocols by increasing the concurrent clients (or threads), each of which generates transactions sequentially. We leveraged Amazon EC2 machines from 5 different datacenters for the experiments. The datacenters used were N.California (C), N.Virginia (V), Ireland (I), Sao Paolo (SP) and Tokyo (T). In what follows we use the capitalized first initial of each datacenter as its label. Cross datacenter round trip latencies are shown in Table 2. In these experiments, we used compute optimized EC2 c4.large machines with 2 vCPUs and 3.75 GiB RAM.

Although G-PAC can be built upon any concurrency control, for equivalent comparison, both G-PAC and 2PC/SMR implementations used **Two Phase Locking** (2PL) as a concurrency control technique. In 2PC/SMR, only the shard leaders maintain the lock table; whereas in G-PAC, all participating servers maintain their own lock tables. Both protocols execute the decision phase asynchronously.

Table 2: RTT latencies across different datacenters in ms.

	V	I	SP	T
C	60.3	150	201	111
V	-	74.4	139	171
I	-	-	183	223
SP	-	-	-	269

7.1 Micro Benchmark

As the first step in our evaluation, we performed the experiments using a transactional YCSB-like [4] micro- benchmark that generated read-write transactions. Every operation within a transaction accessed different keys, thus, generating multi-record transactional workloads. Each shard consisted of 10000 data items. Every run generated 2500 transactions; each plotted data point is an average of 3 runs. To imitate real global data access patterns, the transactions perform a skewed data access *i.e*, 90% of the transactions access 10% of the data objects, while the remaining transactions access the other 90% of the data items.

7.1.1 Varying number of shards

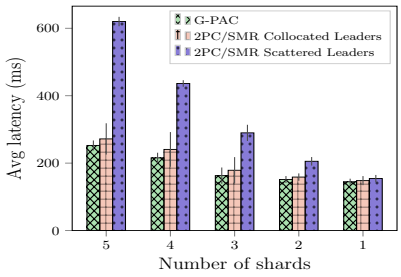
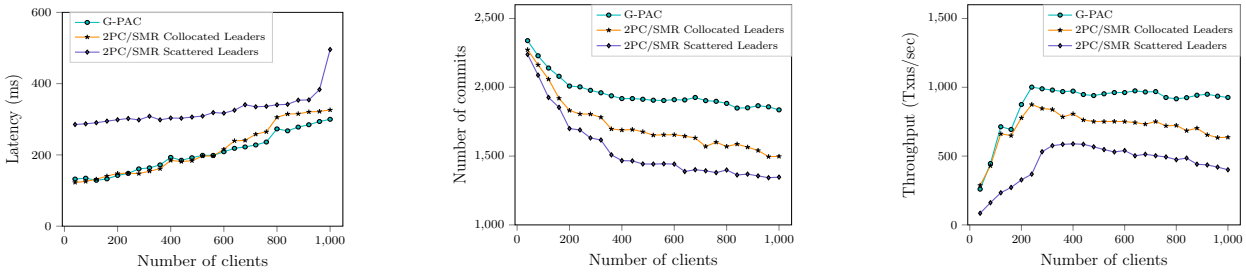


Figure 6: Commit latency vs. number of shards.

Commit latency, as measured by the client, is the time taken to commit a transaction once the client sends an *end_transaction* request. As the first set of experiments, we assessed the average commit latency of transactions when the transactions spanned increasing number of shards. In this experiment, both the coordinator and the client are located on datacenter C. We measure the average latencies for the three protocols by increasing the number of shards accessed by transactions from 1 to 5. And the data was **partially replicated** *i.e.,* each shard was replicated in only 3 datacenters. The results are depicted in Figure 6. When the data objects accessed by the transactions are from all 5 shards (at datacenters C, V, I, SP and T), 2PC/ SMR with leaders scattered on 5 different datacenters, has the highest commit latency, as the coordinator is required to communicate with geo-distributed shard leaders. For 2PC/SMR with scattered leaders, the average commit latency decreases with the reduction in the number of involved shards. This is because with each reduction in number of shards, we removed the farthest datacenter from the experiment. Whereas, the average commit latency for G-PAC does not increase substantially with increasing shards as it communicates only with the closest replicas of each shard, before responding to the client. When the clients access data from a single shard, the average latencies for all three protocols converge almost to the same value. This is because with a single shard, all



(a) Commit latency. **(b)** Number of commits. **(c)** Throughput.

Figure 7: Various performance evaluations using TPC-C benchmark

three protocols need to communicate with only a majority of the 3 replicas for the shard.

