CEP-15: Fast General Purpose Transactions

Elliott Smith, Benedict
benedict@apple.com

Zhang, Tony
nudzhang@umich.edu

Eggleston, Blake
beggleston@apple.com

Andreas, Scott
cscotta@apple.com

Abstract

Modern applications replicate and shard their state to achieve fault tolerance and scalable performance. This presents a coordination problem that modern databases address using leader-based techniques that entail trade-offs: either a scalability bottleneck or weaker isolation. Recent advances in leaderless protocols that claim to address this coordination problem have not yet translated into production systems. This paper outlines distinct performance compromises entailed by existing leaderless protocols in comparison to leader-based approaches. We propose techniques to address these short-comings and describe a new distributed transaction protocol ACCORD, integrating these techniques. ACCORD is the first protocol to achieve the same steady-state performance as leader-based protocols under important conditions such as contention and failure, while delivering the benefits of leaderless approaches to scaling, transaction isolation and geo-distributed client latency. We propose that this combination of features makes ACCORD uniquely suitable for implementing general purpose transactions in Apache Cassandra.

1 Introduction

Modern applications rely upon remote database services to ensure their state is durable and available to clients. To provide these properties, modern databases partition their state into geo-replicated shards. This permits some tolerated combination of failures to coincide without interrupting the service, while ensuring the database may scale to meet user demand. However, a distributed coordination problem is introduced for transaction execution.

Real-world database systems address this by imposing restrictions on functionality or sacrificing performance. Systems that offer transactions using Raft [34] or Multi-Paxos [21] are now common-place [4,13,14,16,29,36,42,44,47], but most do not offer cross-shard transactions. These were first introduced by Spanner [8], but required specialised hardware and multiple WAN round-trips. More recently, systems using commodity hardware have begun to catch up: FaunaDB and FoundationDB offer strict-serializable isolation, but order transactions with a global leader process [14,47]; CockroachDB, YugaByte and DynamoDB avoid this bottleneck, but claim only serializable isolation [6,40,44]. Neither group therefore achieves the optimal combination of isolation properties and scalability. Furthermore, being leader-based these systems require additional wide area round-trips for clients that are not co-located with the leader, and for transactions that involve keys whose leaders are not co-located.

Raft and Multi-Paxos confer some important properties though: they may assign their leader role to any healthy process and require only a simple majority of votes, so they may suffer the loss of any minority of replicas and be able to promptly restore their prior steady-state performance. Transactions that share leaders also do not suffer contention penalties, and reads may be performed concurrently - they may even circumvent the leader entirely [23,31]. Leaderless quorum-based protocols have been proposed [2,11,12,23,30,32,45] that utilise a fast-path to achieve optimal commit latency under low contention, but these have not been used in real systems. We propose that this is in part explained by their unpredictable performance under these same conditions.

In particular, these protocols have fast-path quorums that are disabled by fewer failures than are tolerated overall. For example, Tempo [11] tolerates f failures using $2f+1$ replicas, but at most one replica may fail before its fast-path is unable to reach decisions. Tapir [45] fares better, with a fast path that survives $\lfloor \frac{f}{2} \rfloor$ failures - but this is half as many as it tolerates overall, and its optimistic concurrency control fails to guarantee forward progress for all transactions.

Protocol	Contention Round-trips	Concurrent Reads	No Livelock	No Aborts	General Purpose
TAPIR	2	✓			✓
Janus	2	✓		✓	
Tempo	2		✓	✓	
ACCORD	1	✓	✓	✓	✓

Table 1: Leaderless protocol properties

Switching to the slow-path imposes a burden on the wider system: each operation now consumes more system resources, as the remaining replicas must process and send more messages. Such additional burden introduces the risk of cascading failure or service degradation. This translates particularly poorly to common failure models. Services that are expected to continue in the face of major incidents such as natural disaster, or the widespread loss of power or connectivity, typically utilise multiple regions for redundancy. Existing leaderless protocols require more regions to predictably accommodate the same number of failures. Since each region incurs an independent risk of failure, the risk of experiencing any failure increases as more regions are added. By requiring more regions to accommodate the same number of failures, these protocols afford less protection overall - as well as any additional fixed costs associated with each region.

In addition, existing leaderless protocols offer a choice of undesirable trade-offs: either commutative commands must be serialized [11] or forward progress is not guaranteed [2, 32, 45] (Table 1). These are both properties that we would rather avoid. Guaranteed forward progress in practice means progress is more predictable, while resource-intensive read-only transactions are common — serializing them introduces a bottleneck that cannot be readily circumvented through scaling.

This paper describes techniques to address these shortcomings of leaderless protocols.

We introduce the idea of flexible [18] fast-path quorums for reaching fast-path consensus, offering protocols the ability to modify their fast-path criteria. We demonstrate that by modifying these quorums in response to failure, we can retain fast-path availability under any number of tolerated failures. This supports any replica layout without steady-state performance penalty versus leader-based approaches.

We additionally show how leaderless protocols that utilise timestamps to order transactions [2, 11, 12] may be combined with topographical knowledge and per-replica message buffers [43] to guarantee single round-trip consensus under tolerated latency and clock skew, regardless of message arrival order.

Finally, we describe ACCORD, a leaderless protocol designed to exploit these techniques while further guaranteeing forward progress and permitting commutative commands to be processed concurrently. We propose that this combination of properties makes ACCORD uniquely suited to providing general purpose distributed transactions.

2 Predictable Leaderless Protocols

There exists a broad literature of leaderless state-machine replication (SMR) protocols [2, 11, 12, 30, 32, 45] that utilise quorums, with a fast-path that can achieve optimal commit latency. These protocols utilise a process that executes a transaction on behalf of a client that we shall refer to as the transaction's *coordinator*. This process is responsible for seeking votes and advancing the distributed state machine.

We assume a partially synchronous network, that processes fail only by crashing, and that there are no more than f failed processes in any replica set containing r processes where $f = \lfloor \frac{r-1}{2} \rfloor$. We additionally assume a weak failure detector [5], and that each process has a logical clock that is loosely synchronized with real time.

We introduce the concept of *robustness* and *stability* to illustrate some of the compromises entailed by these protocols.

Robustness. Fast-path quorums are said to be collectively *robust to f failures* if, in the presence of f failed processes, there remains at least one such quorum of correct processes.

Stability. A protocol is *stable to f failures* if no additional per-transaction messages must be exchanged in the presence of f failed processes. A protocol is similarly *stable to contention* if no additional messages must be exchanged when conflicting transactions reach processes out of order.

Protocol	Fast-Path	Failure Tolerance	Failure Robustness	Failure Stability
TAPIR	$\lceil \frac{3f}{2} \rceil + 1$	f	$\lfloor \frac{f}{2} \rfloor$	$\lfloor \frac{f}{2} \rfloor$
Janus	$2f+1$	f	0	0
Tempo_f	$2f$	f	1	1
Tempo_l	$f+1$	1	1	1
ACCORD	$f+1$ to $\lceil \frac{3f}{2} \rceil + 1$	f	0 to $\lfloor \frac{f}{2} \rfloor$	f

Table 2: Comparison of failure properties with $2f+1$ replicas per shard

2.1 Flexible Fast-Path Quorums

Existing protocols [2, 11, 32, 45] have fast-path quorums that are *robust* to fewer failures than are tolerated overall (Table 2). Thus, the protocols are *unstable* in the face of failure: if fast-path quorums become unreachable, remaining processes must exchange additional messages to reach slow-path consensus. Recent work [37] has demonstrated that this trade-off is inherent to leaderless SMR, and that a protocol tolerating f failures cannot have a fast path that is robust to more than $\lfloor \frac{f}{2} \rfloor$ failures. Fortunately, the principles of Flexible Paxos [18] provide an alternative.

We declare that fast-path quorums may be reconfigured, but for now leave the mechanism aside. In particular we may reduce the number of quorums, thereby reducing their collective *robustness*. We will demonstrate that in doing so we are able to reduce the size of the remaining quorums, thereby improving *stability*. The trick is that this is a one-sided trade: quorums that contain one or more crashed processes no longer contribute to robustness, so we lose nothing by removing them.

We assume that a protocol utilises three kinds of quorum, or sets of processes that may unanimously reach decisions: fast-path quorums $\mathcal{F}$ for one round-trip consensus, simple quorums $\mathcal{Q}$ for slow-path consensus, and recovery quorums $\mathcal{R}$ for completing interrupted operations. We make no assumptions about the kind or configuration of these quorums but define the set of fast-path quorums to be $\mathbb{F}$. For recovery to arbitrate between transactions that may each have potentially taken the fast path, it is common to require that any two fast-path quorums must intersect with each other and any recovery quorum at one or more correct replicas [2, 17, 30, 32], i.e., $\mathcal{F} \cap \mathcal{F}' \cap \mathcal{R} \neq \emptyset$. For protocols that follow this design it is possible to guarantee *stability* to all tolerated failures, as we will now demonstrate.

Under maximal tolerated failures there must always be a simple quorum $\mathcal{Q}_0$ consisting of correct processes that may reach decisions, and thus we may always assign our fast-path the set containing only $\mathcal{Q}_0$. In this case all fast paths use the same quorum, so their intersection is also $\mathcal{Q}_0$. For recovery to be possible on the slow-path, all simple and recovery quorums must intersect at one or more correct replicas, therefore this set of fast-path quorums maintains our above properties. That is to say, if we fix $\mathbb{F} = \{\mathcal{Q}_0\}$ so that $\mathcal{F} = \mathcal{F}'$ we only require that $\mathcal{F} \cap \mathcal{R} \neq \emptyset$. Since $\mathcal{Q} \cap \mathcal{R} \neq \emptyset$ and $\mathcal{F} = \mathcal{Q}_0$ it must therefore be that $\mathcal{F} \cap \mathcal{R} \neq \emptyset$, and we may reach fast-path consensus under maximal tolerated failures.

This ensures no additional per-transaction messages must be exchanged under any number of tolerated failures, providing optimal *stability to failure*.

2.1.1 Fast-Path Electorates

Most quorum based protocols [2, 11, 21, 30, 34] use simple vote thresholds, where any replica may vote in combination with any other. This is the simplest model to reason about, so we adopt it for ACCORD. We introduce the concept of an *electorate*, namely those replicas whose votes may be counted towards such a vote threshold. Ordinarily all replicas are members of both the simple and fast-path electorates, but in the case of the fast-path we may remove replicas in order to reduce the successful vote threshold. We define a fast-path electorate $\mathbb{E}$, the membership of which makes fast-path decisions. To maintain the previously outlined recovery properties, namely that the intersection of any recovery quorum with any two fast-path quorums contains at least one correct replica, we derive the vote threshold as follows:

$$|\mathcal{F}| - (|\mathbb{E}| - |\mathcal{F}|) - f > 0 \tag{1}$$

$$\implies \quad |\mathcal{F}| = \left\lceil \frac{|\mathbb{E}| + f + 1}{2} \right\rceil \tag{2}$$

For example, a replica set of $r = 9$ processes tolerating $f = 4$ failures could choose an electorate of size 9, 7 or 5 conferring fast-path quorums of 7, 6 and 5 respectively. An electorate of 9 could then reach fast-path decisions with fewer than 3 failed processes, at which point the electorate would need to be reduced to either 7 or 5. Under the maximum 4 tolerated failures, all 4 failed processes would need to be excluded from the electorate so that all fast- and slow-path decisions would use the same quorum.

