Consolidating Concurrency Control and Consensus for Commits under Conflicts

Shuai Mu*, Lamont Nelson*, Wyatt Lloyd†, and Jinyang Li*
*New York University, †University of Southern California

NYU-TR-2016-983

Abstract

Conventional fault-tolerant distributed transactions layer a traditional concurrency control protocol on top of the Paxos consensus protocol. This approach provides scalability, availability, and strong consistency. When used for wide-area storage, however, this approach incurs cross-data-center coordination twice, in serial: once for concurrency control, and then once for consensus. In this paper, we make the key observation that the coordination required for concurrency control and consensus is highly similar. Specifically, each tries to ensure the serialization graph of transactions is acyclic. We exploit this insight in the design of Janus, a unified concurrency control and consensus protocol. Janus targets one-shot transactions written as stored procedures, a common, but restricted, class of transactions. Like MDCC [16] and TAPIR [50], Janus can commit unconflicted transactions in this class in one round-trip. Unlike MDCC and TAPIR, Janus avoids aborts due to contention: it commits conflicted transactions in this class in at most two round-trips as long as the network is well behaved and a majority of each server replica is alive.

We compare Janus with layered designs and TAPIR under a variety of workloads in this class. Our evaluation shows that Janus achieves ~5× the throughput of a layered system and 90% of the throughput of TAPIR under a contention-free microbenchmark. When the workloads become contended, Janus provides much lower latency and higher throughput (up to 6.8×) than the baselines.

1 Introduction

Scalable, available, and strongly consistent distributed transactions are typically achieved through layering a traditional concurrency control protocol on top of shards of a data store that are replicated by the Paxos [19, 32] consensus protocol. Sharding the data into many small subsets that can be stored and served by different servers provides scalability. Replicating each shard with consensus provides availability despite server failure. Coordinating transactions across the replicated shards with concurrency control provides strong consistency despite conflicting data accesses by different transactions. This approach is commonly used in practice. For example, Spanner [9] implements two-phase-locking (2PL) and two phase commit (2PC) in which both the data and locks are replicated by Paxos. As another example, Percolator [34] implements a variant of opportunistic concurrency control (OCC) with 2PC on top of BigTable [7] which relies on primary-backup replication and Paxos.

When used for wide-area storage, the layering approach incurs cross-data-center coordination twice, in serial: once by the concurrency control protocol to ensure transaction consistency (strict serializability [33, 44]), and another time by the consensus protocol to ensure replica consistency (linearizability [14]). Such double coordination is not necessary and can be eliminated by consolidating concurrency control and consensus into a unified protocol. MDCC [16] and TAPIR [50] showed how unified approaches can provide the read-commited and strict serializability isolation levels, respectively. Both protocols optimistically attempt to commit and replicate a transaction in one wide-area roundtrip. If there are conflicts among concurrent transactions, however, they abort and retry, which can significantly degrade performance under contention. This concern is more pronounced for wide area storage: as transactions take much longer to complete, the amount of contention rises accordingly.

This paper proposes the Janus protocol for building fault-tolerant distributed transactions. Like TAPIR and MDCC, Janus can commit and replicate transactions in one cross-data-center roundtrip when there is no contention. In the face of interference by concurrent conflicting transactions, Janus takes at most one additional cross-data-center roundtrip to commit.

The key insight of Janus is to realize that strict serializability for transaction consistency and linearizability for replication consistency can both be mapped to the same underlying abstraction. In particular, both require the execution history of transactions be equivalent to some linear total order. This equivalence can be checked by constructing a serialization graph based on the execution history. Each vertex in the graph corresponds to a transaction. Each edge represents a dependency between two transactions due to replication or conflicting data access. Specifically, if transactions T_1 and T_2 make conflicting access to some data item, then there exists an edge $T_1 \rightarrow T_2$ if a shard responsible for the conflicted data item executes *or replicates* T_1 before T_2. An equivalent linear total order for both strict serializability and linearizability can be found if there are no cycles in this graph.

Janus ensures an acyclic serialization graph in two ways. First, Janus captures a preliminary graph by tracking the dependencies of conflicting transactions as they *arrive* at the servers that replicate each shard. The coordinator of a transaction T then collects and propagates T's dependencies from sufficiently large quorums of the servers that replicate each shard. The actual execution of T is deferred until the coordinator is certain that all of T's participating shards have obtained all dependencies for T. Second, although the dependency graph based on the arrival orders of transactions may contain cycles, T's participating shards re-order transactions in cycles in the same deterministic order to ensure an acyclic serialization graph. This provides strict serializability because all servers execute conflicting transactions in the same order.

In the absence of contention, all participating shards can obtain all dependencies for T in a single cross-data-center roundtrip. In the presence of contention, two types conflicts can appear: one among different transactions and the other during replication of the same transaction. Conflicts between different transactions can still be handled in a single cross-data-center roundtrip. They manifest as cycles in the dependency graph and are handled by ensuring deterministic execution re-ordering. Conflicts that appear during replication of the same transaction, however, require a second cross-data-center roundtrip to reach consensus among different server replicas with regard to the set of dependencies for the transaction. Neither scenario requires Janus to abort a transaction. Thus, Janus always commits with at most two cross-data-center roundtrips in the face of contention as long as network and machine failures do not conspire to prevent a majority of each server replica from communicating.

Janus's basic protocol and its ability to always commit assumes a common class of transactional workloads: one-shot transactions written as stored procedures. Stored procedures are frequently used [12, 13, 15, 22, 31, 39, 42, 45] and send transaction logic to servers for execution as pieces instead of having clients execute the logic while simply reading and writing data to servers. One-shot transactions preclude the execution output of one piece being used as input for a piece that executes on a different server. One-shot transactions are sufficient for many workloads, including the popular and canonical TPC-C benchmark. While the design of Janus can be extended to support general transactions, doing so comes at the cost of additional inter-server messages and the potential for aborts.

This approach of dependency tracking and execution re-ordering has been separately applied for concurrency control [31] and consensus [20, 30]. The insight of Janus is that both concurrency control and consensus can use a common dependency graph to achieve consolidation without aborts.

We have implemented Janus in a transactional key-value storage system. To make apples-to-apples comparisons with existing protocols, we also implemented 2PL+MultiPaxos, OCC+MultiPaxos and TAPIR [50] in the same framework. We ran experiments on Amazon EC2 across multiple availability regions using microbenchmarks and the popular TPC-C benchmark [1]. Our evaluation shows that Janus achieves ~5× the throughput of layered systems and 90% of the throughput of TAPIR under a contention-free microbenchmark. When the workloads become contended due to skew or wide-area latency, Janus provides much lower latency and more throughput (up to 6.8×) than existing systems by eliminating aborts.

2 Overview

This section describes the system setup we target, reviews background and motivation, and then provides a high level overview of how Janus unifies concurrency control and consensus.

2.1 System Setup

We target the wide-area setup where data is sharded across servers and each data shard is replicated in several geographically separate data centers to ensure availability. We assume each data center maintains a full replica of data, which is a common setup [4, 26]. This assumption is not strictly needed by existing protocols [9, 50] or Janus. But, it simplifies the performance discussion in terms of the number of cross-data-center roundtrips needed to commit a transaction.

We adopt the execution model where transactions are represented as *stored procedures*, which is also commonly used [12, 13, 15, 22, 31, 39, 42, 45]. As data is sharded across machines, a transaction is made up of a series of stored procedures, referred to as *pieces*, each of which accesses data items belonging to a single shard. The execution of each piece is atomic with regard

System	Commit latency in wide-area RTTs	How to resolve conflicts during replication	How to resolve conflicts during execution/commit
2PL+MultiPaxos [9]	2*	Leader assigns ordering	Locking†
OCC+MultiPaxos [34]	2*	Leader assigns ordering	Abort and retry
Replicated Commit [28]	2	Leader assigns ordering	Abort and retry
TAPIR [50]	1	Abort and retry commit	Abort and retry
MDCC [16]	1	Retry via leader	Abort
Janus	1	Reorder	Reorder

Table 1: Wide-area commit latency and conflict resolution strategies for Janus and existing systems. MDCC provides read commited isolation, all other protocols provide strict serializability.

to other concurrent pieces through local locking. The stored-procedure execution model is crucial for Janus to achieve good performance under contention. As Janus must serialize the execution of conflicting pieces, stored procedures allows the execution to occur at the servers, which is much more efficient than if the execution was carried out by the clients over the network.

2.2 Background and Motivation

The standard approach to building fault-tolerant distributed transactions is to layer a concurrency control protocol on top of a consensus protocol used for replication. The layered design addresses three paramount challenges facing distributed transactional storage: consistency, scalability and availability.