This experiment not only shows that G-PAC has highly stabilized performance when the number of involved shards increase, but it also highlights the fact that, with 2PC/SMR with collocated leaders, at least one datacenter must be overloaded with all shards in order to obtain optimal results. This is in contrast to G-PAC which equally distributes the load on all datacenters, while preserving optimal latency.

From the results shown in Figure 6, we choose 3 shards to run each of the experiments that follow, as it is representative of the trade-offs offered by the three protocols.

7.2 TPC-C Benchmark

As a next step, we evaluate G-PAC using TPC-C benchmark, which is a standard for benchmarking OLTP systems. TPC-C includes 9 different tables and 5 types of transactions that access the data stored in the tables. There are 3 read-write transactions and 2 read-only transactions in the benchmark. In our evaluation, we used 3 warehouses, each with 10 districts, each of which in-turn maintained 3000 customer details (as dictated by the spec). One change we adapted was to allocate disjoint sets of items to different warehouses, as the overall goal is to evaluate the protocols for distributed, multi-record transactions. Hence, each warehouse consisted of 33,333 items and the New Order transaction (which consisted of 70% of the requests) accessed items from all 3 warehouses. Each run consisted of 2500 transactions and every data point presented in all the experiments is an average of three runs.

We measured various aspects of the performance of G-PAC and contrasted it with the two variations of 2PC/SMR. We used AWS on 3 different datacenters for the following experiments: C, V and I. In 2PC/SMR with dispersed leaders, the three shard leaders are placed on 3 different datacenters. And for 2PC/SMR with collocated leaders, all shard leaders were placed in datacenter C. Although not required, for ease of evaluation, each shard was replicated across all three datacenters.

7.2.1 Commit Latency

In this experiment, we measure the commit latencies for G-PAC and the two versions of 2PC/SMR while increasing the number of concurrent clients from 20 to 1000. The results are shown in Figure 7a. Both G-PAC and the optimized 2PC/SMR respond to the client after two RTTs (as the decision is sent asynchronously to replicas), and hence, both protocols start off with almost the same latency values for lower concurrency levels. But with high concurrency, 2PC/SMR has higher latency as all commitments need to go through the leaders, which can become a bottleneck for highly concurrent workloads. 2PC/SMR with dispersed leaders is the least efficient with the highest latency. This is because the coordinator of each transaction needs at least one round of communication with all geo-distributed leaders for the first phase of 2PC (2PC-get-value and 2PC-value). Hence, we observed that G-PAC, when compared to the most and the least performance efficient versions of 2PC/SMR, provides the lowest commit latency of the three.

7.2.2 Number of Commits

In this set of experiments, we measured the total number of committed transactions out of 2500 transactions by all three protocols while increasing the number of concurrent clients from 20 to 1000. The results are shown in Figure 7b. From the graph, we observe that G-PAC commits, on an average, 15.58% more transactions than 2PC/SMR with collocated leaders and and 32.57% more than 2PC/SMR with scattered leaders.

Both G-PAC and 2PC/SMR implemented 2-Phase Locking for concurrency control among contending transactions. The locks acquired by one transaction are released after the decision is propagated by the coordinator of the transaction. The leader based layered architecture of 2PC/ SMR, and its disjoint phases of commitment and consensus, takes an additional round-trip communication before it can release the locks, as compared to G-PAC. And in 2PC/SMR, since lock tables are maintained only at the leaders, at higher contention, more transactions end up aborted. Hence, this experiment shows that flattening the architecture by one level and off-loading the concurrency overhead to all the replicas, G-PAC can release the locks obtained by a transaction earlier than 2PC/SMR, and thus can commit more transactions than 2PC/SMR.

7.2.3 Throughput

We next show the throughput measurements for G-PAC and 2PC/SMR. Throughput is measured as number of successful transaction commits per second and hence, the high contention of data access affects the throughput of the system. Figure 7c shows the performance as measured by transactions executed per second with increasing number of concurrent clients. G-PAC provides 27.37% more throughput on an average than 2PC/SMR with collocated leaders and 88.91% higher throughput than 2PC/SMR with scattered leaders, thus indicating that G-PAC has significantly higher throughput than 2PC/SMR. Although Figure 7a showed similar commit latencies for G-PAC and optimized

2PC/SMR, the throughput difference between the two is large. This behavior is due to lower number of successful transaction commits for 2PC/SMR, as seen in Figure 7b. The scattered leader approach for 2PC/SMR provides low throughput due to larger commit delays. The lower latencies along with greater number of successful transactions boosts the throughput of G-PAC as compared to 2PC/SMR.