2.2 Timestamp Reorder Buffer

Leaderless consensus protocols can directly process transactions from clients in any region, but this capability introduces the risk of contention: two transactions started at the same time but in different regions may be unable to both reach fast-path consensus. Those replicas closest to each coordinator will witness its corresponding transaction first, so that no fast-path quorum may witness a consistent order of arrival for either transaction. Recent work [43] has demonstrated the potential for loosely synchronized clocks to improve this scenario. Coordinators may assign transactions a future time to be processed on recipient replicas, with the time selected to allow conflicting transactions sent by further-away coordinators to arrive first.

This approach can be refined to guarantee consensus for every transaction under realistic conditions. Firstly, since clocks are not perfectly synchronized messages may still be processed too early. By measuring clock asynchrony, or clock skew we can extend our processing delay to reliably accommodate differences in link latency. Secondly, by utilising a protocol such as Caesar [2] or Tempo [11], that proposes an execution timestamp. Unlike protocols that agree dependencies [30], where every fast-path vote must witness identical histories, timestamp protocols only require that voters witness timestamps in ascending order. Therefore, if the message processing time is derived from the transaction's proposed execution timestamp we are able to guarantee fast-path consensus for all transactions proposed by a correct and responsive coordinator, using only two weak assumptions: that we can measure both the latency of our network and the error margins of our clocks.

We assume a protocol P that utilises a fast-path for one-round consensus, that its correctness is independent of message arrival order, and that it proposes a timestamp t_0 as a transaction's intended execution time. We will refer to this round as $PreAccept$. If a fast path quorum witnesses t_0 before any higher timestamp then it must be accepted, and this must uniquely determine the transaction's execution order.

We define the set $\mathbb{C}$ of all coordinators, $Skew_{Max}$ the maximum instantaneous difference between the logical clocks on any two nodes, and $Latency(C,P)$ as the latency between some coordinator C and some replica P. We finally define $Deadline(t_0,C,P) = t_0 + Skew_{Max} + \mathbf{max}(Latency(C',P) \mid C' \in \mathbb{C}) - Latency(C,P)$. On replica P, receiving a message from coordinator C, this computes the last point in time at which a transaction might arrive from other members of $\mathbb{C}$ with an earlier t_0.

There are no modifications necessary to protocol P. On receiving a *PreAccept* request, replicas do not immediately process the message, instead buffering it in a queue ordered by t_0 until the computed $Deadline$, at which point the buffer is processed up to and including this message.

Since P's safety does not depend on message arrival order, it is unaffected by this change. We assume that our bounds on message latency and clock synchrony are not breached and that processes are responsive. Under these conditions, this generic change guarantees that for any two transactions γ and τ with $t_{0\gamma} < t_{0\tau}$ so that τ could prevent fast-path consensus for γ, it must be that γ is known to $\mathbb{E}$ before τ is processed, and will be processed first. As a result, out-of-order transaction arrival does not result in any additional messages being exchanged.

This provides *stability to contention*.

2.2.1 Clock Synchrony versus Latency

This technique trades wide area round-trips for an additional latency penalty equal to the bounds on clock synchrony. How useful is this exchange in practice? Many modern data centres have at least one high quality time source such as GPS that is typically combined with a protocol such as NTP or PTP to synchronize commodity time sources residing on the local network, such as those embedded into modern CPUs. In this configuration clock skew may be sub-millisecond [15], so that this penalty will likely be dwarfed by wide-area latency. Importantly any additional latency is uncorrelated with wide-area latency: each locale has its own time source and dissemination mechanism, so that accommodating clock skew represents a fixed global penalty.

3 ACCORD Basic Protocol

ACCORD builds upon a rich history of leaderless state-machine replication (SMR) protocols [2, 11, 12, 32] exploiting the intuitions of EPaxos [30]. This broad approach can be viewed as performing a separate instance of classic Paxos for each transaction to agree its execution order, with a special initial round that permits faster consensus. Recent work has shown that this technique can be easily extended to partial state-machine replication (PSMR) scenarios for improved scalability, with shards replicating only a portion of the global state-machine [11, 32]. ACCORD utilises this approach to agree a timestamp to determine a transaction's execution order, reducing the incidence of contention. Like Caesar [2], ACCORD uses its consensus acknowledgments to determine the set of conflicting transactions that may execute earlier, however the worst case behaviour is improved so that consensus is reached in at most two round-trips, and live-lock [11] is avoided. Uniquely, this dependency set is not decided consistently during consensus - only a superset of each transaction's final dependencies is determined, that may be filtered during execution to ensure consistency. This approach achieves an optimal baseline combination of characteristics compared to comparable state-of-the-art protocols, providing a suitable platform for utilising a fast-path electorate and reorder buffer.

As is typical for Paxos [21] derivatives, ballots are used to ensure only the most recent command for a given phase may complete [2, 11, 23, 30]. For brevity we elide the handling of these ballots from the Consensus protocol, and implicitly discard messages if a newer ballot has been witnessed. We additionally discard messages from earlier phases of the protocol.

Transactions are denoted by γ, τ and υ. We say that two transactions γ and τ **conflict** ($\gamma \sim \tau$) if their execution is not commutative, so that either their response or the database state would differ if their execution order were reversed.

Variables declared with a τ subscript are persistent, storing the most recent value for τ on that process. $\rho, \mathbb{E}, \mathbb{Q}, \mathcal{F}$ represent, respectively, a replica-set for a single shard, its fast-path electorate, and a simple and fast-path quorum. By the same token, $\mathbb{P}^\tau, \mathbb{E}^\tau, \mathbb{Q}^\tau, \mathcal{F}^\tau$ denote the replica-sets participating in τ, the union of their fast-path electorates, and a set of responses constituting a simple and fast-path quorum in each replica-set in $\mathbb{P}^\tau$.

3.1 Overview

A client process that wishes to execute a transaction selects a nearby non-replica coordinator *C* to perform consensus with Algorithm 1 so that an execution timestamp may be decided. Along with this timestamp, the protocol also determines a set of conflicting transactions that might precede our execution. Algorithm 2 is then performed, also by *C*, so that general-purpose transactions combining cross-shard state may be executed. *C* steps through several phases across these two protocols: *PreAccept*, *Accept*, *Commit*, *Execute* and *Apply*, all of which must occur in sequence. Only *Accept* may be skipped, if the fast path is taken. *PreAccept* and *Accept* collectively decide the order of execution, which is durably decided prior to entry to *Commit*, where this decision is disseminated to dependent transactions. *Execute* then awaits the execution of those dependencies with lower execution timestamps before computing the result of the transaction. *Apply* persists this result to all replicas. Should the coordinator fail, a weak failure detector invokes the recovery protocol on a replica that has witnessed the transaction, that then takes over the coordinator role from the last successfully completed phase.

In the following sections the consensus and execution protocols will be described in detail, alongside certain simple properties that they maintain. In Section 4 the recovery protocol will be described for finishing interrupted transactions. In Section 3.4 and Section 4 these properties will be used to demonstrate that these protocols in combination maintain the following properties:

Consistency. Conflicting transactions are applied in the same order on all participating replicas.

Real-time order. Any transaction γ acknowledged before a conflicting transaction τ has been submitted is executed and applied before τ on all replicas in common.

Liveness. Any transaction that is known by at least one correct replica will execute eventually and apply on all correct participating replicas.

3.2 Consensus

ACCORD imposes a total order on conflicting transactions by assigning them each a unique execution timestamp. Timestamps consist of a tuple $(time, seq, id)$, where *time* is assigned from a per-process monotonically increasing value that is loosely synchronised with the wall clock, *seq* is a logical time component and *id* is a unique identifier of the process that created the timestamp. A timestamp sorts by its components in precedence of their declared order. We

use three symbols to represent timestamps $t_0 \leq t \leq T$ where t_0 denotes the timestamp first proposed by *PreAccept*, t a proposed execution timestamp and T_τ the highest timestamp witnessed by a process for τ.

At a high level, a Lamport clock of these timestamps assigns execution times to conflicting transactions, imposing a total order [20]. ACCORD ensures this order is consistent for transactions that have been acknowledged to clients, so that in the event of failure the same total order will be recovered. Initially, a transaction coordinator proposes a timestamp for execution. If a fast-path quorum of replicas unanimously accept and record this timestamp as the most recent, then only this single timestamp may be recovered and it is decided immediately. Otherwise some replicas have responded proposing newer timestamps, so that there are multiple possible Lamport timestamps that may be recovered, and therefore distinct total orders. In this case a slow path using classic Paxos durably agrees which of these possibilities is decided. To execute transactions in timestamp order, replicas also inform coordinators of any conflicting transactions that may take an earlier timestamp. Execution proceeds once these have all committed, and any with an earlier timestamp have executed. This is now described in more detail, alongside certain properties that are maintained.

Algorithm 1 Consensus Protocol

receive τ **on coordinator** C **from client:**

1: $t_0 \leftarrow (\text{now}, 0, C)$
2: **send** $PreAccept(\tau, t_0)$ to $\forall p \in \mathbb{E}^\tau$

receive $PreAccept(\tau, t_{0\tau})$ **on** p**:**

3: **if** $t_{0\tau} > \max(T_\gamma \mid \gamma \sim \tau)$ **then**
4: $\quad t_\tau \leftarrow t_{0\tau}$
5: **else**
6: $\quad t_\tau \leftarrow \max(T_\gamma \mid \gamma \sim \tau)$
7: $\quad t_\tau.(seq, id) \leftarrow (t_\tau.seq + 1, p)$
8: **end if**
9: $T_\tau \leftarrow t_\tau$
10: $PreAccepted_\tau \leftarrow$ **true**
11: **reply** $PreAcceptOK(t_\tau, deps : \{\gamma \mid \gamma \sim \tau \wedge t_{0\gamma} < t_{0\tau}\})$

receive $PreAcceptOK(t, deps)$ **from** $\mathbb{Q}^\tau$**:**

12: $deps \leftarrow \bigcup\{p.deps \mid p \in \mathbb{Q}^\tau\}$
13: **if** $\exists \mathcal{F}^\tau \subseteq \mathbb{Q}^\tau \ (\forall p \in \mathcal{F}^\tau \cdot p.t = t_0)$ **then**
14: $\quad$ **send** $Commit(\tau, t_0, t_0, deps)$ to $\forall p \in \rho \in \mathbb{P}^\tau$
15: $\quad$ **go to** Execution Protocol
16: **else**
17: $\quad t \leftarrow \max(p.t \mid p \in \mathbb{Q}^\tau)$
18: $\quad$ **send** $Accept(\tau, t_0, t, deps)$ to $\forall p \in \rho \in \mathbb{P}^\tau$
19: **end if**

receive $Accept(\tau, t_{0\tau}, t_\tau, deps_\tau)$**:**

20: $T_\tau \leftarrow \max(t_\tau, T_\tau)$
21: $Accepted_\tau \leftarrow$ **true**
22: **reply** $AcceptOK(deps : \{\gamma \mid \gamma \sim \tau \wedge t_{0\gamma} < t_\tau\})$

receive $AcceptOK(deps)$ **on** C **from** $\mathbb{Q}^\tau$**:**

23: $deps \leftarrow \bigcup\{p.deps \mid p \in \mathbb{Q}^\tau\}$
24: **send** $Commit(\tau, t_0, t, deps)$ to $\forall p \in \rho \in \mathbb{P}^\tau$
25: **go to** Execution Protocol

Property 3.1 (Timestamp ordering)**.** *For any two conflicting transactions $\gamma \sim \tau$ where γ commits before τ is submitted by a client, $t_\gamma < t_\tau$.*

The coordinator assigns transaction τ a globally unique timestamp t_0 and proposes this to $\mathbb{E}^\tau$, the members of all fast-path electorates participating in τ. This is the lowest possible execution timestamp t that τ may take. On recipient replicas, t_0 is compared with the largest timestamp witnessed for all conflicting transactions $\gamma \sim \tau$. If it is larger, the replica votes in favour of setting $t = t_0$; otherwise a new larger t is proposed. In either case the proposed t is stored by the replica so that it may only vote for higher timestamps for future conflicting transactions.