- **Consistency** refers to the ability to preclude "bad" interleaving of concurrent conflicting operations. There are two kinds of conflicts: 1) transactions containing multiple pieces performing non-serializable data access at different data shards. 2) transactions replicating their writes to the same data shard in different orders at different replica servers. Concurrency control handles the first kind of conflict to achieve strict serializability [44]; consensus handles the second kind to achieve linearizability [14].
- **Scalability** refers to the ability to improve performance by adding more machines. Concurrency control protocols naturally achieve scalability because they operate on individual data items that can be partitioned across machines. By contrast, consensus protocols do not address scalability as their state-machine approach duplicates all operations across replica servers.
- **Availability** refers to the ability to survive and recover from machine and network failures including crashes and arbitrary message delays. Solving the availability challenge is at the heart of consensus protocols while traditional concurrency control does not address it.

∗Additional roundtrips are required if the coordinator logs the commit status durably before returning to clients.

†2PL also aborts and retries due to false positives in distributed deadlock detection.

Because concurrency control and consensus each solves only two out of the above three challenges, it is natural to layer one on top of the other to cover all bases. For example, Spanner [9] layers 2PL/2PC over Paxos. Percolator [34] layers a variant of OCC over BigTable which relies on primary-backup replication and Paxos.

The layered design is simple conceptually, but does not provide the best performance. When one protocol is layered on top of the other, the coordination required for consistency occurs twice, serially; once at the concurrency control layer, and then once again at the consensus layer. This results in extra latency that is especially costly for wide area storage. This observation has been made by TAPIR and MDCC [16, 50], which point out that layering suffers from over-coordination.

A unification of concurrency control and consensus can reduce the overhead of coordination. Such unification is possible because the consistency models for concurrency control (strict serializability [33, 44]) and consensus (linearizability [14]) share great similarity. Specifically, both try to satisfy the ordering constraints among conflicting operations to ensure equivalence to a linear total ordering. Table 1 classifies the techniques used by existing systems to handle conflicts that occur during transaction execution/commit and during replication. As shown, optimistic schemes rely on abort and retry, while conservative schemes aim to avoid costly aborts. Among the conservative schemes, consensus protocols rely on the (Paxos) leader to assign the ordering of replication operations while transaction protocols use per-item locking.

In order to unify concurrency control and consensus, one must use a *common* coordination scheme to handle both types of conflicts in the same way. This design requirement precludes leader-based coordination since distributed transactions that rely on a single leader to assign ordering [40] do not scale well. MDCC [16] and TAPIR [50] achieve unification by relying on aborts and retries for both concurrency control and consensus. Such an optimistic approach is best suited for workloads with low contention. The goal of our work is to develop a unified approach that can work well under contention by avoiding aborts.

2.3 A Unified Approach to Concurrency Control and Consensus

Can we design unify concurrency control and consensus with a common coordination strategy that minimizes costly aborts? A key observation is that the ordering constraints desired by concurrency control and replication can both be captured by the same serialization graph abstraction. In the serialization graph, vertexes represent transactions and directed edges represent the dependencies among transactions due to the two types of conflicts. Both strict serializability and linearizability can be reduced to checking that this graph is free of cycles [20, 44].

Janus attempts to explicitly capture a preliminary serialization graph by tracking the dependencies of conflicting operations without immediately executing them. Potential replication or transaction conflicts both manifest as cycles in the preliminary graph. The cycle in Figure 2a is due to conflicts during replication: T_1's write of data item X arrives before T_2's write on X at server S_x, resulting in $T_1 \rightarrow T_2$. The opposite arrival order happens at a different server replica S'_x, resulting in $T_1 \leftarrow T_2$. The cycle in Figure 2b is due to transaction conflicts: the server S_x receives T_1's write on item-x before T_2's write on item-x while server S_y receives the writes on item-y in the opposite order. In order to ensure a cycle-free serialization graph and abort-free execution, Janus deterministically re-orders transactions involved in a cycle before executing them.

To understand the advantage and challenges of unification, we contrast the workflow of Janus with that of OCC+MultiPaxos using a concrete example. The example transaction, T_1: x++; y++;, consists of two pieces, each is a stored procedure incrementing item-x or item-y.

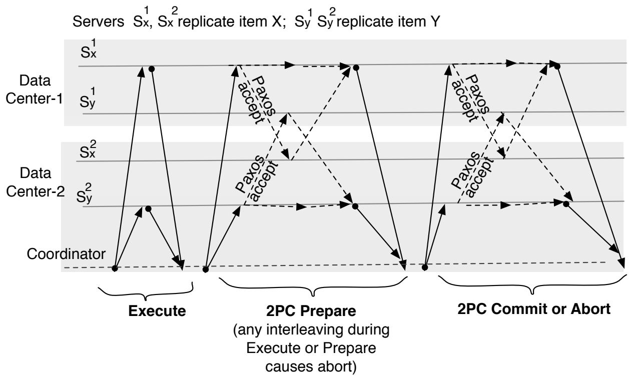
Figure 1a shows how OCC+MultiPaxos executes and commits T_1. First, the coordinator of T_1 dispatches the two pieces to item-x and item-y's respective Paxos leaders, which happen to reside in different data centers. The leaders execute the pieces and buffer the writes locally. To commit T_1, the coordinator sends 2PC-prepare requests to Paxos leaders, which perform OCC validation and replicate the new values of x and y to others. Because the pieces are stored procedures, we can combine the dispatch and 2PC-prepare messages so that the coordinator can execute and commit T_1 in two cross-data-center roundtrips in the absence of conflicts. Suppose a concurrent transaction T_2 also tries to increment the same two counters. The dispatch messages (or 2PC-prepares) of T_1 and T_2 might be processed in different orders by different Paxos leaders, causing T_1 and/or T_2 to be aborted and retried. On the other hand, because Paxos leaders impose an ordering on replication, all replicas for item-x (or item-y) process the writes of T_1 or T_2 consistently, but at the cost of an additional wide-area roundtrip.

Figure 1b shows the workflow of Janus. To commit and execute T_1, the coordinator moves through three phases: *PreAccept*, *Accept* and *Commit*, of which the *Accept* phase may be skipped. In *PreAccept*, the coordinator dispatches T_1's pieces to their corresponding replica servers. Unlike OCC+MultiPaxos, the server does not execute the pieces immediately but simply tracks their arrival orders in its local dependency graph. Servers reply to the coordinator with T_1's dependency information. The coordinator can skip the *Accept* phase if enough replica servers reply with identical dependencies, which happens when there is no contention among server replicas. Otherwise, the coordinator engages in another round of communication with servers to ensure that they obtain identical dependencies for T_1. In the *Commit* phase, each server executes transactions according to their dependencies. If there is a dependency cycle involving T_1, the server adheres to a deterministic order in executing transactions in the cycle. As soon as the coordinator hears back from the nearest server replica which typically resides in the local data center, it can return results to the user.

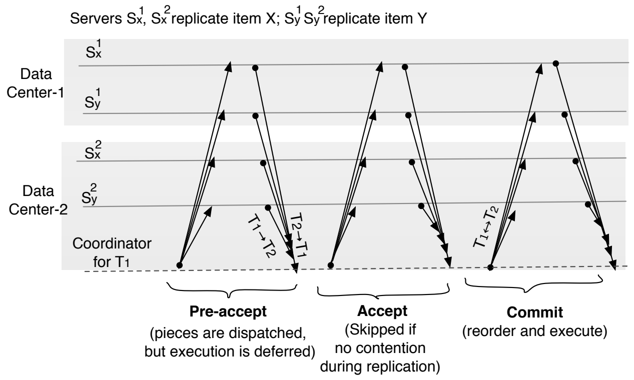
As shown in Figure 1b, Janus attempts to coordinate both transaction commit and replication in one shot with the *PreAccept* phase. In the absence of contention, Janus completes a transaction in one cross-data-center roundtrip, which is incurred by the *PreAccept* phase. (The commit phase incurs only local data-center latency.) In the presence of contention, Janus incurs two wide-area roundtrips from the *PreAccept* and *Accept* phases. These two wide-area roundtrips are sufficient for Janus to be able to resolve replication and transaction commit conflicts. The resolution is achieved by tracking the dependencies of transactions and then re-ordering the execution of pieces to avoid inconsistent interleavings. For example in Figure 1b, due to contention between T_1 and T_2, the coordinator propagates $T_1 \leftrightarrow T_2$ to all server replicas during commit. All servers detect the cycle $T_1 \leftrightarrow T_2$ and deterministically choose to execute T_1 before T_2.