7.2.4 Latency of each phase in G-PAC

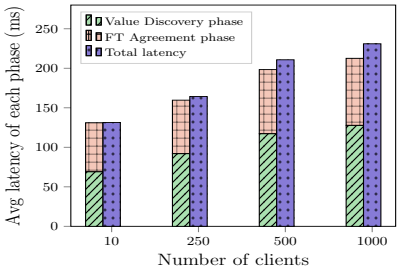


Figure 8: Latency of each phase in G-PAC

G-PAC consists of 3 phases: Value Discovery, Fault Tolerant Agreement (essentially replicating the decision) and the Decision phase. In our implementation, the decision is first disseminated to the client, and then asynchronously sent to the cohorts. In this experiment, we show the breakdown of the commit latency and analyze the time spent during each phase of G-PAC. Figure 8 shows the average latency spent during each phase, as well as the overall commit latency, with low to high concurrency. The results indicate that the majority of the time is spent on Value Discovery phase (which requires response from super-set of replicas as well as involves acquiring locks using 2PL) and the FT-Agreement time is quite consistent throughout the experiment (which needs responses only from super-majority and does not involve locking). The increased concurrency adds additional delays to the overall commit latency.

7.3 Availability Analysis

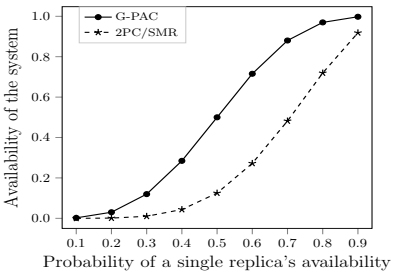


Figure 9: Availability Analysis

In this section, we perform a simple availability analysis to better understand how the availability of G-PAC and 2PC/SMR vary with the availability of individual replicas. Let p be the probability with which an individual replica is available. Consider a system involving three shards, where each shard is replicated three ways ($2*f+1$ with $f = 1$). For the Value Discovery phase, both protocols require a majority of replicas from *all* the shards, hence there is no difference in their availability. But to replicate the decision, G-PAC needs only a super-majority (majority of majority) while 2PC/SMR requires a majority from *all* the shards.

We mathematically represent the availability of both the protocols. First, the availability of each shard (each of which is replicated 3 ways) is computed as shown in Equation 1: either all 3 replicas of the shard are available or a majority of the replicas are available. Second, based on the probability, p_{shard}, of each shard being available, we derive the availability of the two protocols. In G-PAC, the decision needs to be replicated in a majority of shards. Provided there are 3 shards, the availability of G-PAC is given by Equation 2: either all shards are alive or a majority of them are alive. Similarly, the availability of 2PC/SMR is given by Equation 3 which indicates that *all* shards need to be alive to replicate the decision in 2PC/SMR.

$$p_{shard} = 3C_3 * p^3 + 3C_2 p^2 * (1-p) \quad (1)$$

$$A_{G\_PAC} = 3C_3 * p_{shard}^3 + 3C_2 p_{shard}^2 * (1-p_{shard}) \quad (2)$$

$$A_{2PC/SMR} = 3C_3 * p_{shard}^3 \quad (3)$$

Figure 9, shows the availability of G-PAC and 2PC/SMR with increasing probability of an individual site being available. The analysis indicates that G-PAC has higher tolerance to failures than 2PC/SMR. In particular, G-PAC achieves four nines of availability (i.e., 99.99%) when each replica is available with probability $p = 0.96$, where as to achieve the same, 2PC/SMR requires each replica to be available with probability $p = 0.997$.

8. RELATED WORK

Distributed transaction management and replication protocols have been studied extensively [1, 31, 16, 23]. These works involve many different aspects, including but not restricted to concurrency control, recovery, quorums, commitment etc. Our focus in this paper has been on the particular aspect of transaction commitment and how this relates to the consensus problem. The other aspects are often orthogonal to the actual mechanisms of how the transaction commits. Furthermore, the renewed interest in this field has been driven by the recent adoption of replication and transaction management in large scale cloud enterprises. Hence, in this section we focus on commitment and consensus protocols that are specifically appropriate for Cloud settings that require both sharding and replication, and contrast them with some of the proposed protocols derived from C&C.

We start-off by discussing one of the early and landmark solutions for integrating Paxos and Atomic Commitment, namely, Paxos Commit, proposed by Gray and Lamport [8]. Abstractly, Paxos Commit uses Paxos to fault-tolerantly store the commitment state of 2PC on multiple replicas. Paxos Commit optimizes on the number of message exchanges by collocating multiple replicas on the same node. This is quite similar to the 2PC/SMR protocol of Section 6.1.