Each replica additionally includes in its *PreAcceptOK* response the set $deps_\tau$ of conflicting transactions $\gamma \sim \tau$ witnessed by the replica that might take a lower execution timestamp, namely those where $t_{0\gamma} < t_{0\tau}$. Note that any transitive dependency of another $\gamma \in deps_\tau$ where $Committed_\gamma$ may be pruned from $deps_\tau$, as it is durably a transitive dependency of τ.

Property 3.2 (Timestamp consistency)**.** *All processes that commit a transaction do so with the same timestamp.*

If a fast-path quorum $\mathcal{F}^\tau$ votes to accept $t = t_0$, this timestamp is durably decided and is committed to all replicas participating in τ. Fast-path quorums and their durability are described in more detail in Section 4.

If insufficient fast-path responses are received, the maximum t from any simple quorum $\mathcal{Q}^\tau$ may be taken to impose a valid total order [20]. However there is no single t that can be durably recovered should this coordinator fail. To ensure consistency we require that a recovery coordinator will select the same t that we proceed to *Execute* with. We achieve this with an *Accept* phase that can be viewed as a classic Paxos *Accept* phase with a privileged initial ballot b_0, similar to Fast Paxos [23]. This phase requires only a simple quorum for all $\rho \in \mathbb{P}^\tau$ before τ may be committed.

In this case every replica includes in its *AcceptOK* response the set of conflicting transactions $\gamma \sim \tau$ witnessed by the replica that might take a lower execution timestamp, namely those where $t_{0\gamma} < t_\tau$.

3.3 Execution

Once t_τ is decided it is logically committed, and disseminated to every shard alongside $deps_\tau$ via *Commit* so that other transactions with an earlier execution timestamp that had witnessed us during consensus may proceed. Note that only t_τ is durably committed, $deps_\tau$ is only recorded for recovery purposes outlined in Section 4. Simultaneously the coordinator issues *Read* messages to at least one process in each $\rho \in \mathbb{P}^\tau$, including the nearest correct process for each shard. In common configurations this will permit a response without any wide-area latency. The *Read* request includes the subset of $deps_\tau$ that interacts with ρ, and replicas wait to answer this message until every such dependency has either been witnessed as committed with a higher execution timestamp $t_\gamma > t_\tau$, or its result has been applied locally. Replicas do not advance further until the coordinator evaluates τ against these responses and submits *Apply* messages, that must in turn be received by replicas before they may process conflicting transactions with a higher timestamp.

Algorithm 2 Execution Protocol

Coordinator C**:**

for $\rho \in \mathbb{P}^\tau$ **do**
  $deps_\rho \leftarrow \{\gamma \mid \gamma \in deps \wedge \rho \in \mathbb{P}^\gamma\}$
  send $Read(\tau, t, deps_\rho)$ to some nearby $p \in \rho$
end for

receive $Commit(\tau, t_{0\tau}, t_\tau, deps_\tau)$**:**

$Committed_\tau \leftarrow$ **true**

receive $Read(\tau, t_\tau, deps_{\tau,\rho})$ **on** p**:**

await $Committed_\gamma\ \forall \gamma \in deps_{\tau,\rho}$
await $Applied_\gamma\ \forall \gamma \in deps_{\tau,\rho},\ t_\gamma < t_\tau$
$reads \leftarrow \text{read}(\tau)$
reply $ReadOK(reads)$

receive $ReadOK(reads)$ **from each shard:**

$result \leftarrow \text{execute}(\tau, reads)$
send $Apply(\tau, t, deps_{\tau,\rho}, result)$ to $\forall p \in \rho \in \mathbb{P}^\tau$
send *result* to client

receive $Apply(\tau, t_\tau, deps_{\tau,\rho}, result_\tau)$**:**

await $Committed_\gamma\ \forall \gamma \in deps_{\tau,\rho}$
await $Applied_\gamma\ \forall \gamma \in deps_{\tau,\rho},\ t_\gamma < t_\tau$
$\text{apply}(writes, t_\tau)$
$Applied_\tau \leftarrow$ **true**

Note. The result of execution is persisted at replicas to avoid the following situation: if a transaction involving shards a and b is applied at all correct ρ_a, but is unapplied at all correct ρ_b, we would be unable to re-compute the result

of τ, and transactions that conflict on b would be unable to proceed. By persisting its result, these transactions may consult a to complete τ on b.

3.4 Safety

We now demonstrate that this protocol maintains the atomicity and strict-serializable isolation properties, namely that every transaction is executed by every replica, in the same order, and that to an external observer every transaction occurs in an order consistent with the real-time order of client-visible events.

Property 3.3 (Dependency safety)**.** *Any coordinator committing τ with t_τ does so with $deps_\tau$ containing all conflicting γ that may be committed with $t_\gamma < t_\tau$.*
Proof. Suppose τ is committed with t_τ and $deps_\tau$, and consider any γ that commits with $t_\gamma < t_\tau$. Each of γ, τ may have committed via the slow- or fast-paths. We illustrate the case where both γ, τ committed via the slow-path; the other cases are similar. By assumption, γ is pre-accepted at some slow-path quorum $\mathbb{Q}^\gamma$, where each replica in $\mathbb{Q}^\gamma$ pre-accepted γ for some execution timestamp $\leq t_\gamma$. Meanwhile, τ must be accepted at some slow-path quorum $\mathbb{Q}^\tau$ with t_τ. Then for any replica P in $\mathbb{Q}^\gamma \cap \mathbb{Q}^\tau$, P must have pre-accepted γ before accepting τ, since otherwise P will not have pre-accepted γ with an execution timestamp less than t_τ. As a result, by the Accept Phase of the protocol, $\gamma \in deps_\tau$. □

Property 3.4 (Timestamp order)**.** *For any transactions γ and τ where $\gamma \sim \tau$ and $t_\gamma < t_\tau$, τ executes after γ.*
Proof. By Property 3.3 the coordinator knows all γ that may be committed with $t_\gamma < t_\tau$. By waiting for these to commit we know all γ where $t_\gamma < t_\tau$ and wait for them to apply on a replica of each shard. □

Property 3.5 (Application order)**.** *For any transaction τ, τ applies at p after all $\gamma \sim \tau$ where, once γ is committed, $t_\gamma < t_\tau$.*
Proof. By Property 3.3 the coordinator knows all γ that may be committed with $t_\gamma < t_\tau$, and waits for them to apply before sending the result of executing τ to p. The subset of $deps_\tau$ that applies on p is included, and p waits for this subset to apply locally before applying τ. □

Theorem 3.1 (Consistency)**.** *For any two conflicting transactions $\gamma \sim \tau$ where γ is applied before τ on some replica, γ will be applied before τ on all replicas in common.*
Proof. By Properties 3.2, 3.3 and 3.4, $t_\gamma < t_\tau$ and $\gamma \in deps_\tau$. By Properties 3.4 and 3.5 γ executes before τ and applies before τ at all replicas. □

Theorem 3.2 (Real-time order)**.** *For any two conflicting transactions $\gamma \sim \tau$ where τ is proposed by a client after γ has been committed, γ executes before τ, and is applied before τ at all replicas in common.*
Proof. By Properties 3.1, 3.2 and 3.3, $t_\gamma < t_\tau$ and $\gamma \in deps_\tau$. By Properties 3.4 and 3.5 γ executes before τ and applies before τ at all replicas. □

4 Recovery Protocol

Algorithm 3 is invoked by a weak failure detector to recover a transaction τ whose coordinator has failed. This protocol contacts a recovery quorum $\mathcal{R}^\tau$ that ensures τ is pre-accepted before responding, ensuring the properties of normal execution are maintained against any υ that had not previously committed. For those υ that had already committed we must now demonstrate that we maintain our declared properties while recovering τ. In the case where τ has reached a slow-path decision, any quorum $\mathcal{R} \subseteq \rho$ where $|\mathcal{R}| \geq r - f$ must witness at least one response from the last completed phase of the state machine. The state machine can be picked-up from this point, similarly maintaining the properties of normal execution. We now only require a similar capability for fast-path decisions.

Fast-path recovery operates on a simple intuition: if we may deduce that an incomplete transaction τ either cannot have been committed on the fast path *or* that any υ that may execute after τ has $\tau \in deps_\upsilon$, then we may safely propose $t = t_0$ by the slow path. If υ is committed after the recovery quorum was reached then it must witness τ. Otherwise this is achieved by using the *deps* associated with all *Accepted* or *Committed* υ that supersede τ: if any of these had

Algorithm 3 Recovery Protocol

Coordinator C**:**

1: $b \leftarrow$ fresh ballot
2: **send** $Recover(b, \tau, t_0)$ to $\forall p \in \rho \in \mathbb{P}^\tau$

receive $Recover(b, \tau, t_0)$ **on** p**:**

3: **if** $b \leq MaxBallot_\tau$ **then**
4: **reply** $NACK(b_\tau)$
5: **else**
6: $MaxBallot_\tau \leftarrow b$
7: $Accepts \leftarrow \{\gamma \mid \gamma \sim \tau \wedge \tau \notin deps_\gamma \wedge Accepted_\gamma\}$
8: $Commits \leftarrow \{\gamma \mid \gamma \sim \tau \wedge \tau \notin deps_\gamma \wedge Committed_\gamma\}$
9: $Wait \leftarrow \{\gamma \in Accepts \mid t_{0\gamma} < t_{0\tau} \wedge t_\gamma > t_{0\tau}\}$
10: $Superseding \leftarrow \{\gamma \in Accepts \mid t_{0\gamma} > t_{0\tau}\}$
11: $\cup \{\gamma \in Commits \mid t_\gamma > t_{0\tau}\}$
12: **if** $\nexists PreAccepted_\tau$ **then**
13: **run** Consensus Protocol L3 to L10 ▷ *PreAccept*
14: **end if**
15: **if** $\neg Accepted_\tau \wedge \neg Committed_\tau \wedge \neg Applied_\tau$ **then**
16: $deps_\tau \leftarrow \{\gamma \mid \gamma \sim \tau \wedge t_{0\gamma} < t_{0\tau}\}$
17: **end if**
18: **reply** $RecoverOK(*_\tau, Superseding, Wait)$
19: **end if**