3 Design

This section describes the design for the basic Janus protocol. Janus is designed to handle a common class of transactions with two constraints: 1) the transactions do not have any user-level aborts. 2) the transactions do not contain any piece whose input is dependent on the execution of another piece, i.e., they are one-shot transactions [15]. Both the example in Section 2 and the popular TPC-C workload belong to this class of transactions. We

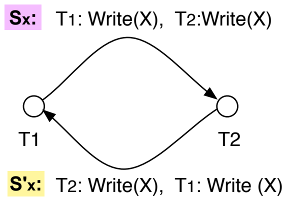

Servers S_x^1, S_x^2 replicate item X; S_y^1 S_y^2 replicate item Y
Data Center-1
Data Center-2
Coordinator
Paxos accept
Paxos accept
Paxos accept
Paxos accept
Execute
2PC Prepare (any interleaving during Execute or Prepare causes abort)
2PC Commit or Abort

(a) OCC over Multi-Paxos.

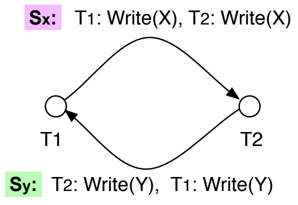

Servers S_x^1, S_x^2 replicate item X; S_y^1 S_y^2 replicate item Y
Data Center-1
Data Center-2
Coordinator for T1
T1→T2
T2→T1
T1↔T2
Pre-accept (pieces are dispatched, but execution is deferred)
Accept (Skipped if no contention during replication)
Commit (reorder and execute)

(b) Janus

Figure 1: Workflows for committing a transaction that increments x and y that are stored on different shards. The execute and 2PC-prepare messages in OCC over Multi-Paxos can be combined for stored procedures.


Sx: T1: Write(X), T2:Write(X)
T1
T2
S'x: T2: Write(X), T1: Write (X)

(a) A cycle representing replication conflicts.


Sx: T1: Write(X), T2: Write(X)
T1
T2
Sy: T2: Write(Y), T1: Write(Y)

(b) A cycle representing transaction conflicts.

Figure 2: Replication and transaction conflicts can be represented as cycles in a common serialization graph.

discuss how to extend Janus to handle general transactions and the limitations of the extension in Section 5.

3.1 Basic Protocol

The Janus system has three roles: clients, coordinators, and servers. A client sends a transaction request as a set of stored procedures to a nearby coordinator. The coordinator is responsible for communicating with servers storing the desired data shards to commit the transaction and then proxying the result to the client. We require that the shards a transaction involves to be known before the coordinator dispatches it. Coordinators have no global nor persistent state and they do not communicate with each other. In our evaluation setup, coordinators are co-located with clients.

Suppose a transaction T involves n pieces, α_1,...,α_n. Let r be the number of servers replicating each shard. Thus, there are r servers processing each piece α_i. We refer to them as α_i's participating servers. We refer to the set of $r * n$ servers for all pieces as the transaction T's participating servers.

There is a fast path and standard path of execution in Janus. When there is no contention, the coordinator takes the fast path which consists of two phases: *pre-accept* and *commit*. When there is contention, the coordinator takes the standard path of three phases: *pre-accept*, *accept* and *commit*, as shown in Figure 1b.

Dependency graphs. At a high level, the goal of *pre-accept* and *accept* phase is to propagate the necessary dependency information among participating servers. To track dependencies, each server S maintains a local dependency graph, $\mathcal{G}_S$. This is a directed graph where each vertex represents a transaction and each edge represents the chronological order between two conflicting transactions. A vertex's incoming neighbors represent the set of direct dependencies for transaction T, which is stored in the graph as T's dependency list ($\mathcal{G}_S[T].dep$). Transaction T' is called an ancestor of T if there exists a path $T' \rightsquigarrow T$. Additionally, each vertex stores the status of the transaction, which captures the stage of the protocol the transaction is currently in. The three status types are pre-accepted, accepted and committing. Servers and coordinators exchange and merge various dependency lists to ensure that the required dependencies reach the relevant servers before they execute a transaction.

In Janus, the key invariant for correctness is that all participating servers must obtain the exact same set of dependencies for a transaction T and its ancestors before executing T. This is challenging due to the interference from other concurrent conflicting transactions and from the failure recovery mechanism.

Next, we describe the fast path of the protocol, which is taken by the coordinator in the absence of interference by concurrent transactions. Pseudocode for the coordinator is shown in Algorithm 1.

The PreAccept phase. The coordinator sends *PreAccept* messages both to replicate transaction T and to establish a preliminary ordering for T among its participating servers. As shown in Algorithm 2, upon

Algorithm 1: Coordinator::DoTxn(T=[$\alpha_1, ..., \alpha_N$])

```
1  $T$'s metadata is a globally unique identifier, a shard
   list, and an abandon flag.
2  PreAccept Phase:
3  send $PreAccept(T, ballot)$ to participating servers,
   $ballot$ defaults to 0
4  if $\forall$ piece $\alpha_i \in \alpha_1 ... \alpha_N$, $\alpha_i$ has a fast quorum of
   $PreAcceptOK$s with the same dependency list $dep_i$
   then
5      $dep \leftarrow$ Union($dep_1, dep_2, ..., dep_N$)
6      goto commit phase
7  else if $\forall \alpha_i \in \alpha_1 ... \alpha_N$, $\alpha_i$ has a majority quorum of
   $PreAcceptOK$s then
8      foreach $\alpha_i \in \alpha_1 ... \alpha_N$ do
9          $dep_i \leftarrow$ Union dependencies returned by the
           majority quorum for piece $\alpha_i$.
10     $dep \leftarrow$ Union($dep_1, dep_2, ..., dep_N$)
11     goto accept phase
12 else
13     return FailureRecovery(T)
14 Accept Phase:
15 send $Accept(T, dep, ballot)$ to participating servers
16 if $\forall \alpha_i$ has a majority quorum of $Accept\text{-}OK$s then
17     goto commit phase
18 else
19     return FailureRecovery(T)
20 Commit Phase:
21 send $Commit(T, dep)$ to participating servers
22 return to client after receiving execution results.
```

Algorithm 2: Server S::PreAccept(T, ballot)

```
23 if $highest\_ballot_S[T] > ballot$ then
24     return $PreAccept\text{-}NotOK$
25 $highest\_ballot_S[T] \leftarrow ballot$
26 $\mathcal{G}_S[T].dep \leftarrow T$'s direct dependencies in $\mathcal{G}_S$
27 $\mathcal{G}_S[T].ballot \leftarrow ballot$
28 $\mathcal{G}_S[T].status \leftarrow$ pre-accepted
29 return $PreAccept\text{-}OK$, $\mathcal{G}_S[T].dep$
```

Algorithm 3: Server S::Accept(T, dep, $ballot$)

```
30 if $\mathcal{G}_S[T].status$ is committing or
31   $highest\_ballot_S[T] > ballot$ then
32     return $Accept\text{-}NotOK$, $highest\_ballot_S[T]$
33 $highest\_ballot_S[T] \leftarrow ballot$
34 $\mathcal{G}_S[T].dep \leftarrow dep$
35 $\mathcal{G}_S[T].ballot \leftarrow ballot$
36 $\mathcal{G}_S[T].status \leftarrow$ accepted
37 return $Accept\text{-}OK$
```

receiving the pre-accept message for T, server S inserts T into its local dependency graph $\mathcal{G}_s$ by creating the dependency list for T. The dependency list contains a transaction T' if T' and T both intend to make conflicting access to a common data item at server S. The server then replies *PreAccept-OK* with T's dependency list. The coordinator waits to collect a sufficient number of pre-accept replies for each piece. There are several scenarios (Algorithm 1, lines 4-11). In order to take the fast path, the coordinator must receive a *fast quorum* of replies containing the same dependency list for each piece of the transaction. A fast quorum $\mathcal{F}$ in Janus contains *all* r replica servers.

The fast quorum concept is due to Lamport who originally applied it in the Fast Paxos consensus protocol [21]. In Fast Paxos, a fast quorum must contain at least three quarters of the replicas. It allows one to skip the paxos leader, and directly send a proposed value to the replica servers. By contrast, when applying the idea to unify consensus and concurrency control, the size of the fast quorum is increased to include all replica servers. Furthermore, the fast path of Janus has the additional requirement that the dependency list returned by each server in the fast quorum must be the same. This ensures that servers obtain the same set of dependencies for T on the fast path as they do on the standard path and during failure recovery.