Google Spanner [5] adapts an approach similar to Paxos Commit to perform transactional commitment but unlike Paxos Commit, Spanner replicates on geo-distributed servers. 2PC/SMR, developed in Section 6.1 is a high level abstraction of Spanner. Replicated Commit by Mahmoud et al. [22] is a commit protocol that is comparable to Spanner, but unlike Spanner, it assumes full replication of data. Replicated Commit can also be viewed as an instance of the C&C framework, as it can be materialized from the R-PAC

protocol (Section 5). MDCC by Kraska et al. [14] is another commit protocol for geo-replicated transactions. MDCC guarantees commitment in one cross datacenter round trip for a collision free transaction. However, in the presence of collision, MDCC requires two message rounds for commitment. Furthermore, MDCC restricts the ability of a client to abort a transaction once the end transaction request has been sent.

More recently, there have been some attempts to consolidate the commitment and consensus paradigms. One such work is TAPIR by Zhang et al. [32]. TAPIR identifies the expensive redundancy caused due to consistency guarantees provided by both the commitment and the replication layer. TAPIR can be specified by the abstractions defined in the C&C framework. In a failure-free and contention-free case, TAPIR uses a *fast-path* to commit transactions, where the coordinator communicates with $\frac{3}{2}f+1$ replicas of each shard to get the value of the transaction. This follows the Value Discovery phase of the C&C framework. G-PAC contrasts with TAPIR mainly in failure recovery during a coordinator crash. TAPIR executes an explicit *cooperative termination protocol* to terminate a transaction after a crash whereas G-PAC has the recovery tightly integrated in its normal execution. There are other subtle differences between G-PAC and TAPIR: TAPIR does not allow aborting a transaction by the coordinator once commitment is triggered. And although *fast paths* provide an optimization over G-PAC, in a contentious workload, TAPIR's *slow paths* make the complexity of both protocols comparable. Finally, G-PAC provides flexibility in choosing any quorum definition across different phases, unlike the *fast-path* quorum $(3/2f+1)$ in TAPIR.

Janus by Mu et al. [25] in another work attempting towards combining commitment and consensus in a distributed setting. In Janus, the commitment of a conflict free transaction needs one round of cross-datacenter message exchange to commit, and with conflicts, it needs two rounds. Although Janus provides low round-trip delays in conflict-free scenarios, the protocol is designed for *stored procedures*. The protocol also requires explicit *a priori* knowledge of write sets in order to construct conflict graphs, which are used for consistently ordering transactions. In comparison, G-PAC is more general, as it does not make any of the assumptions required by Janus.

Another on-going line of work is on deterministic databases, where the distributed execution of transactions are planned a priori to increase the scalability of the system. Calvin [30] is one such example. However, this planning requires declaring the read and write sets before the processing of each transaction. This limits the applicability of deterministic approaches, whereas G-PAC is proposed as a more generalized atomic commit protocol that can be built on top of any transactional concurrency mechanism.

9. CONCLUSION

A plethora of consensus, replication and commitment protocols developed in the past years poses a need to study their similarities and differences and to unify them into a generic framework. In this work, using Paxos and 2PC as the underlying consensus and commit protocols, we construct a Consensus and Commitment (C&C) unification framework. The C&C framework is developed to model many existing data management solutions for the Cloud and also aid in developing new ones. This abstraction pedagogically helps explain and appreciate the subtle similarities and differences between different protocols. We demonstrate the benefits of the C&C framework by instantiating a number of novel or existing protocols and argue that the seemingly simple abstractions presented in the framework capture the essential requirements of many important distributed protocols. The paper also presents an instantiation of a novel distributed atomic commit protocol, Generalized-Paxos Atomic Commit (G-PAC), catering to sharded and replicated data. We claim that separating fault-tolerant replication from the transaction commitment mechanism can be expensive and provide an integrated replication mechanism in G-PAC. We conclude the paper by evaluating the performance of G-PAC with a Spanner-like solution and highlight the performance gains in consolidating consensus with commitment.

10. APPENDIX

Safety in the C&C framework

PROOF. In this section we discuss the safety guarantees that any consensus or commit protocol derived from the C&C framework will provide. **The safety condition states that a value once decided, will never be changed.** Although majority quorums are used to explain the state transitions of the C&C framework, for the safety proof we do not assume any specific form of quorum.