receive *NACK* **on** C**:**

20: **yield** to competing coordinator

receive $RecoverOK(*, Superseding, Wait)$ **from** $p \in \mathcal{R}^\tau$

21: **if** $\exists p \in \mathcal{R}^\tau\ (p.Applied_\tau)$ **then**
22: **send** $response(result)$ to client
23: **send** $Apply(\tau, t_0, p.t, p.deps, result)$ to $\forall p' \in \rho \in \mathbb{P}$
24: **else if** $\exists p \in \mathcal{R}^\tau\ (p.Committed_\tau)$ **then**
25: **send** $Commit(\tau, t_0, p.t, p.deps) \forall p' \in \rho \in \mathbb{P}^\tau$
26: **go to** Execution Protocol
27: **else if** $\exists p \in \mathcal{R}^\tau\ (Accepted_\tau)$ **then**
28: **select** p with highest *accepted* ballot
29: $t \leftarrow p.t$; $deps \leftarrow p.deps$
30: **go to** Consensus Protocol L18 ▷ Accept
31: **else**
32: $t \leftarrow t_0$; $deps \leftarrow \bigcup\{p.deps_\tau \mid p \in \mathcal{R}^\tau\}$
33: **if** $\exists i \left|\{p \in \mathbb{E} \mid p.t > p.t_0\}\right| > |\mathbb{E}| - |\mathcal{F}_i|$ **then**
34: $t \leftarrow \max(p.t \mid p \in \mathcal{R}^\tau)$
35: **else if** $\exists p \in \mathcal{R}^\tau\ (p.Superseding \neq \emptyset)$ **then**
36: $t \leftarrow \max(p.t \mid p \in \mathcal{R}^\tau)$
37: **else if** $\bigcup\{p.Wait \mid p \in \mathcal{R}^\tau\} \neq \emptyset$ **then**
38: **await** $Committed_\gamma\ (\forall \gamma \in \bigcup\{p.Wait \mid p \in \mathcal{R}^\tau\})$
39: **restart** Recovery Protocol
40: **end if**
41: **go to** Consensus Protocol L18 ▷ Accept
42: **end if**

not witnessed τ in $deps_\upsilon$ then τ had not reached a fast-path before they did, and since υ supersedes τ it may not reach fast-path consensus after. Conversely, if all have witnessed τ then they will wait for τ to commit so that we may propose any t, including t_0.

In more detail, given transactions $\upsilon \sim \tau$ where υ supersedes τ, any shard ρ that υ and τ have in common, any recovery quorum $\mathcal{R}^\tau$, and any subset E of the shard's electorate that may be witnessed by this quorum[1], we must be able to determine either that τ is not pre-accepted or that $\tau \in deps_\upsilon$. To achieve this, as outlined in Section 2.1.1 we ensure that $\mathcal{F}^\tau \cap \mathcal{F}^\upsilon \cap E \neq \emptyset$, i.e. that $|\mathcal{F}| = \lceil \frac{|\mathbb{E}|+f+1}{2} \rceil$. Now any τ that did not take the fast path must have fewer than $r - |\mathcal{F}|$ votes for $t = t_0$. However not all τ with this many votes took the fast path, so we must either (1) deduce from other information that they did not, so that we may propose a higher timestamp; or (2) determine that it is safe to propose $t = t_0$ regardless. If τ was committed on the fast-path then proposing $t = t_0$ is correct. If it was not then *Timestamp consistency* and *Timestamp ordering* are unaffected by proposing $t = t_0$, so that we must only demonstrate that *Dependency safety* continues to be maintained in this case.

By definition, if *both* τ and υ may have reached fast-path consensus then $\mathcal{F}^\tau \cap \mathcal{F}^\upsilon \neq \emptyset$ so that $\tau \in deps_\upsilon$. By simple induction this maintains *Dependency safety* between all transactions that may have committed on the fast-path.

For any υ committed on the slow path, an *Accept* or *Commit* must be witnessed by $\mathcal{R}^\tau$ so that we may inspect $deps_\upsilon$. If $Accepted_\upsilon$, υ must have reached a simple quorum $\mathcal{Q}^\upsilon$ during *PreAccept* before proposing $deps_\upsilon$, so that if $\tau \notin deps_\upsilon$ and $t_{0\upsilon} > t_{0\tau}$ we know that τ could not have reached fast-path consensus and we may proceed on the slow path. Similarly, if $Committed_\upsilon$, υ must have reached $\mathcal{Q}^\upsilon$ during *Accept* so we may test $t_\upsilon > t_{0\tau} \wedge \tau \notin deps_\upsilon$. Conversely, if $\tau \in deps_\upsilon$ and υ successfully committed, any execution of υ will know of τ so that *Dependency Safety* is maintained.

This leaves only those transactions γ where $t_{0\gamma} < t_{0\tau}$ that may have reached slow-path consensus with $t_\gamma > t_{0\tau}$ but have not been committed. Such transactions may not have witnessed τ without prohibiting τ from reaching fast-path consensus. If we have not determined the correct course of action by other means, we may simply wait for these transactions to commit, or commit them ourselves.

Theorem 4.1 (Liveness)**.** *A transaction τ that is known by at least one healthy replica will always execute eventually and apply on all correct participating replicas*

Proof. A healthy coordinator executing the consensus protocol defined in Algorithm 1 always commits τ in a fixed number of steps unless superseded by a recovery coordinator. If the coordinator process is faulty then, given synchrony conditions and weak failure detectors, a recovery coordinator is eventually picked. The recovery protocol defined in Algorithm 3 commits in a fixed number of steps unless there exists an γ where $Accepted_\gamma \wedge t_{0\gamma} < t_{0\tau} \wedge t_\gamma > t_{0\tau}$. Given loosely synchronized clocks there are finite γ, so by simple induction all commands are able to commit eventually. Once committed, execution waits only for those commands $\gamma \in deps_\tau$ where $t_\gamma < t_\tau$, so all commands are able to eventually execute. This result is sent to *Apply* on all healthy replicas. If a healthy process witnesses τ, but does not witness *Apply*, it eventually invokes the recovery protocol. □

5 Reconfiguring Electorates

A configuration consists of a monotonically increasing *epoch* identifier, the membership of all shards ρ and their fast-path electorates $\mathbb{E}$. We say that a configuration e_{i+1} has taken effect on ρ once a simple quorum of ρ_{e_i}, the membership of ρ as decided by e_i, is aware of it.

Timestamps are modified to include *epoch* as their first value, so that a transaction τ is coordinated by C with configuration e_1 using $t_{0\tau}.epoch = e_1$. Any $p \in \rho_{e_1}$ with a newer configuration e_2 will respond to *PreAccept* with $t_\tau.epoch = e_2$ so that p may not participate in fast-path decision. This alerts C to the new configuration, which is fetched asynchronously. If a fast-path quorum is not reached with ρ_{e_1}, we wait for the new configuration and for $\mathcal{Q}_{e_2}$ *PreAcceptOK* responses - if necessary sending additional *PreAccept* to ρ_{e_2}. The remainder of the Consensus protocol executes for both configurations simultaneously, i.e. so that *Accept* is sent to all $\mathbb{P}^\tau = \mathbb{P}^\tau_{e_1} \cup \mathbb{P}^\tau_{e_2}$, and $\mathcal{Q}^\tau = \mathcal{Q}^\tau_{e_1} \cup \mathcal{Q}^\tau_{e_2}$ must respond with *AcceptOk* before we may proceed to the *Commit* phase. The Execution protocol must wait for $deps_\tau$ to apply in $\mathbb{P}^\tau_{e_1}$ only, before applying in $\mathbb{P}^\tau_{e_2}$ only. The Recovery protocol remains unmodified, depending only on these changes to the Consensus and Execution protocols.

This ensures atomic migration from one configuration to another. No replica that is aware of e_2 may participate in a fast-path decision using e_1, so if sufficient fast-path responses are received e_2 has not taken effect and τ completes

[1] That is, where $t_{0\tau} < t_{0\upsilon}$, $\rho \in (\mathbb{P}^\tau \cap \mathbb{P}^\upsilon)$, $\mathbb{E} \subseteq \rho$ and $E = \mathcal{R}^\tau \cap \mathbb{E}$

normally. A slow-path decision executes the protocol for both configurations, necessarily informing a majority of ρ of the new configuration, so that it takes effect for transactions that follow τ.

However, by itself this modification does not maintain consistency during recovery. Recall that the simultaneous intersection of any two fast-path quorums with any recovery quorum must include at least one correct process. To ensure this property is maintained, replicas that join the electorate must not participate in any fast-path decision until all fast-path decisions from the prior configuration are known to them. This is achieved by requiring that at least $1 + \left|\mathbb{E}_{e_1}\right| - \left|\mathcal{F}_{e_1}\right|$ members of $\mathbb{E}_{e_1}$ inform the new members of $\mathbb{E}_{e_2}$ of all transactions they accepted on the fast-path under configurations $\leq e_1$. Since these replicas will no longer accept transactions under e_1, and no fast-path decision may be taken under e_1 without at least one of these replicas participating, this constitutes knowledge of all fast-path decisions taken under e_1. Therefore, all fast-path decisions involving the new members of $\mathbb{E}_{e_2}$ must witness all fast-path decisions taken against e_1. If another configuration e_3 adds additional members, those members that joined in e_2 must wait until they have received their join notifications from members of $\mathbb{E}_{e_1}$ before propagating this knowledge on to those members joining in e_3. This way all new members in any configuration are aware of every fast-path decision taken by all earlier configurations. Specifically, we introduce a new boolean property $\mathit{ReadyElectorate}_e$, and modify the Consensus Protocol at Line 3 so that a replica $p \in \mathbb{E}_{e_i}$ where $p \notin \mathbb{E}_{e_{i-1}}$ also requires Ready_{e_i} to hold, else the slow path branch is taken.

The complete Reconfiguration Protocol is included in the appendix, alongside fully specified versions of the Consensus, Execution and Recovery Protocols integrating the above details. A detailed proof of correctness and liveness properties for these protocols is also included in the appendix.

6 Related Work

State machine replication (SMR). SMR [38] is a common foundation for building reliable, fault-tolerant systems. Traditional SMR protocols [21–23, 25, 33, 34] rely on a stable leader to efficiently assign a linearizable ordering to client requests. Replicas execute requests in this order, and are able to adopt the leader role should it fail. This provides clients the illusion of a unified service running on a single machine. However, the single leader in these protocols is a bottleneck that limits throughput, and results in increased latency for remote clients in a wide area network.

For better performance in geo-replicated systems, SMR protocols such as Mencius [28] and EPaxos [30] have been proposed. While Mencius permits the leader role to be efficiently rotated between regions to improve client latency, if any replica is faulty progress stalls until this is detected and recovered. EPaxos instead discards the concept of a leader, permitting any replica to coordinate a command at the same time as any other. Dependency graphs and topological sorting ensure a consistent order of execution for non-commutable commands. Recently EPaxos has been demonstrated to suffer degraded performance under contended workloads [2, 12, 43]. Two approaches to this problem have been outlined: loosely synchronized clocks have been employed to synchronize the order in which replicas process conflicting commands [43], and alternative protocols such as Caesar [2] and Atlas [12] have been proposed that agree an execution timestamp, permitting replicas to reach consensus without witnessing identical histories, so that conflicts may be reduced. ACCORD combines both techniques, proposing an execution timestamp and processing messages in timestamp order. This combination provides ACCORD stronger *stability* properties. Additionally, ACCORD has a worst case execution path of two wide area round-trips instead of Caesar's three, ACCORD does not order non-commutative commands in comparison to Atlas.