The Commit phase. The coordinator aggregates the dependency list of every piece and sends the result in a *Commit* message to all servers (Algorithm 1, lines 4-6). When server S receives the commit request for T (Algorithm 4), it replaces T's dependency list in its local graph $\mathcal{G}_S$ and upgrades the status of T to committing. In order to execute T, the server must ensure that all ancestors of T have the status committing. If server S participates in ancestor transaction T', it can simply wait for the commit message for T'. Otherwise, the server issues an Inquire request to the nearest server S' that participates in T' to request the dependency list of T' after T' has become committing on S'.

In the absence of contention, the dependency graph at every server is acyclic. Thus, after all the ancestors of T become committing at server S, it can perform a topological sort on the graph and executes the transaction according to the sorted order. After executing transaction T, the server marks T as processed locally. After a server executes a (piece of) transaction, it returns the result back to the coordinator. The coordinator can reply to the client as soon as it receives the necessary results

Algorithm 4: Server S::COMMIT(T, *dep*)

```
$\mathcal{G}_S[T].dep \leftarrow dep$
$\mathcal{G}_S[T].status \leftarrow$ committing
Wait & Inquire Phase:
Operator $\overset{\epsilon}{\rightsquigarrow}$ defined as a path with no tombstone
repeat
    choose $T' \overset{\epsilon}{\rightsquigarrow} T$: $\mathcal{G}_S[T'].status <$ committing
    if $T'$ does not involve $S$ then
        send $Inquire(T')$ to a server that $T'$ involves
    wait until $\mathcal{G}_S[T'].status$ is committing
until $\forall T' \overset{\epsilon}{\rightsquigarrow} T$ in $\mathcal{G}_S$: $\mathcal{G}_S[T'].status$ is committing
Execute Phase:
repeat
    choose $T' \in \mathcal{G}_S$: READYTOPROCESS($T'$)
    $scc \leftarrow$ STRONGLYCONNECTEDCOMPONENT($\mathcal{G}_S$, $T'$)
    for each $T''$ in DETERMINISTICSORT($scc$) do
        if $T''$ involves $S$ and not $T''.abandon$ then
            $T''.result \leftarrow$ execute $T''$
        $processed_S[T''] \leftarrow true$
        $\mathcal{G}_S[T''].status \leftarrow$ decided
until $processed_S[T]$ is $true$
reply $CommitOK$, $T.abandon$, $T.result$
```

Algorithm 5: Server S::READYTOPROCESS(T)

```
if $processed_S[T]$ or $\mathcal{G}_S[T].status <$ committing then
    return $false$
$scc \leftarrow$ STRONGLYCONNECTEDCOMPONENT($\mathcal{G}_S$, $T$)
for each $T' \notin scc$ and $T' \overset{\epsilon}{\rightsquigarrow} T$ do
    if $processed_S[T'] \neq true$ then
        return $false$
return $true$
```

from the nearest server replica, which usually resides in the local data center. Thus, on the fast path, a transaction can commit and execute with only one cross-data center round-trip, taken by the pre-accept phase.

3.2 Handling Contention Without Aborts

Contention manifests itself in two ways during the normal execution (Algorithm 1). As an example, consider two concurrent transactions T_1, T_2 of the form: `x++; y++`. First, the coordinator may fail to obtain a fast quorum of identical dependency lists for T_1's piece `x++` due to interference from T_2's pre-accept messages. Second, the dependency information accumulated by servers may contain cycles, e.g., with both $T_1 \rightsquigarrow T_2$ and $T_2 \rightsquigarrow T_1$ if the pre-accept messages of T_1 and T_2 arrive in different orders at different servers. Janus handles these two scenarios through the additional *accept* phase and deterministic re-ordering.

The Accept phase. If some piece does not have a fast quorum of identical dependencies during pre-accept, then consensus on T's dependencies has not yet been reached. In particular, additional dependencies for T may be inserted later, or existing dependencies may be replaced (due to the failure recovery mechanism, Section 3.3).

The accept phase tries to reach consensus via a ballot mechanism similar to the one used by Paxos. The coordinator aggregates the dependency lists returned by a majority quorum of *pre-accepts* and sends an *accept* message to all participating servers with ballot number 0 (Algorithm 1, line 15). The server handles an accept message as shown in Algorithm 3; It first checks whether T's status in its local dependency graph is committing and whether the highest ballot seen for T is greater than the one in the *accept* request. If so, the server rejects the accept request. Otherwise, the server replaces T's dependency list with the one received, updates its status and the highest ballot seen for T before replying *Accept-OK*. If the coordinator receives a majority quorum of *Accept-OKs*, it moves on to the commit phase. Otherwise, the accept phase has failed and the coordinator initiates the failure recovery mechanism (Section 3.3). In the absence of active failure recovery done by some other coordinator, the accept phase in Algorithm 1 always succeeds.

For a piece α_i, once the coordinator obtains a fast quorum of *PreAccept-OKs* with the same dependency list dep_i or a majority quorum of *Accept-OKs* accepting the dependency list dep_i, then servers have reached consensus on α_i's dependency list (dep_i). When there are concurrent conflicting transactions, the dependency lists obtained from different pieces ($dep_1, ..., dep_N$) may not be identical. The coordinator aggregates them together and sends the resulting dependencies in the commit phase.

Deterministic execution ordering. In the general case with contention, the servers can observe cyclic dependencies among T and its ancestors after waiting for all T's ancestors to advance their status to committing. In this case, the server first computes all strongly connected components (SCCs) among T and its ancestors in $\mathcal{G}_S$. It then performs a topological sort across all SCCs and executes SCCs in sorted order. Each SCC contains one or more transactions. SCCs with multiple transactions are executed in an arbitrary, but deterministic order (e.g., sorted by transaction ids). Because all servers observe the same dependency graph and deterministically order transactions within a cycle, they execute conflicting transactions in the same order.

Algorithm 6: Coordinator::FailureRecovery(T)

```
66 Prepare Phase:
67 $ballot \leftarrow$ highest ballot number seen + 1
68 send $Prepare(T, ballot)$ to participating servers
69 if $\exists$ $Tx\text{-}Done, dep$ among replies then
70     //$T$'s dependency list $dep$ has been agreed upon
71     goto commit phase
72 else if $\exists$ $\alpha_i$ without a majority quorum of
   $Prepare\text{-}OK$s then
73     goto prepare phase
74 let $\mathcal{R}$ be the set of replies with the highest ballot
75 if $\exists$ ($dep$, accepted) $\in \mathcal{R}$ then
76     goto accept phase with $dep$
77 else if $\mathcal{R}$ contains at least $\mathcal{F} \cap \mathcal{M}$ identical
   dependencies ($dep_i$, pre-accepted) for each piece
   then
78     $dep \leftarrow$Union($dep_1, dep_2, .. \, dep_N$)
79     goto accept phase with $dep$
80 else if $\exists$ ($dep$, pre-accepted) $\in \mathcal{R}$ then
81     goto pre-accept phase, but avoid the fast path
82 else
83     $T.abandon \leftarrow true$
84     goto accept phase with $dep \leftarrow nil$
```

Algorithm 7: Server S::Prepare(T, $ballot$)

```
85 $dep \leftarrow \mathcal{G}_S[T].dep$
86 if $\mathcal{G}_S[T].status \geq$ commiting then
87     return $Tx\text{-}Done, dep$
88 else if $highest\_ballot_S[T] > ballot$ then
89     return $Prepare\text{-}NotOK, highest\_ballot_S[T]$
90 $highest\_ballot_S[T] \leftarrow ballot$
91 reply $Prepare\text{-}OK, \mathcal{G}_S[T].ballot, dep, \mathcal{G}_S[T].status$
```

3.3 Handling Coordinator Failure

The coordinator may crash at anytime during protocol execution. Consequently, a participating server for transaction T will notice that T has failed to progress to the committing status for a threshold amount of time. This triggers the failure recovery process where some server acts as the coordinator to try to commit T.

The recovery coordinator progresses through *prepare*, *accept* and *commit* phases, as shown in Algorithm 6. Unlike in the normal execution (Algorithm 1), the recovery coordinator may repeat the *pre-accept*, *prepare* and *accept* phases many times using progressively higher ballot numbers, to cope with contention that arises when multiple servers try to recover the same transaction or when the original coordinator has been falsely suspected of failure. Therefore, when a server pre-accepts or accepts the dependency list of T, it also remembers its corresponding ballot number (lines 27 and 35).

In the *prepare* phase, the recovery coordinator picks a unique ballot number $b > 0$ and sends it to all of T's participating servers.[‡] Algorithm 7 shows how servers handle *prepares*. If the status of T in the local graph is committing, the server replies *Tx-Done* with T's dependency list. Otherwise, the server checks whether it has seen a ballot number for T that is higher than that included in the prepare message. If so, it rejects. Otherwise, it replies *Prepare-OK* with T's dependency list and its corresponding ballot number. If the coordinator fails to receive a majority quorum of *Prepare-OKs* for some piece, it retries the prepare phase with a higher ballot after a back-off.