Let $\mathcal{Q}_L$ be the set of all possible leader election quorums used in the Leader Election phase and $\mathcal{Q}_R$ be the set of all possible replication quorums used in the Fault-Tolerant Agreement phase of the C&C framework. Any protocol instantiated from the C&C framework should satisfy the following *intersection condition*:

$$\forall Q_L \in \mathcal{Q}_L, \forall Q_R \in \mathcal{Q}_R : Q_L \cap Q_R \neq \emptyset \tag{4}$$

The safety condition states that: If a value v is decided for ballot number b, and if a value v' is decided for another ballot number b', then $v{=}v'$.

Let $\mathcal{L}$ be the leader elected with ballot number b and v be the value chosen by method $\mathcal{V}$ based on the c&c-value responses. For the chosen value v to be decided, v must be fault-tolerantly replicated on a quorum $Q_R \in \mathcal{Q}_R$.

Now consider another node $\mathcal{L}'$ decides to become leader. $\mathcal{L}'$ sends out c&c-elect-me message with ballot b' to all the other nodes. $\mathcal{L}'$ becomes a leader if it receives c&c-elect-you messages from a quorum $Q_L \in \mathcal{Q}_L$. Based on condition 4, $Q_L \cap Q_R$ is non-empty *i.e.*, there is at least one node $\mathcal{A}$ such that $\mathcal{A} \in Q_L$ and $\mathcal{A} \in Q_R$. There can be two possibilities for ballot b'.

- $b' < b$**:** In this case, $\mathcal{L}'$ will not be able to get c&c-elect-you replies from a quorum Q_L as there is at least one node $\mathcal{A}$ that a ballot $b > b'$ and hence will reject $\mathcal{L}'$'s message.
- $b' > b$**:** In this case, as a response to $\mathcal{L}'$'s c&c-elect-me message, $\mathcal{A}$ sends the previously accepted value v to the new leader. $\mathcal{L}'$ then updates the value to propose from v' to v.

Hence, we show that the C&C framework is safe and any protocol instantiated from the framework will be safe as long as condition 4 is satisfied. □

11. ACKNOWLEDGEMENTS

This work is funded by NSF grants CNS-1703560, CNS-1815733 and CNS-1815212.

12. REFERENCES

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Pub. Co. Inc., Reading, MA, 1987.

[2] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[3] B. Charron-Bost. Comparing the atomic commitment and consensus problems. In *Future directions in distributed computing*, pages 29–34. Springer, 2003.

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[5] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[6] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[7] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 15–28. ACM, 2011.

[8] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.

[9] J. N. Gray. Notes on data base operating systems. In *Operating Systems*, pages 393–481. Springer, 1978.

[10] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *International Workshop on Distributed Algorithms*, pages 87–100. Springer, 1995.

[11] V. Hadzilacos. On the relationship between the atomic commitment and consensus problems. In *In Fault-Tolerant Distributed Computing*, pages 201–208. Springer-Verlag, 1990.

[12] H. Howard, D. Malkhi, and A. Spiegelman. Flexible paxos: Quorum intersection revisited. *arXiv preprint arXiv:1608.06696*, 2016.

[13] I. Keidar and D. Dolev. Increasing the resilience of atomic commit, at no additional cost. In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 245–254. ACM, 1995.

[14] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM, 2013.

[15] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[16] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[17] L. Lamport, D. Malkhi, and L. Zhou. Stoppable paxos. *TechReport, Microsoft Research*, 2008.

[18] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 312–313. ACM, 2009.

[19] B. Lampson and D. B. Lomet. A new presumed commit optimization for two phase commit. In *VLDB*, volume 93, pages 630–640, 1993.

[20] B. Lampson and H. Sturgis. Crash recovery in a distributed system. Technical report, Xerox PARC Research Report, 1976.

[21] B. Lindsay, P. Selinger, C. Galtieri, J. Gray, R. Lorie, T. Price, F. Putzolu, and B. W. Wade. *Notes on distributed databases*. Thomas J. Watson IBM Research Center, 1979.

[22] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *PVLDB*, 6(9):661–672, 2013.

[23] C. Mohan, R. Strong, and S. Finkelstein. Method for distributed transaction commit and recovery using byzantine agreement within clusters of processors. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 89–103. ACM, 1983.

[24] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372. ACM, 2013.

[25] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating concurrency control and consensus for commits under conflicts. In *OSDI*, pages 517–532, 2016.

[26] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[27] D. Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 133–142. ACM, 1981.

[28] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, (3):219–228, 1983.

[29] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)*, 4(2):180–209, 1979.

[30] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[31] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[32] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 263–278. ACM, 2015.