Separately, SMR approaches have been proposed that vary failure handling properties. Vertical Paxos [24] demonstrated consensus with an optimal two replicas, by permitting the membership to be reconfigured between rounds. Flexible Paxos [18] demonstrated that quorums need not be homogenous, and that these quorums may be reconfigured between rounds, affording similar advantages to Vertical Paxos. Fast Flexible Paxos [17], Atlas [12] and Tempo [11] exploit Flexible Paxos' non-homogenous quorums to trade failure tolerance for steady-state performance. ACCORD demonstrates that optimal failure tolerance may be maintained alongside optimal *stability to failure* by permitting the set of quorums that may reach fast-path decisions to be reconfigured, and that by reducing the number of such quorums we may reach fast-path decisions under any tolerated failures.

Transactional systems. Transactional storage systems typically partition data into multiple shards for scalability, and replicate each shard for fault tolerance. Transactions that span across shards hence rely on concurrency control mechanisms to achieve varying degrees of consistency. For performance reasons, however, many systems [3, 7, 10, 19,

27, 39, 46] offer isolation levels short of strict serializability, or accept scalability bottlenecks [14, 47]. ACCORD differs by featuring scalable strict serializable transactions.

Among systems that do offer strict serializable transactions, Spanner [8], CLOCC [1, 26], and Granola [9] build a transaction layer on top of a replication layer, leading to increased latency [45]. Calvin [41] and SLOG [35] similarly rely on a distinct Paxos layer to order multi-shard transactions. Tapir [32] is the first to address the over-coordination inherent to such architectures, offering strict serializable multi-shard transactions in a single wide area round-trip. Likewise, Janus [32] uses an EPaxos-like approach to provide multi-shard transactions, offering better performance than Tapir under contended workloads, albeit with limited stored procedure semantics and no stability to failure. More recently, Tempo [11] improves upon Janus by leveraging timestamps instead of explicit dependencies, at a cost of ordering non-commutative transactions.

Compared to prior systems that achieve strict serializable multi-shard transactions, ACCORD achieves optimal performance by utilizing real time timestamps and a message reorder buffer. Unlike Tempo, ACCORD does not rely on additional periodic broadcast mechanisms for timestamp stability, and commutative commands do not interfere. Importantly, ACCORD addresses the poor fast-path stability of existing systems by introducing a configurable fast-path electorate. This provides optimal failure tolerance and consistent performance under any number of tolerated failures. ACCORD is the first leaderless protocol that is sufficiently stable for practical use in a large scale industrial database system. Finally, to the best of our knowledge, no commercial or open-source database systems offer strict serializable transactions across regions in a single wide area round-trip.

References

[1] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, page 23–34, New York, NY, USA, 1995. Association for Computing Machinery.

[2] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up consensus by chasing fast decisions. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 04 2017.

[3] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, June 2013. USENIX Association.

[4] Apache Cassandra. https://cassandra.apache.org/.

[5] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

[6] CockroachDB. https://www.cockroachlabs.com.

[7] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[8] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, October 2012. USENIX Association.

[9] James Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, USA, June 2012. USENIX.

[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. 41(6):205–220, 2007.

[11] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. *Efficient Replication via Timestamp Stability*, page 178–193. Association for Computing Machinery, New York, NY, USA, 2021.

[12] Vitor Enes, Carlos Baquero, Tuanir Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In *EuroSys '20: Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 04 2020.

[13] etcd. https://etcd.io.

[14] FaunaDB. https://fauna.com.

[15] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 81–94, Renton, WA, April 2018. USENIX Association.

[16] Hazelcast. https://hazelcast.com.

[17] Heidi Howard, Aleksey Charapko, and Richard Mortier. Fast flexible paxos: Relaxing quorum intersection for fast paxos. In *International Conference on Distributed Computing and Networking 2021*, ICDCN '21, page 186–190, New York, NY, USA, 2021. Association for Computing Machinery.

[18] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. *CoRR*, abs/1608.06696, 2016.

[19] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 113–126, New York, NY, USA, 2013. Association for Computing Machinery.

[20] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[21] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[22] Leslie Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, March 2005.

[23] Leslie Lamport. Fast paxos. *Distributed Computing*, 19:79–103, October 2006.

[24] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. Technical Report MSR-TR-2009-63, May 2009.

[25] Leslie Lamport and Mike Massa. Cheap paxos. In *International Conference on Dependable Systems and Networks (DSN 2004)*, June 2004.

[26] Barbara Liskov, Miguel Castro, Liuba Shrira, and Atul Adya. Providing persistent objects in distributed systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, page 230–257, Berlin, Heidelberg, 1999. Springer-Verlag.

[27] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 401–416, New York, NY, USA, 2011. Association for Computing Machinery.

[28] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 369–384, USA, 2008. USENIX Association.

[29] MongoDB. https://www.mongodb.com.

[30] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 358–372, New York, NY, USA, 2013. Association for Computing Machinery.

[31] Iulian Moraru, David G. Andersen, and Michael Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–13, New York, NY, USA, 2014. Association for Computing Machinery.

[32] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 517–532, Savannah, GA, November 2016. USENIX Association.

[33] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, page 8–17, New York, NY, USA, 1988. Association for Computing Machinery.

[34] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[35] Kun Ren, Dennis Li, and Daniel J. Abadi. Slog: Serializable, low-latency, geo-replicated transactions. *Proc. VLDB Endow.*, 12(11):1747–1761, July 2019.

[36] RethinkDB. https://rethinkdb.com.

[37] Tuanir França Rezende and P. Sutra. Leaderless state-machine replication: Specification, properties, limits (extended version). *ArXiv*, abs/2008.02512, 2020.

[38] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[39] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 385–400, New York, NY, USA, 2011. Association for Computing Machinery.

[40] Doug Terry. Transactions and scalability in cloud databases - why can't we have both? In *Proceedings of the 18th International Workshop on High Performance Transaction Systems (HPTS)*, November 2019.

[41] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

[42] TiKV. https://tikv.org.

[43] Sarah Tollman, Seo Jin Park, and John Ousterhout. Epaxos revisited. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 613–632. USENIX Association, April 2021.

[44] YugaByte. https://www.yugabyte.com.

[45] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. *ACM Trans. Comput. Syst.*, 35(4), December 2018.

[46] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 276–291, New York, NY, USA, 2013. Association for Computing Machinery.

[47] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, page 2653–2666, New York, NY, USA, 2021. Association for Computing Machinery.

A Full Protocol Specification

Notation. $\mathbb{Q}_t^{\mathfrak{r}}$ is shorthand for $\mathbb{Q}_{t.epoch}^{\mathfrak{r}}$

Algorithm 4 Consensus Protocol (with reconfiguration and ballots)

receive τ **on coordinator** C **from client:**

$t_0 \leftarrow (Epoch, \text{now}, 0, C)$
send $PreAccept(\tau, t_0)$ to $\forall p \in \mathbb{E}^\tau$

receive $PreAccept(\tau, t_{0\tau})$ **on** p**:**

if $MaxBallot_\tau > 0$ **then**
  reply *NACK*
else if $PreAccepted_\tau \vee Accepted_\tau \vee Committed_\tau \vee Applied_\tau$ **then**
  return
else
  if $t_0.epoch = Epoch \wedge t_{0\tau} > \max(T_\gamma \mid \gamma \sim \tau) \wedge ReadyElectorate_e$ **then**
    $t_\tau \leftarrow t_{0\tau}$
  else
    $t_\tau \leftarrow \max(T_\gamma \mid \gamma \sim \tau)$
    $t_\tau.(seq, id) \leftarrow (t_\tau.seq + 1, p)$
  end if
  $T_\tau \leftarrow t_\tau$
  $PreAccepted_\tau \leftarrow$ **true**
  reply $PreAcceptOK(t_\tau, deps : \{\gamma \mid \gamma \sim \tau \wedge t_{0\gamma} < t_{0\tau}\})$
end if

receive $PreAcceptOK(t, deps)$ **from** $\mathbb{Q}^\tau_{t_0}$ **where** $\mathbb{Q}^\tau_{t_0} \not\supseteq \mathbb{Q}^\tau_t$ **:**

send $PreAccept(\tau, t_0)$ to $\forall p \in \mathbb{E}^\tau_t \setminus \mathbb{E}^\tau_{t_0}$

receive $PreAcceptOK(t, deps)$ **from** $\mathbb{Q}^\tau \supseteq \mathbb{Q}^\tau_{t_0} \cup \mathbb{Q}^\tau_t$**:**

$deps \leftarrow \bigcup\{p.deps \mid p \in \mathbb{Q}^\tau\}$
if $\exists \mathcal{F}^\tau \subseteq \mathbb{Q}^\tau \; (\forall p \in \mathcal{F}^\tau \cdot p.t = t_0)$ **then**
  send $Commit(\tau, t_0, t_0, deps)$ to $\forall p \in \rho \in \mathbb{P}^\tau_{t_0} \cup \mathbb{P}^\tau_t$
  go to Execution Protocol
else
  $t \leftarrow \max(p.t \mid p \in \mathbb{Q}^\tau)$
  send $Accept(0, \tau, t_0, t, deps)$ to $\forall p \in \rho \in \mathbb{P}^\tau_{t_0} \cup \mathbb{P}^\tau_t$
end if

receive $Accept(b, \tau, t_{0\tau}, t_\tau, deps_\tau)$**:**

if $b < MaxBallot_\tau$ **then**
  reply *NACK*
else if $Committed_\tau \vee Applied_\tau$ **then**
  return
else
  $MaxBallot_\tau \leftarrow b$
  $AcceptedBallot_\tau \leftarrow b$
  $T_\tau \leftarrow \max(t_\tau, T_\tau)$
  $Accepted_\tau \leftarrow$ **true**
  reply $AcceptOK(deps : \{\gamma \mid \gamma \sim \tau \wedge t_{0\gamma} < t_\tau\})$
end if

receive *NACK* **on** C**:**

yield to competing coordinator

receive $AcceptOK(deps)$ **on** C **from** $\mathbb{Q}^\tau_{t_0} \cup \mathbb{Q}^\tau_t$**:**

$deps \leftarrow \bigcup\{p.deps \mid p \in \mathbb{Q}^\tau\}$
send $Commit(\tau, t_0, t, deps)$ to $\forall p \in \rho \in \mathbb{P}^\tau_{t_0} \cup \mathbb{P}^\tau_t$
go to Execution Protocol

Algorithm 5 Execution Protocol (with reconfiguration and ballots)

Coordinator C:

for $\rho \in \mathbb{P}^\tau$ **do**
  $deps_\rho \leftarrow \{\gamma \mid \gamma \in deps \wedge \rho \in \mathbb{P}^\gamma\}$
  send $Read(\tau, t, deps_\rho)$ to some nearby correct $p \in \rho$
end for

receive $Commit(\tau, t_{0\tau}, t_\tau, deps_\tau)$**:**

$Committed_\tau \leftarrow$ **true**

receive $Read(\tau, t_\tau, deps_{\tau,\rho})$ **on** p**:**

await $Committed_\gamma\ \forall \gamma \in deps_{\tau,\rho}$
await $Applied_\gamma\ \forall \gamma \in deps_{\tau,\rho},\ t_\gamma < t_\tau$
$reads \leftarrow \text{read}(\tau)$
reply $ReadOK(reads)$

receive $ReadOK(reads)$ **from each shard:**

$result \leftarrow \text{execute}(\tau, reads)$
send $Apply(\tau, t, deps_{\tau,\rho}, result)$ to $\forall p \in \rho \in \mathbb{P}^\tau_t$
send *result* to client

receive $Apply(\tau, t_\tau, deps_{\tau,\rho}, result_\tau)$**:**

if $\neg Applied_\tau$ **then**
  await $Committed_\gamma\ \forall \gamma \in deps_{\tau,\rho}$
  await $Applied_\gamma\ \forall \gamma \in deps_{\tau,\rho},\ t_\gamma < t_\tau$
  $\text{apply}(writes, t_\tau)$
  $Applied_\tau \leftarrow$ **true**
end if