The coordinator continues if it receives a majority quorum ($\mathcal{M}$) of *Prepare-OKs* for each piece. Next, it must distinguish against several cases in order to decide whether to proceed to the pre-accept or the accept phase (lines 74-84). In one case (lines 77-79), the recovery coordinator has received a majority quorum of *Prepare-OKs* with the same dependency list for each piece and T's status is pre-accepted on any of servers. In this case, transaction T could have succeeded on the fast path. Thus, the recovery coordinator unions the resulting dependencies and proceeds to the accept phase. This case also illustrates why Janus must use a fast quorum $\mathcal{F}$ containing all r server replicas. For any two conflicting transactions that both require recovery, a recovery coordinator is guaranteed to observe their dependencies only if for arbitrary majority quorums $\mathcal{M}, \mathcal{M}'$ and fast quorums $\mathcal{F}, \mathcal{F}'$ we have $(\mathcal{F} \cap \mathcal{M}) \cap (\mathcal{F}' \cap \mathcal{M}') \neq \emptyset$.

The pre-accept and accept phase used for recovery is the same as in the normal execution (Algorithm 1) except that both phases use the ballot number chosen in the prepare phase. If the coordinator receives a majority quorum of *Accept-OK*s, it proceeds to the commit phase. Otherwise, it restarts in the prepare phase using a higher ballot number.

The recovery coordinator attempts to commit transaction T if possible, but may *abandon* T if the coordinator cannot recover the inputs for T. This is possible if T's original coordinator fails to dispatch or sufficiently replicate all pieces of T. A recovery coordinator abandons a transaction T using the normal protocol except it sets the abandon flag for T during the accept and commit phase (lines 83-84). Abandoning a transaction this way ensures that servers reach consensus on abandoning T even if the the original coordinator can be falsely suspected of failure. During transaction execution, if a server encounters a transaction T with the abandon flag set, it simply skips

‡Unique server ids can be appended as low order bits to ensure unique ballot numbers.

the execution of T.

4 Optimizations

The Janus protocol described thus far is simplified in that it does not truncate nor set a boundry on the dependencies it needs to track and propagates. To commit a transaction, the simplified protocol may need to trace back the entire dependency graph, which grows over time. In this section we explain the main techniques for limiting the graph size.

4.1 Tracking Less

In the simplified protocol, when a server tracks dependencies for a new coming transaction T handling the *PreAccept* request (line 26), it inserts a direct edge $T' \rightarrow T$ for all T' that conflicts with T. We can add fewer edges. Specifically, if there already exists a path $T' \rightsquigarrow T$, where all vertexes on the path have the status committing or beyond, then we can safely omit the edge $T' \rightarrow T$.

The reason why this works is that the Janus protocol guarantees that once a transaction is committing, its dependency list *dep* is then fixed forever. Imagine there is a global dependency graph that is aggregated by all committing vertexes on each server. When we omit an edge from $T' \rightarrow T$ while there is still another path $T' \rightsquigarrow T$, it will not cause any damage to the execution order because this transitive dependency will still be captured by each server trying to commit T after the incursive inquiry phase.

The reason why this optimization could go wrong if we apply it directly for a path that contains pre-accepted and accepted vertex is that the dependencies for these two status are not fixed yet. They may possibly get emptied due to a failure recovery coordinator invovled (lines 84). Once it happens we may lose the dependency information between two conflicting transactions. Therefore we cannot simply omit edges due to path that has a (pre-)accepted vertex on it without making changes to the failure recovery algorithm. To enable this further optimization, we need to alter the failure recovery at line 84 to go to the pre-accept phase with a special placeholder *noop*. A *noop* can be considered as a sepcial transaction that conflicts with every other transaction. Therefore it will not lose any dependency information that may have been established before.

4.2 Inquiring Less

In the simplified protocol, in order to commit a transaction, we needs to wait until all its ancestors in the dependency graph to become committing. If an ancestor does not invovle this server (we define this as an *foreign* ancestor, the server needs to recursively inquire other servers to collect enough dependency information. Without proper

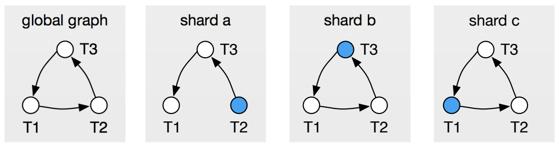

global graph
shard a
shard b
shard c
T3
T1
T2
T1: Write(a), Write(b) T2: Write(b), Write(c) T3: Write(a), Write(c)

Figure 3: An counter-example if not exchanging the entire SCC. Blue for foreign transaction at each shard.

cut-off this process essentially goes all the way back to the root (the initial state) in the graph.

We aggresively reduce the number of inquiries based the following observation. The ancestors of a foreign ancestor are often foreign ancestors too. Because these ancestors do not involve this server at all, we can safely execute the transaction without process all these foreign ancestors. In the case that the foreign ancestor has some ancestor that is not foreign, i.e., that invovles this server, we need to make sure this ancestor and decendant (the target transaction) are ordered correctly during execution. If the two transaction are non-conflicting and merely have a transitive path in the global dependency graph, the server can actually execute them in any order without breaking the isolation guarantee; If they are conflicting, however, we then need to make sure the relationship can be preserved somehow elsewhere instead of this path.

This requires an extra restriction for tracking dependencies. We can no longer omit a direct dependency $T' \rightarrow T$ simply because there is an another path $T' \rightsquigarrow T$. We can only omit the direct dependency if all the vertexes on the path $T' \rightsquigarrow T$ involves the server which is doing the tracking. In implementation this can be done without much computation overhead by tracking dependencies by keys. Each key maintains a list of ongoing transactions. A new coming transaction adds dependencies according to the lists of its accessing keys.

If the foreign ancestor and the target transaction is in the same SCC in the global dependency graph, stop inquiring at the foreign ancestor may lead to wrong result. For example in Figure 3, the correct sorting and execution order is T_1,T_2,T_3. Suppose servers in shard b and shard c first send inquiries to shard a and are able to construct the cycle and execute the transactions in the cycle following the same order T_1, T_2, T_3. After that servers in shard a send inquire to shard b about T_2. If shard b returns a hint that says T_2 is finished and shard a stops inquiring at T_2, it may execute T_3 before T_1, which is a wrong scheduling. Thus for correctness, we need to ensure that the foreign ancestor is not in the same SCC with the target transaction.

In summary of above discussion, the correct optimization works as follows. We introduce an extra status named decided, that is the highest status (> committing). decided is set after a server finishes a transaction (line 56), which indicates its position and the position of all other transactions in its enclosing SCC in the dependency graph are fixed.

When a server responds to an *Inquire* message with a decided vertex it always also includes all other vertexes in the enclosing SCC of that vertex. Servers execute SCCs atomically with respect to *Inquire* messages so all of the other vertexes in the returned SCC will also be decided. The inquiring server will *tombstone* a SCC, i.e., stop inquiring about its earlier dependencies, if none of the transactions in the SCC involves the inquiring server. Thus, tombstoning a SCC requires the SCC be fully decided and exclusive of this server. These two conditions ensures that tombstoned SCCs do not hide transitive dependencies.

5 General Transactions

This section discusses how to extend Janus to handle dependent pieces and limited forms of user aborts. Although Janus avoids aborts completely for one-shot transactions, our proposed extensions incur aborts when transactions conflict.

Let a transaction's keys be the set of data items that the transaction needs to read or write. We classify general transactions into two categories: 1) transactions whose keys are known prior to execution, but the inputs of some pieces are dependent on the execution of other pieces. 2) transactions whose keys are not known beforehand, but are dependent on the execution of certain pieces.

Transactions with pre-determined keys. For any transaction T in this category, T's set of participating servers are known apriori. This allows the coordinator to go through the *pre-accept*, *accept* phases to fix T's position in the serialization graph while deferring piece execution. Suppose servers S_i, S_j are responsible for the data shards accessed by piece α_i and α_j ($i < j$) respectively and α_j's inputs depend on the outputs of α_i. In this case, we extend the basic protocol to allow servers to communicate with each other during transaction execution. Specifically, once server S_j is ready to execute α_j, it sends a message to server S_i to wait for the execution of α_i and fetch the corresponding outputs. We can use the same technique to handle transactions with a certain type of user-aborts. Specifically, the piece α_i containing user-aborts is not dependent on the execution of any other piece that performs writes. To execute any piece α_j of the transaction, server S_j communicates with server S_i to find out α_i's execution status. If α_i is aborted, then server S_j skips the execution of α_j.