Algorithm 6 Reconfiguration Protocol

receive new configuration $Config_{e+1}$ **on process** p **:**

1: **await** $ReadyEpoch = e$
2: **if** $p \in \mathbb{E}_{e-1}$ **then**
3: **await** $ReadyElectorate_e$
4: **send** $JoinElectorate(\{\tau \mid t_\tau = t_{0\tau}\}, e+1)$ to $\forall p \in (\mathbb{E}_{e+1} \setminus \mathbb{E}_e)$
5: **end if**
6: **if** $p \in \mathbb{E}_{e+1} \wedge p \notin \mathbb{E}_e$ **then**
7: $ReadyElectorate_{e+1} \leftarrow$ **false**
8: **else**
9: $ReadyElectorate_{e+1} \leftarrow$ **true**
10: **end if**
11: $Epoch \leftarrow e+1$
12: $ReadyReconfigure_{e+1} \leftarrow$ **true**
13: **if** $p \in \rho_e \cup \rho_{e+1}$ **then**
14: **await** $ReadyShard$
15: **await** $Committed_\tau \cdot \forall \tau \in \{\tau \mid PreAccepted_\tau \wedge t_\tau.epoch \leq e\}$ ▷ Should only wait for those notified on join, so introduce a special flag, to prevent forward progress
16: **await** $Applied_\tau \cdot \forall \tau \in \{\tau \mid Committed_\tau \wedge t_\tau.epoch \leq e\}$
17: **send** $JoinShard(\{(\tau, *_\tau) \mid Applied_\tau \wedge t_\tau.epoch \leq e\}, e+1)$ to $\forall p \in (\rho_{e+1} \setminus \rho_e)$ ▷ TODO: also send commits that are dependencies!
18: **end if**
19: $ReadyEpoch \leftarrow e$

receive $JoinElectorate(Txns, e)$ **on replica** p **from** p' **:**

20: **await** $ReadyReconfigure_{e+1}$
21: **for** $\forall \tau \in Txns$ **do**
22: **if** $\neg PreAccepted_\tau$ **then**
23: **run** Consensus Protocol L8 to L15 ▷ *PreAccept*
24: **end if**
25: **end for**
26: $ReceivedJoinElectorate_{e+1} \leftarrow ReceivedJoinElectorate_{e+1} \cup \{p'\}$
27: $ReadyElectorate_{e+1} \leftarrow |ReceivedJoinElectorate_{e+1}| > \mathbb{E}_e - \mathcal{F}_e$

receive $JoinShard(Txns, e+1)$ **on replica** p **from** p' **:**

28: **await** $ReadyReconfigure_{e+1}$
29: **for** $\forall(\tau, *_\tau) \in Txns$ **do**
30: **if** $Applied_\tau$ **then**
31: **async receive** $Apply(\tau, t_\tau, deps_{\tau,\rho}, result_\tau)$
32: **else if** $Committed_\tau$ **then**
33: **async receive** $Commit(\tau, t_{0\tau}, t_\tau, deps_\tau)$
34: **end if**
35: **end for**
36: **await**
37: $ReceivedJoinShard_{e+1} \leftarrow ReceivedJoinShard_{e+1} \cup \{p'\}$
38: $ReadyShard \leftarrow \exists \Omega_e \subseteq ReceivedJoinShard_{e+1}$

Algorithm 7 Recovery Protocol

Coordinator C:

1: $b \leftarrow$ fresh ballot
2: **send** $Recover(b, \tau, t_0)$ to $\forall p \in \rho \in \mathbb{P}^\tau_{[}t_0]$

receive $Recover(b, \tau, t_0)$ **on** p**:**

3: **if** $b \leq MaxBallot_\tau$ **then**
4: **reply** $NACK(b_\tau)$
5: **else**
6: $MaxBallot_\tau \leftarrow b$
7: $Accepts \leftarrow \{\gamma \mid \gamma \sim \tau \wedge \tau \notin deps_\gamma \wedge Accepted_\gamma\}$
8: $Commits \leftarrow \{\gamma \mid \gamma \sim \tau \wedge \tau \notin deps_\gamma \wedge Committed_\gamma\}$
9: $Wait \leftarrow \{\gamma \in Accepts \mid t_{0\gamma} < t_{0\tau} \wedge t_\gamma > t_{0\tau}\}$
10: $Superseding \leftarrow \{\gamma \in Accepts \mid t_{0\gamma} > t_{0\tau}\}$
11: $\cup\{\gamma \in Commits \mid t_\gamma > t_{0\tau}\}$
12: **if** $\nexists PreAccepted_\tau$ **then**
13: **run** Consensus Protocol L8 to L15 ▷ *PreAccept*
14: **end if**
15: **if** $\neg Accepted_\tau \wedge \neg Committed_\tau \wedge \neg Applied_\tau$ **then**
16: $deps_\tau \leftarrow \{\gamma \mid \gamma \sim \tau \wedge t_{0\gamma} < t_{0\tau}\}$
17: **end if**
18: **reply** $RecoverOK(*_\tau, Superseding, Wait)$
19: **end if**

receive *NACK* **on** C**:**

20: **yield** to competing coordinator

receive $RecoverOK(*, Superseding, Wait)$ **from** $p \in \mathcal{R}^\tau \supseteq \mathcal{Q}^\tau_{t_0}$

21: **if** $\exists p \in \mathcal{R}^\tau\ (p.Applied_\tau)$ **then**
22: **send** $response(result)$ to client
23: **send** $Apply(\tau, t_0, p.t, p.deps, result)$ to $\forall p' \in \rho \in \mathbb{P}^\tau_t$
24: **else if** $\exists p \in \mathcal{R}^\tau\ (p.Committed_\tau)$ **then**
25: **send** $Commit(\tau, t_0, p.t, p.deps) \forall p' \in \rho \in \mathbb{P}^\tau_{[}t_0] \cup \mathbb{P}^\tau_t$
26: **go to** Execution Protocol
27: **else if** $\exists p \in \mathcal{R}^\tau\ (Accepted_\tau)$ **then**
28: **select** p with highest $AcceptedBallot_\tau$
29: $t \leftarrow p.t$; $deps \leftarrow p.deps$
30: **go to** Consensus Protocol L19 ▷ Accept
31: **else**
32: $t \leftarrow t_0$; $deps \leftarrow \bigcup\{p.deps_\tau \mid p \in \mathcal{R}^\tau\}$
33: **if** $\exists i \left|\{p \in \mathbb{E} \mid p.t > p.t_0\}\right| > |\mathbb{E}| - |\mathcal{F}_i|$ **then**
34: $t \leftarrow \max(p.t \mid p \in \mathcal{R}^\tau)$
35: **else if** $\exists p \in \mathcal{R}^\tau\ (p.Superseding \neq \emptyset)$ **then**
36: $t \leftarrow \max(p.t \mid p \in \mathcal{R}^\tau)$
37: **else if** $\bigcup\{p.Wait \mid p \in \mathcal{R}^\tau\} \neq \emptyset$ **then**
38: **await** $Committed_\gamma\ (\forall \gamma \in \bigcup\{p.Wait \mid p \in \mathcal{R}^\tau\})$
39: **restart** Recovery Protocol
40: **end if**
41: **go to** Consensus Protocol L19 ▷ Accept
42: **end if**

B Detailed Proof

We introduce the boolean predicates PREACCEPTED(P,τ), ACCEPTED(P,τ), COMMITTED(P,τ) and APPLIED(P,τ) that indicate the state of the transaction τ on replica P. Specifically, they each correspond to the boolean propertes $PreAccepted_\tau$, $Accepted(\tau)$, $Committed_\tau$ and $Applied_\tau$ on P. Each of these predicates are true iff P records the corresponding boolean property as $true$, and all following properties (in the order given) as $false$.

B.1 Validity

Theorem B.1 (Validity)**.** *Transaction τ executes and applies at replicas only if it was submitted by a client.*
Proof. A transaction may only execute after the PreAccept Phase, and may only apply after being executed. The PreAccept Phase is only performed for transactions submitted by clients. □

B.2 Isolation

B.2.1 Consistency

Observation B.1. *For any replica P and transaction τ, if $PreAccepted_\tau$ then the following are true:*

1. $P.t_{0\tau}$ *is that which is assigned by the original coordinator for* τ
2. $P.t_{0\tau}.seq = 0$
3. $P.t_{0\tau} \leq P.t_\tau$

Proof. $t_{0\tau}$ is assigned only by L1 of the Consensus Protocol. This is executed only by the original coordinator, and always assigns a sequence of 0. Candidate values for t_τ are assigned by L4 and L6 of the Consensus Protocol, which both select a value at least as large as $t_{0\tau}$. The ultimate t_τ is selected in a multitude of places, but is always either $t_{0\tau}$ or the maximum of some collection of these candidate t_τ. □

Observation B.2. *No two transactions are assigned the same t_0.*
Proof. Let γ and τ be distinct transactions. By Observation B.1, $t_{0\gamma}$ and $t_{0\tau}$ each take only the value assigned by their respective original coordinators. If $t_{0\gamma}$ and $t_{0\tau}$ are assigned by different coordinators, then the $t_{0\gamma}.id \neq t_{0\tau}.id$. Otherwise, $t_{0\gamma}.time \neq t_{0\tau}.time$ by assumption. □

Observation B.3. *For any transaction τ, and ballot b belonging to some coordinator C, C attempts to commit τ using b for at most one distinct tuple.*
Proof. The Consensus Protocol may only be invoked once and attempts only one *Commit* with a single tuple, using the special ballot b_0 that may not be used by the Recovery Protocol. By the Recovery Protocol, b must be larger than any previous ballot witnessed by a majority of replicas, so a coordinator may not execute the Recovery Protocol twice using b, and each execution of the Recovery Protocol submits at most one *Commit* tuple. □

Observation B.4. *For any replica P and conflicting transactions γ,τ, if* PREACCEPTED(P,γ) *and* PREACCEPTED(P,τ)*, with γ arriving at P before τ, then $t_\gamma \neq t_\tau$.*
Proof. By assumption, t_γ is set prior to processing *PreAccept* for τ, so $t_\tau = t_{0\tau}$ only if $t_{0\tau} > t_\gamma$; otherwise $t_\tau \geq (t_\gamma.time, t_\gamma.seq + 1, P) > t_\gamma$. □

Lemma B.1. *For any replicas P,R and conflicting transactions γ,τ, if* PREACCEPTED(P,γ) *and* PREACCEPTED(R,τ) *then $P.t_\gamma \neq R.t_\tau$.*
Proof. Let P and R be replicas that pre-accepted γ and τ respectively as described. Suppose for the sake of contradiction that $t_\gamma = t_\tau$. First, given Observation B.4 it must be that $P \neq R$. Then there are four cases to consider:

1. $t_\gamma = t_{0\gamma}$ and $t_\tau = t_{0\tau}$. This means that $t_{0\gamma} = t_{0\tau}$. However, this contradicts Observation B.2.