Transactions with dependent keys. As an example transaction of this type, consider transaction T(x): $y \leftarrow$Read(x); Write(y, ...) The key to be accessed by T's second piece is dependent on the value read by the first piece. Thus, the coordinator cannot go through the usual phases to fix T's position in the serialization graph without execution. We support such a transaction by requiring users to transform it to transactions in the first category using a known technique [40]. In particular, any transaction with dependent keys can be re-written into a combination of several read-only transactions that determine the unknown keys and a conditional-write transaction that performs writes if the read values match the input keys. For example, the previous transaction T can be transformed into two transactions, T1(x): $y \leftarrow$ Read(x); return y; followed by T2(x,y): $y' \leftarrow$ Read(x); if $y' \neq y$ {abort;} else {Write(y, ...);} This technique is optimistic in that it turns concurrency conflicts into user-level aborts. However, as observed in [40], real-life OLTP workloads seldom involve key-dependencies on frequently updated data and thus would incur few user-level aborts due to conflicts.

6 Implementation

In order to evaluate Janus together with existing systems and enable an apples-to-apples comparison, we built a modular software framework that facilitates the construction and debugging of a variety of concurrency control and consensus protocols efficiently.

Our software framework consists of 36,161 lines of C++ code, excluding comments and blank lines. The framework includes a custom RPC library, an in-memory database, and several benchmarks. The RPC library uses asynchronous socket I/O (`epoll/kqueue`). It can passively batch RPC messages to the same machine by reading or writing multiple RPC messages with a single system call whenever possible. The framework also provides common library functions shared by most concurrency control and consensus protocols, such as a multi-phase coordinator, quorum computation, logical timestamps, epochs, database locks, etc. By providing this common functionality, the library simplifies the task of implementing a new concurrency control or consensus protocol. For example, a TAPIR implementation required only 1,209 lines of new code, and the Janus implementation required only 2,433 lines of new code. In addition to TAPIR and Janus we also implemented 2PL+MultiPaxos and OCC+MultiPaxos.

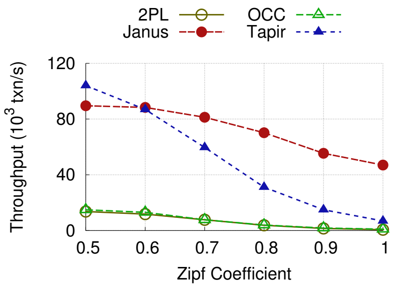
Our OCC implementation is the standard OCC with 2PC. It does not include the optimization to combine the execution phase with the 2PC-prepare phase. Our 2PL is different from conventional implementations in that it dispatches pieces in parallel in order to minimize execu-

	Oregon	Ireland	Seoul
Oregon	0.9	140	122
Ireland		0.7	243
Seoul			1.6

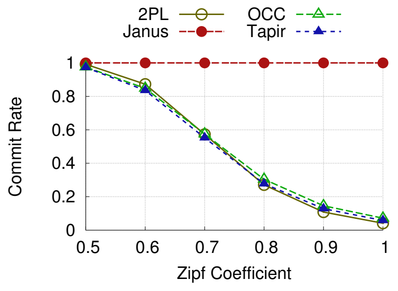
Table 2: Ping latency between EC2 datacenters (ms).

tion latency in the wide area. This increases the chance for deadlocks significantly. We use the wound-wait protocol [36] (also used by Spanner [9]) to prevent deadlocks. With a contended workload and parallel dispatch, the wound-wait mechanism results in many false positives for deadlocks. In both OCC and 2PL, the coordinator does not make a unilateral decision to abort transactions. The MultiPaxos implementation inherits the common optimization of batching many messages to and from leaders from the passive batching mechanism in the RPC library.

Apart from the design described in Section 3, our implementation of Janus also includes the garbage collection mechanism to truncate the dependency graph as transactions finish. We have not implemented Janus's coordinator failure recovery mechanism nor the extensions to handle dependent pieces. Our implementations of 2PL/OCC+MultiPaxos and TAPIR also do not include failure recovery.



(a) Cluster Throughput



(b) Commit Rates

Figure 4: Single datacenter performance with increasing contention as controlled by the Zipf coefficient.

7 Evaluation

Our evaluation aims to understand the strength and limitations of the consolidated approach of Janus. How does it compare against conventional layered systems? How does it compare with TAPIR's unified approach, which aborts under contention? The highlights include:

- In a single data center with no contention, Janus achieves 5× the throughput of 2PL/OCC+MultiPaxos, and 90% of TAPIR's throughput in microbenchmarks.
- All systems' throughput decrease as contention rises. However, as Janus avoids aborts, its performance under moderate or high contention is better than existing systems.
- Wide-area latency leads to higher contention. Thus, the relative performance difference between Janus and the other systems is higher in multi-data-center experiments than in single-data-center ones.

7.1 Experimental Setup

Testbed. We run all experiments on Amazon EC2 using m4.large instance types. Each node has 2 virtual CPU cores, 8GB RAM. For geo-replicated experiments, we use 3 EC2 availability regions, us-west-2 (Oregon), ap-northeast-2 (Seoul) and eu-west-1 (Ireland). The ping latencies among these data centers are shown in Table 2.

Experimental parameters. We adopt the configuration used by TAPIR [50] where a separate server replica group handles each data shard. Each microbenchmark uses 3 shards with a replication level of 3, resulting in a total of 9 server machines being used. In this setting, a majority quorum contains at least 2 servers and a fast quorum must contain all 3 servers. When running geo-replicated experiments, each server replica resides in a different data center. The TPC-C benchmark uses 6 shards with a replication level of 3, for a total of 18 processes running on 9 server machines.

We use closed-loop clients: each client thread issues one transaction at a time back-to-back. Aborted transactions are retried for up to 20 attempts. We vary the injected load by changing the number of clients. We run client processes on a separate set of EC2 instances than servers. In the geo-replicated setting, experiments of 2PL and OCC co-locate the clients in the datacenter containing the underlying MultiPaxos leader. Clients of Janus and Tapir are spread evenly across the three datacenters. We use enough EC2 machines such that clients are not bottlenecked.

Other measurement details. Each of our experiment lasts 30 seconds, with the first 7.5 and last 7.5 seconds excluded from results to avoid start-up, cool-down, and potential client synchronization artifacts. For each set of experiments, we report the throughput, 90th percentile latency, and commit rate. The system throughput is calculated as the number of committed transactions divided by the measurement duration and is expressed in transactions per second (tps). We calculate the latency of a transaction as the amount of time taken for it to commit, including retries. A transaction that is given up after 20 retries is considered to have infinite latency. We calculate the commit rate as the total number of committed transactions divided by the total number of commit attempts (including retries).

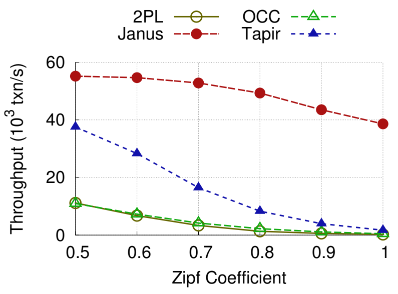
Calibrating TAPIR's performance. Because we re-implemented the TAPIR protocol using a different code base, we calibrated our results by running the most basic experiment (one-key read-modify-write transactions), as presented in Figure 7 of TAPIR's technical report [48]. The experiments run within a single data center with only one shard. The keys are chosen uniformly randomly. Our TAPIR implementation (running on EC2) achieves a peak throughput of ~165.83K tps, which is much higher than the amount (~8K tps) reported for Google VMs [48]. We also did the experiment corresponding to Figure 13 of [50] and the results show a similar abort rate performance for TAPIR and OCC (TAPIR's abort rate is an order of magnitude lower than OCC with lower zipf coefficients). Therefore, we believe our TAPIR implementation is representative.

7.2 Microbenchmark

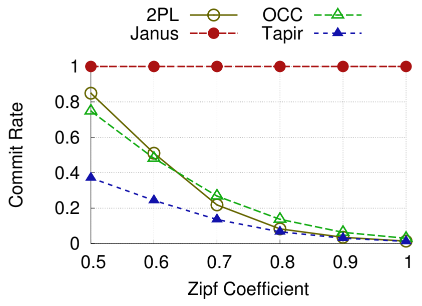
Workload. In the microbenchmark, each transaction performs 3 read-write access on different shards by incrementing 3 randomly chosen key-value pairs. We pre-populate each shard with 1 million key-value pairs before starting the experiments. We vary the amount of contention in the system by choosing keys according to a zipf distribution with a varying zipf coefficient. The larger the zipf coefficient, the higher the contention. This type of microbenchmark is commonly used in evaluations of transactional systems [9, 50].