2. $t_\gamma = t_{0\gamma}$ but $t_\tau \neq t_{0\tau}$. This means that R reassigned t_τ during pre-accept. As a result, $t_\tau.seq \geq 1$, while by Observation B.1 $t_\gamma.seq = 0$.

3. $t_\gamma \neq t_{0\gamma}$ but $t_\tau = t_{0\tau}$. This is symmetric to the above case.

4. $t_\gamma \neq t_{0\gamma}$ and $t_\tau \neq t_{0\tau}$. This means that P and R reassigned the execution timestamp of γ and τ respectively during pre-accept. Moreover, $t_\gamma = t_\tau$ implies $t_\gamma.id = t_\tau.id$. But this means that $P = R$, which is a contradiction. □

Corollary B.1.1. *No two interfering commands commit with the same execution timestamp.*
Proof. Suppose for the sake of contradiction that γ and τ are interfering transactions that committed with the same execution timestamp. Then there must be some replica that pre-accepted γ and some replica that pre-accepted τ with that execution timestamp. This contradicts Lemma B.1. □

Lemma B.2. *Let $b_{smallest}$ be the smallest ballot number with which a transaction τ was committed at any replica. Then any other commits for τ with any ballot number $b \geq b_{smallest}$ must be of the same execution timestamp.*
Proof. Suppose τ is committed at a replica P with $t_{0\tau}, t_\tau, deps_\tau$ using ballot $b_{smallest}$. Proof by induction on ballot number b.

Base case: $b = b_{smallest}$. By design, b belongs to a single coordinator C. This means C sent $m = Commit(\tau, t_{0\tau}, t_\tau, deps_\tau)$ to P using ballot b. By Observation B.3, m is the only *Commit* message C sends to any replica using ballot b.

Inductive case: Consider some ballot number $b_1 \geq b_{smallest}$. Suppose that τ is committed using ballot number b_1 with execution timestamp t_τ. We now show that the next attempted ballot $b_2 > b_1$ will attempt τ with the same execution timestamp.

Let b_2 be the next highest ballot number than b_1 attempted for τ. Note that b_2 cannot be the default ballot for τ as there is a ballot smaller than it. Hence b_2 is attempted via the Recovery Phase, in which case there are two cases to consider:

1. τ committed via the fast-path using ballot b_1. Then τ must be pre-accepted at $\mathcal{F}^\tau$ with $t_{0\tau} = t_\tau$. Let C be the coordinator running Recovery for τ using ballot b_2. If C observes from its *RecoverOK* responses that $Committed_\tau$ or $Applied_\tau$ then we are done given the induction hypothesis together with Observation B.3. Note that C will not observe that $Accepted_\tau$ since τ was fast-path committed by ballot b_1, and by assumption there is no intermediate ballot b' where $b_1 < b' < b_2$ such that a replica could have accepted τ with b'. Hence, for all ρ, C must observe no more than $|\mathbb{E}| - |\mathcal{F}|$ replicas in $\mathbb{E}$ pre-accepted without $t_{0\tau}$ as their execution timestamps.

   It remains to show that C evaluates the condition in line 35 to false. Suppose otherwise. This means that some replica P either (a) accepted a conflicting transaction γ, such that $\tau \notin deps_\gamma$ and $t_{0\gamma} > t_{0\tau}$; or (b) committed a conflicting transaction γ such that $\tau \notin deps_\gamma$ and $t_\gamma > t_{0\tau}$.

   (a) Consider the former case where P accepted some conflicting transaction γ, such that $\tau \notin deps_\gamma$ and $t_{0\gamma} > t_{0\tau}$. Since γ has been accepted at P, it must have been pre-accepted by a slow-path quorum $\mathcal{Q}^\gamma$. Moreover, by assumption τ has been pre-accepted by a fast-path quorum $\mathcal{F}^\tau$. As such, there must be some replica $P \in \mathcal{Q}^\gamma \cap \mathcal{F}^\tau$ that pre-accepted both γ and τ. If P pre-accepted γ before τ, then τ could not have committed via the fast-path since $t_{0\gamma} > t_{0\tau}$. Otherwise, P pre-accepted τ before γ, in which case $\tau \in deps_\gamma$. Either way, we arrive at a contradiction.

   (b) Consider the latter case where P committed some conflicting transaction υ, such that $\tau \notin deps_\upsilon$ and $t_\upsilon > t_{0\tau}$. υ could not have committed via the fast-path as if $t_\upsilon = t_{0\upsilon}$ then P would have included τ in its response and so we would have $\tau \in deps_\upsilon$. Hence, υ committed via the slow-path, and there is a quorum $\mathcal{Q}^\upsilon$ that accepted υ. Moreover, by assumption τ has been pre-accepted by a fast-path quorum $\mathcal{F}^\tau$. As such, there must be some replica $R \in \mathcal{Q}^\upsilon \cap \mathcal{F}^\tau$ that both pre-accepted τ and accepted υ. If R accepted υ before pre-accepting τ, then τ could not have committed via the fast-path since $t_\upsilon > t_{0\tau}$. Otherwise, R pre-accepted τ before accepting υ, in which case $\tau \in deps_\upsilon$. Either way, we arrive at a contradiction.

2. τ committed via the slow-path using ballot b_1. This means that there is a slow-path quorum $\mathcal{Q}^\tau$ that accepted τ using ballot b_1. Let C be the coordinator running Recovery for τ using ballot b_2. By quorum intersection, C must observe a *RecoverOK* response from a replica in $\mathcal{Q}^\tau$. By assumption there is no ballot intermediate ballot $b_1 < b' < b_2$ such that a replica could have accepted τ with ballot b'. Hence, by the induction hypothesis, C

observes that τ has been accepted with execution timestamp t_τ using the largest ballot number b_1, and proceeds to attempt τ with the same execution timestamp. By Observation B.3, ballot b_2 will only be used to attempt this value. □

Lemma B.3. *If a transaction τ is committed at some replica, then for any interfering transaction γ committed at some replica with a lower execution timestamp $t_\gamma < t_\tau$, γ must be in $deps_\tau$.*
Proof. Suppose $Committed_\tau$ on some replica and $Committed_\gamma$ on some replica, where $t_\gamma < t_\tau$. There are four cases to consider:

1. γ and τ both committed via the fast-path. Then γ must be pre-accepted at some fast-path quorum $\mathcal{F}^\gamma$ with $t_{0\gamma} = t_\gamma$. Likewise, τ must be pre-accepted at some fast-path quorum $\mathcal{F}^\tau$ with $t_{0\tau} = t_\tau$. Then for any replica P in $\mathcal{F}^\gamma \cap \mathcal{F}^\tau$, P must have pre-accepted γ before pre-accepting τ, since otherwise P will not have pre-accepted γ with $t_{0\gamma} = t_\gamma < t_{0\tau}$. As a result, by the PreAccept phase of the Consensus Protocol, $\gamma \in deps_\tau$.

2. γ committed via the fast-path, while τ committed via the slow-path. Then γ must be pre-accepted at some fast-path quorum $\mathcal{F}^\gamma$ with $t_\gamma = t_{0\gamma}$. Meanwhile, τ must be accepted at some slow-path quorum $\mathcal{Q}^\tau$ with t_τ. Then for any replica P in $\mathcal{F}^\gamma \cap \mathcal{Q}^\tau$, P must have pre-accepted γ before accepting τ, since otherwise P will not have pre-accepted γ with $t_\gamma = t_{0\gamma} < t_{0\tau}$. As a result, by the Accept phase of the Consensus Protocol, $\gamma \in deps_\tau$.

3. γ committed via the slow-path, while τ committed via the fast-path. Then γ must be pre-accepted at some slow-path quorum $\mathcal{Q}^\gamma$ with each member voting for an execution timestamp $t \le t_\gamma$. Meanwhile, τ must be pre-accepted at some fast-path quorum $\mathcal{F}^\tau$ with $t_{0\tau} = t_\tau$. Then for any replica P in $\mathcal{Q}^\gamma \cap \mathcal{F}^\tau$, P must have pre-accepted γ before pre-accepting τ else P will not have pre-accepted γ with $t \le t_\gamma < t_\tau$. As a result, by the PreAccept phase of the Consensus Protocol, $\gamma \in deps_\tau$.

4. γ and τ both committed via the slow-path. Then γ must be pre-accepted at some slow-path quorum $\mathcal{Q}^\gamma$, where each replica in $\mathcal{Q}^\gamma$ pre-accepted γ for some execution timestamp $t \le t_\gamma$. Meanwhile, τ must be accepted at some slow-path quorum $\mathcal{Q}^\tau$ with t_τ. Then for any replica P in $\mathcal{Q}^\gamma \cap \mathcal{Q}^\tau$, P must have pre-accepted γ before accepting τ, since otherwise P will not have pre-accepted γ with $t \le t_\gamma < t_\tau$. As a result, by the Accept phase of the Consensus Protocol, $\gamma \in deps_\tau$.

Theorem B.2 (Execution consistency)**.** *For any two interfering transactions τ and γ, if both τ and γ are committed, then τ and γ will be applied in the same order by every replica involved in both transactions.*
Proof. Suppose $Committed_\tau$ on some replica and $Committed_\gamma$ on some replica, where $t_\gamma < t_\tau$. By Observation B.1 and Lemma B.2 every replica with $Committed_\tau$ record the same t_τ, and by Lemma B.3 $\gamma \in deps_\tau$. As a result, by the Execution Protocol, any replica participating in γ that applies τ does so only after γ has been executed and applied. □

B.2.2 Real-time order

Lemma B.4. *Let τ and γ be interfering commands such that τ is proposed by a client only after γ is committed at some replica. Then for any replicas P,R where $P.Committed_\tau$ and $R.Committed_\gamma$, $t_\tau > t_\gamma$.*
Proof. Suppose γ is committed at some replica before τ is proposed. By Observation B.1 and Lemma B.2, any replica that commits γ must do so with the same low timestamp and execution timestamps, namely $t_{0\gamma}$ and t_γ. Then there are two cases:

1. γ committed via the fast-path. Then γ must be pre-accepted by some fast-path quorum $\mathcal{F}^\gamma \supset \mathcal{Q}^\gamma$ and all $R \in \mathcal{F}^\gamma$ must now have $T_\gamma \ge t_\gamma$.

2. γ committed via the slow-path. Then γ must be accepted by some slow-path quorum $\mathcal{Q}^\gamma$ with $R.T_\gamma \ge t_\gamma$ for all $R \in \mathcal{Q}^\gamma$.

In any case, by the time τ is proposed, some members of $\mathcal{Q}^\gamma$ would have recorded the accepted t_γ and set $T_\gamma \ge t_\gamma$. Any $\mathcal{Q}^\tau$ must intersect this $\mathcal{Q}^\gamma$ at one or more correct replicas, each of which may therefore only accept $t_{0\tau}$ if it is greater than T_γ and otherwise will propose an alternative $t_\tau > T_\gamma$, ensuring that τ is committed with a higher execution timestamp than t_γ. □

Theorem B.3 (Real-time order)**.** *For any two interfering transactions τ and γ where τ is submitted after γ is committed, every replica will apply γ before τ.*

Proof. Let γ and τ be as described. Consider some replica P where $Committed_\gamma$ and $Committed_\tau$. Since γ was committed before τ was proposed, we have $t_\gamma < t_\tau$ by Lemma B.4. Moreover, by Lemma B.3, we have $\gamma \in deps_\tau$. Hence, by the Apply phase of the Execution Protocol, P applies γ before τ. □

B.3 Durability

Theorem B.4 (Durability)**.** *Committed transactions stay committed, maintaining the original real-time order.*
Proof. Consider any transaction τ that is committed by its coordinator. If $Committed_\tau$ or $Applied_\tau$ at all replicas in some $\mathcal{Q}^\tau$ then τ is durably committed. Otherwise, suppose τ has been acknowledged to the client and the coordinator crashes before a quorum of replicas have been sent their *Apply* messages. There are two cases:

1. τ committed via the fast path. Any recovery coordinator C for τ must contact some $\mathcal{R}^\tau$. τ must be pre-accepted by some fast-path quorum $\mathcal{F}^\tau$ with its committed execution timestamp t_τ, and therefore $\mathbb{E}$ for each $\rho \in \mathbb{P}$ must contain at most $|\mathbb{E}| - |\mathcal{F}|$ replicas where $t_\tau > t_{0\tau}$, and so $\mathcal{R}^\tau$ may not witness more than this. Moreover, C must observe $Superseding = \emptyset$ as by Lemma B.3 any $\upsilon \in Commits$ must contain $\tau \in deps_\upsilon$ which implies $Commits = \emptyset$, and any $\upsilon \in Accepts$ must be pre-accepted by some $\mathcal{Q}^\upsilon$ after τ else τ would have been rejected on the fast-path, so $\tau \in deps_\upsilon$ and therefore $Accepts = \emptyset$. Hence C must attempt to commit τ with the same t_τ.

2. τ committed via the slow-path. Then for some slow-path quorum $\mathcal{Q}^\tau$ each member has $Accepted_\tau$ and the committed execution timestamp t_τ. Any recovery coordinator for τ must contact at least one member of $\mathcal{Q}^\tau$ and hence resume committing τ with the same t_τ. □

B.4 Liveness

Given a partially synchronous network we may assume that messages eventually arrive at all correct processes, and the protocol assumes there is always at least one $\mathcal{Q}^\tau$ consisting of correct processes for all τ. We additionally assume that any faulty process that recovers eventually receives messages that were sent to it while it was faulty, that process clocks are loosely synchronised, and that processes join at most one shard, at most once. We finally assume the existence of a separate reliable configuration service that broadcasts new configurations sequentially along with their epochs to all correct processes and that processes start with an initial configuration e_0 and initialise $(Epoch, ReadyEpoch, ReadyElectorate, ReadyShard) \leftarrow (e_0, e_0, P \in \mathbb{E}, P \in \rho) \cdot \forall P$.

Observation B.5. *For any transaction τ, there are a finite number of transactions γ such that $t_{0\gamma} < t_{0\tau}$*
Proof. By the assumption of loosely synchronised clocks, eventually all new γ must be assigned $t_{0\gamma} > t_{0\tau}$ □

Observation B.6. *A transaction τ known to some correct process eventually commits on all correct replicas $P \in \rho \in \mathbb{P}^\tau_{t_\tau}$.*
Proof. The PreAccept, Accept and Commit phases exchange a fixed number of messages that by assumption must eventually be delivered, and execute a fixed number of synchronous steps on receipt of each message. Therefore a correct initial coordinator eventually delivers a *Commit* message to all correct replicas.

If the initial coordinator is faulty, a weak failure detector nominates a coordinator C to invoke the Recovery Protocol with a sufficiently high ballot, in which case one of the following must be true:

1. A fixed number of steps are taken before the Accept or Commit phase is invoked. By the above paragraph this will eventually commit.

2. C must wait for transactions with a lower t_0. This cannot create a circular dependency, by Observation B.5 this set is finite and so by induction, these transactions may themselves be committed.

□

Lemma B.5. *A transaction τ where all $\gamma \in deps_\tau$ with $t_\gamma < t_\tau$ have applied at all correct replicas $P \in \rho \in (\mathbb{P}^\tau_{t_\tau} \cap \mathbb{P}^\gamma_{t_\tau})$ will itself execute and apply on all correct replicas $P \in \rho \in \mathbb{P}^\tau_{t_\tau}$.*
Proof. τ commits by Observation B.6. The coordinator that commits τ invokes the Execution Protocol, issuing a *Read* to each partition. By Observation B.6 all of $deps_\tau$ commit, and by assumption all $\gamma \in deps_\tau$ where $t_\gamma < t_\tau$ have applied, so that a correct replica in each shard responds with *ReadOK* permitting the coordinator to submit *Apply* messages to all correct replicas $P \in \mathbb{P}^\tau_{t_\tau}$. By the same justification as for *Read*, these are applied at each recipient.

If the coordinator becomes faulty, a weak failure detector selects a recovery coordinator to invoke the Recovery Protocol with a sufficiently high ballot. This coordinator must either witness an incomplete *Apply* from the prior attempt and replicate it to all correct replicas, or else will commit τ and by the above paragraph apply it at all correct replicas $P \in \rho \in \mathbb{P}^{\tau}_{t_\tau}$. □

Observation B.7. *A transaction τ with committed execution timestamp t_τ does not apply at all $P \in \rho_e$ where $e < t_\tau.epoch$ and $\rho_e \cup \rho_{t_\tau} \neq \rho_e$*

Proof. By assumptions replicas may participate in at most one shard at most once, so any replica that leaves a shard may not rejoin. By the Execution Protocol only those members of ρ_{t_τ} receive *Apply*, and by the Reconfiguration Protocol τ is only propagated to members of ρ_g where $g > t_\tau$ which by assumption cannot contain any $P \in \rho_e$ where $P \notin \rho_f$ and $e < f \leq g$. □

Lemma B.6. *A transaction τ that applies at all correct replicas $P \in \rho \in \rho^{\tau}_{t_\tau}$ also applies at all correct replicas $R \in \rho \in \rho^{\tau}_{e}$ where ReadyEpoch $\geq e \geq t_\tau.epoch$ on all correct processes.*

Proof. By assumption, for all shards $\rho \in \mathbb{P}^{\tau}$ any replica in $\rho_{t_\tau} \cap \rho_e$ applies τ so we must only demonstrate that τ also applies on $P \in (\rho_e \setminus \rho_{t_\tau})$. Assume otherwise. By the Reconfiguration Protocol any $P \in (\rho_{e+1} \setminus \rho_e)$ must wait for *ReadyShard* before setting *ReadyEpoch* $= e+1$, which may only happen once some $\mathcal{Q}_e$ of *JoinShard* messages has been received and all γ applied where $Applied_\gamma$ on the sender. Since senders first commit and apply all γ where $PreAccepted_\gamma$ and $t_\gamma.epoch \leq e$, there must exist some $\mathcal{Q}_e$ where each member has either $\neg PreAccepted_\gamma$ or $t_\tau.epoch > e$. Since senders set $Epoch = e+1$ prior to this, by the Consensus Protocol any replica with $\neg PreAccepted_\tau$ must on *PreAccept* now assign $t_\tau.epoch > e$. Therefore either τ has already been committed with $t_\tau.epoch > e$ or any quorum that may pre-accept τ must involve at least one node where $t_\tau.epoch > e$, so that τ must commit with $t_\tau.epoch > e$. This contradicts our assumption and Observation B.7. □

Lemma B.7. *A transaction τ committed with $deps_\tau$, where ReadyEpoch $\geq t_\tau.epoch$ on all correct replicas, executes and applies at all correct replicas $P \in \rho \in \mathbb{P}^{\tau}_{t_\tau}$*

Proof. Let $e = t_\tau.epoch$. By the Execution Protocol, τ must wait for $deps_\tau$ to commit, and for all $\gamma \in deps_\tau$ where $t_\gamma < t_\tau$ to apply before executing and applying in a fixed number of steps.

By Observation B.6 $deps_\tau$ all commit. By Observation B.5, the transitive closure of *deps* where $t < t_\tau$ may be ordered by t to create a list where, by Lemma B.5 and Lemma B.6, any transaction in the list may execute and apply if those that precede it have themselves applied. Therefore, the first non-applied transaction in the list may execute and apply at all correct replicas, and by induction this may continue until all transactions have applied. □

Lemma B.8. *A new configuration $Config_e$ applies, setting ReadyEpoch $= e$ and $ReadyElectorate_e =$ **true** on all correct processes, and ReadyShard $=$ **true** on all correct replicas.*

Proof. By assumption $Config_e$ arrives at each correct replica, and nodes start with a correctly initialised $Config_{e_0}$. Proof by induction.

Base case: By assumption all processes are correctly initialised with $Config_{e_0}$

Inductive case: By assumption, *ReadyEpoch* $= e$ and $ReadyElectorate_e =$ **true** on all correct processes so that Lines 1 and 3 of the Reconfiguration Protocol complete. Correct members of the prior electorate therefore send *JoinElectorate* to each new member, $ReadyReconfigure_{e+1} =$ **true** will be set on every correct process, and $ReadyElectorate_{e+1}$ set on each correct process that is not *newly* a member of $\mathbb{E}_{e+1}$.

Given some $\mathcal{Q}_e$ of correct replicas by definition $|\mathbb{E}_e \cap \mathcal{Q}_e| \geq 1 + |\mathbb{E}_e| - |\mathcal{F}_e|$ so that eventually $1 + |\mathbb{E}_e| - |\mathcal{F}_e|$ *JoinElectorate* messages must be received by correct replicas that are new to the electorate. As demonstrated $ReadyReconfigure_{e+1}$ will be set by these processes so that L20 eventually completes and these *JoinElectorate* messages thereby collectively set $ReadyElectorate_{e+1} =$ **true** on each new member of $\mathbb{E}_{e+1}$, so that it is now eventually set on all correct processes.

By assumption *ReadyShard* is set for all replica $P \in \rho_e$, by Observation B.6 Line 15 of the Reconfiguration Protocol completes, and by Lemma B.7 Line 16 completes, so that *JoinShard* messages are sent and *ReadyEpoch* $= e+1$ is set on these replicas.

Some $\mathcal{Q}_e$ of correct replicas must therefore send *JoinShard* messages to all $P \in \rho_{e+1} \setminus \rho_e$ that by assumption are eventually received by each correct replica. *JoinShard* messages contain only transactions that have been applied by

the sender and their dependencies, so by definition each *Apply* on the recipient must succeed and terminate, so that *ReadyShard* is eventually set on each replica that is new to ρ_{e+1}. Therefore, by the prior paragraph, these replicas also eventually set $\mathit{ReadyEpoch} = e+1$ and this eventually holds on all correct processes. By assumption $\mathit{ReadyShard} = \textbf{true}$ holds for all correct $P \in \rho_e$, therefore $\mathit{ReadyShard} = \textbf{true}$ eventually holds on all correct $P \in \rho_{e+1}$. □

Theorem B.5. *A transaction τ known to some correct replica or coordinator commits with some t_τ and applies at all correct replicas $P \in \rho \in \rho_e^\tau$ where $e \geq t_\tau.epoch$.*

Proof. By Observation B.6 τ eventually commits with some t_τ. By the Consensus Protocol some process had $\mathit{Epoch} = t_\tau.epoch$, so the reconfiguration service has sent this configuration and by Lemma B.8 all correct processes set $\mathit{Epoch} \geq t_\tau$. By Lemma B.7 τ applies at all correct replicas $P \in \rho \in \mathbb{P}_{t_\tau}^\tau$ and by Lemma B.6 τ applies at all correct replicas $P \in \rho \in \mathbb{P}_e^\tau$ where $e \geq t_\tau.epoch$. □