Single data center (low contention). Figure 4 shows the single data center performance of different systems when the zipf coefficient increases from 0.5 to 1.0. Zipf coefficients of 0.0~0.5 are excluded because they have negligible contention and similar performance to zipf=0.5.

In these experiments, we use 900 clients to inject load. The number of clients is chosen to be on the "knee" of the latency-throughput curve of TAPIR for a specific zipf value (0.5). In other words, using more than 900 clients results in significantly increased latency with only small throughput improvements. With zipf coefficients of 0.5~0.6, the system experiences negligible amounts of contention. As Figure 4b shows, the commit rates across all systems are almost 1 with zipf coefficient 0.5.



(a) Cluster Throughput

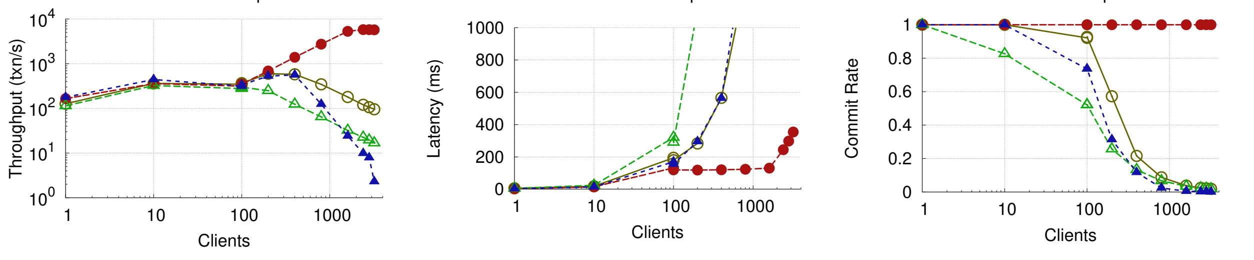


(b) Commit Rates

Figure 5: Geo-replicated performance with increasing contention as controlled by the Zipf coefficient.

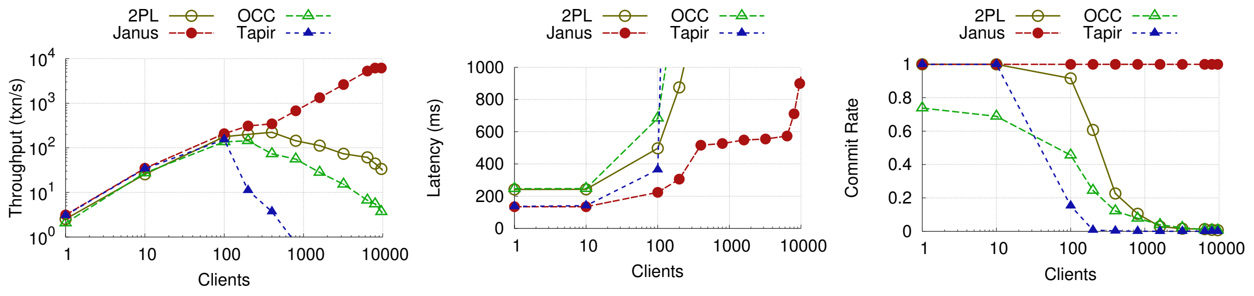
Both TAPIR and Janus achieve much higher throughput than 2PL/OCC+MultiPaxos. As a Paxos leader needs to handle much more communication load (>3×) than non-leader server replicas, 2PL/OCC's performance is bottlenecked by Paxos leaders, which are only one-third of all 9 server machines.

Single data center (moderate to high contention). As the zipf value varies from 0.6 to 1.0, the amount of contention in the workload rises. As seen in Figure 4b, the commit rates of TAPIR, 2PL and OCC decrease quickly as the zipf value increases from 0.6 to 1.0. At first glance, it is surprising that 2PL's commit rate is no higher than OCC's. This is because our 2PL implementation dispatches pieces in parallel. This combined with the large amounts of false positives induced by deadlock detection, makes locking ineffective. By contrast, Janus does not incur any aborts and maintains a 100% commit rate. Increasing amounts of aborts result in significantly lower throughput for TAPIR, 2PL and OCC. Although Janus does not have aborts, its throughput also drops because the size of the dependency graph that is being maintained and exchanged grows with the amount of the contention. Nevertheless, the overhead of graph computation is much less than the cost of aborts/retries, which allows Janus to outperform existing systems. At zipf=0.9, the throughput of Janus (55.41K tps) is 3.7× that of TAPIR (14.91K tps). The corresponding 90-percentile latency for Janus and TAPIR is 24.65ms and 28.90ms, respectively. The



(a) Cluster Throughput **(b) 90% Latency** **(c) Commit Rates**

Figure 6: Performance in the single datacenter setting for the TPC-C benchmark with increasing load and contention as controlled by the number of clients per partition.



(a) Cluster Throughput **(b) 90% Latency** **(c) Commit Rates**

Figure 7: Performance in the geo-replicated setting for the TPC-C benchmark with increasing load and contention as controlled by the number of clients per partition.

latter is higher due to repeated retries.

Multiple data centers (moderate to high contention). We move on to experiments that replicate data across multiple data centers. In this set of experiments, we use 10800 clients to inject load, compared to 900 for in the single data center setting. As the wide-area communication latency is more than an order of magnitude larger, we have to use many more concurrent requests in order to achieve high throughput. Thus, the amount of contention for a given zipf value is much higher in multi-datacenter experiments than that of single-data-center experiments. As we can see in Figure 5b, at zipf=0.5, the commit rate is only 0.37 for TAPIR. This causes the throughput of TAPIR to be lower than Janus at zipf=0.5 (Figure 5a). At larger zipf values, e.g., zipf=0.9, the throughput of Janus drops to 43.51K tps, compared to 3.95K tps for TAPIR, and ~1085 tps for 2PL and OCC. As Figure 5b shows, TAPIR's commit rate is slightly lower than that of 2PL/OCC. Interference during replication, apart from aborts due to transaction conflicts, may also cause TAPIR to retry the commit.

	new-order	**payment**	**order-status**	**delivery**	**stock-level**
ratio	43.65%	44.05%	4.13%	3.99%	4.18%

Table 3: TPC-C mix ratio in a Janus trial.

7.3 TPC-C

Workload. As the most commonly accepted benchmark for testing OLTP systems, TPC-C has 9 tables and 92 columns in total. It has five transaction types, three of which are read-write transactions and two are read-only transactions. Table 3 shows a commit transaction ratio in one of our trials. In this test we use 6 shards each replicated in 3 data centers. The workload is sharded by warehouse; each shard contains 1 warehouse; each warehouse contains 10 districts, following the specification. Because each new-order transaction needs to do a read-modify-write operation on the next-order-id that is unique in a district, this workload exposes very high contention with increasing numbers of clients.

Single datacenter setting. Figure 6 shows the single data center performance of different systems when the number of clients is increased. As the number of clients increases, the throughput of Janus climbs until it saturates the servers' CPU, and then stabilizes at 5.78K tps. On the other hand, the throughput of other systems will first climb, and then drop due to high abort rates incurred as contention increases. TAPIR's the throughput peaks at 560 tps; The peak throughput for for 2PL/OCC is lower than 595/324 tps. Because TAPIR can abort and retry more quickly than OCC, its commit rate becomes lower than OCC as the contention increases. Because of the massive number of aborts, the 90-percentile latency for 2PL/OCC/TAPIR are more than several seconds when the number of clients increases; by contrast, the latency of Janus remains relatively low (<400ms). The latency of Janus increases with the number of clients due to as more outstanding transactions result in increased queueing.

Multi-data center setting. When replicating data across multiple data centers, more clients are needed to saturate the servers. As shown in Figure 7, the throughput of Janus (5.7K tps) is significantly higher than the other protocols. In this setup, the other systems show the same trend as in single data-center but the peak throughput becomes lower because the amount of contention increases dramatically with increased number of clients. TAPIR's commit rate is lower than OCC's when there are $\geq$ 100 clients because the wide-area setting leads to more aborts for its inconsistent replication.

Figure 7b shows the 90-percentile latency. When the number of clients is <= 100, the experiments use only a single client machine located in the Oregon data center. When the number of clients > 100, the clients are spread across all three data centers. As seen in Figure 7b, when the contention is low (<10 clients), the latency of TAPIR and Janus is low (less than 150ms) because both protocols can commit in one cross-data center roundtrip from Oregon. By contrast, the latency of 2PL/OCC is much more (230ms) as their commits require 2 cross data center roundtrips. In our experiments, all leaders of the underlying MultiPaxos happen to be co-located in the same data center as the clients. Thus, 2PL/OCC both require only one cross-data center roundtrip to complete the 2PC prepare phase and another roundtrip for the commit phase as our implementation of 2PL/OCC only returns to clients after the commit to reduce contention. As contention rises due to increased number of clients, the 90-percentile latency of 2PL/OCC/TAPIR increases quickly to tens of seconds. The latency of Janus also rises due to increased overhead in graph exchange and computation at higher levels of contention. Nevertheless, Janus achieves a decent 90-percentile latency (<900ms) as the number of clients increases to 10,000.

8 Related Work

We review related work in fault-tolerant replication, geo-replication, and concurrency control.

Fault-tolerant replication. The replicated state machine (RSM) approach [17, 37] enables multiple machines to maintain the same state through deterministic execution of the same commands and can be used to mask the failure of individual machines. Paxos [18] and viewstamped replication [24, 32] showed it was possible to implement RSMs that are always safe and remain live when f or fewer out of $2f + 1$ total machines fail. Paxos and viewstamped replication solve consensus for the series of commands the RSM executes. Fast Paxos [21] reduced the latency for consensus by having client send commands directly to replicas instead of a through a distinguished proposer. Generalized Paxos [20] builds on Fast Paxos by further enabling commands to commit out of order when they do not interfere, i.e., conflict. Janus uses this insight from Generalized Paxos to avoid specifying an order for non-conflicting transactions.

Mencius [29] showed how to reach consensus with low latency under low load and high throughput under high load in a geo-replicated setting by efficiently round-robining leader duties between replicas. Speculative Paxos [35] can achieve consensus in a single round trip by exploiting a co-designed datacenter network for consistent ordering. EPaxos [30] is the consensus protocol most related to our work. EPaxos builds on Mencius, Fast Paxos, and Generalized Paxos to achieve (near-)optimal commit latency in the wide-area, high throughput, and tolerance to slow nodes. EPaxos's use of dependency graphs to dynamically order commands based on their arrival order at different replicas inspired Janus's dependency tracking for replication.

Janus addresses a different problem than RSM because it provides fault tolerance and scalability to many shards. RSMs are designed to replicate a single shard and when used across shards they are still limited to the throughput of a single machine.

Geo-replication. Many recent systems have been designed for the geo-replicated setting with varying degrees of consistency guarantees and transaction support. Dynamo [11], PNUTS [8], and TAO [6] provide eventual consistency and avoid wide-area messages for most operations. COPS [26] and Eiger [27] provide causal consistency, read-only transactions, and Eiger provides write-only transactions while always avoiding wide-area messages. Walter [38] and Lynx [51] often avoid wide-area messages and provide general transactions with parallel snapshot isolation and serializability respectively. All of these systems will typically provide lower latency than Janus because they made a different choice in the fundamental trade off between latency and consistency [5, 23]. Janus is on the other side of that divide and provides strict serializability and general transactions.

Concurrency control. Fault tolerant, scalable systems typically layer a concurrency control protocol on top of a replication protocol. Sinfonia [3] piggybacks OCC into 2PC over primary-backup replication. Percolator [34] also uses OCC over primary-backup replication. Spanner [9] uses 2PL with wound wait over Multi-Paxos and inspired our 2PL experimental baseline.

CLOCC [2, 25] using fine-grained optimistic concurrency control using loosely synchronized clocks over viewstamped replication. Granola [10] is optimized for single shard transactions, but also includes a global transaction protocol that uses a custom 2PC over viewstamped replication. Calvin [41] uses a sequencing layer and 2PL over Paxos. Salt [46] is a concurrency control protocol for mixing acid and base transactions that inherits MySQL Cluster's chain replication [43]. Salt's successor, Callas, introduces modular concurrency control [47] that enables different types of concurrency control for different parts of a workload.

Replicated Commit [28] executes Paxos over 2PC instead of the typical 2PC over Paxos to reduce wide-area messages. Rococo [31] is a dependency graph based concurrency control protocol over Paxos that avoids aborts by reordering conflicting transactions. Rococo's protocol inspired our use of dependency tracking for concurrency control. All of these systems layer a concurrency control protocol over a separate replication protocol, which incurs coordination twice in serial.

MDCC [16] and TAPIR [48, 49, 50] are the most related systems and we have discussed them extensively throughout the paper. Their fast fault tolerant transaction commits under low contention inspired us to work on fast commits under all levels of contention, which is our biggest distinction from them.

9 Conclusion

We presented the design, implementation, and evaluation of Janus, a new protocol for fault tolerant distributed transactions that are one-shot and written as stored procedures. The key insight behind Janus is that the coordination required for concurrency control and consensus is highly similar. We exploit this insight by tracking conflicting arrival orders of both different transactions across shards and the same transaction within a shard using a single dependency graph. This enables Janus to commit in a single round trip when there is no contention. When there is contention Janus is able to commit by having all shards of a transaction reach consensus on its dependencies and then breaking cycles through deterministic reordering before execution. This enables Janus to provide higher throughput and lower latency than the state of the art when workloads have moderate to high skew, are geo-replicated, and are realistically complex.

Acknowledgments

The work is supported by NSF grants CNS-1514422 and CNS-1218117 as well as AFOSR grant FA9550-15-1-0302. Lamont Nelson is also supported by an NSF graduate fellowship. We thank Michael Walfish for helping us clarify the design of Janus and Frank Dabek for reading an early draft. We thank our shepherd, Dan Ports, and the anonymous reviewers of the OSDI program committee for their helpful comments.

References

[1] TPC-C Benchmark. http://www.tpc.org/tpcc/.

[2] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 1995.

[3] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[4] Amazon. Cross-Region Replication Using DynamoDB Streams. http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.CrossRegionRepl.html, 2016.

[5] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems (TOCS)*, 12(2), 1994.

[6] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, 2013.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of USENIX Symposium on Opearting Systems Design and Implementation (OSDI)*, 2006.

[8] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2008.

[9] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. In *Proceedings of USENIX Sympo-*

sium on Opearting Systems Design and Implementation (OSDI), 2012.

[10] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, 2012.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[12] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *Knowledge and Data Engineering, IEEE Transactions on*, 4(6):509–516, 1992.

[13] H. Garcia-Molina, R. J. Lipton, and J. Valdes. A massive memory machine. *Computers, IEEE Transactions on*, 100 (5):391–399, 1984.

[14] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12 (3):463–492, 1990.

[15] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2008.

[16] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, 2013.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 1978.

[18] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

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

[20] L. Lamport. Generalized consensus and Paxos. Technical report, Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[21] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2): 79–103, October 2006.

[22] K. Li and J. F. Naughton. Multiprocessor main memory transaction processing. In *Proceedings of the first international symposium on Databases in parallel and distributed systems*, 2000.

[23] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report TR-180-88, Princeton Univ., Dept. Comp. Sci., 1988.

[24] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical report, MIT-CSAIL-TR-2012-021, MIT, 2012.

[25] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *ECOOP*, 1999.

[26] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[27] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[28] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2013.

[29] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of USENIX Symposium on Opearting Systems Design and Implementation (OSDI)*, 2008.

[30] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[31] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *Proceedings of USENIX Symposium on Opearting Systems Design and Implementation (OSDI)*, 2014.

[32] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, 1988.

[33] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4), 1979.

[34] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of USENIX Symposium on Opearting Systems Design and Implementation (OSDI)*, 2010.

[35] D. R. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2015.

[36] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems (TODS)*, 3(2):178–198, 1978.

[37] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computer Surveys*, 22(4), Dec. 1990.

[38] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[39] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2007.

[40] A. Thomson and D. J. Abadi. The case for determin-

ism in database systems. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2010.

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

[42] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[43] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of USENIX Symposium on Opearting Systems Design and Implementation (OSDI)*, 2004.

[44] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.

[45] A. Whitney, D. Shasha, and S. Apter. High volume transaction processing without concurrency control, two phase commit, sql or C++. In *Seventh International Workshop on High Performance Transaction Systems, Asilomar*, 1997.

[46] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining ACID and BASE in a distributed batabase. In *Proceedings of USENIX Symposium on Opearting Systems Design and Implementation (OSDI)*, 2014.

[47] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance ACID via modular concurrency control. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[48] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports. Building consistent transactions with inconsistent replication. Technical report, Technical Report UW-CSE-2014-12-01, University of Washington, 2014.

[49] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports. Building consistent transactions with inconsistent replication (extended version). Technical report, Technical Report UW-CSE-2014-12-01 v2, University of Washington, 2015.

[50] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[51] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2013.