Natto: Providing Distributed Transaction Prioritization for High-Contention Workloads

Linguan Yang
l69yang@uwaterloo.ca
University of Waterloo

Xinan Yan
xinan.yan@uwaterloo.ca
University of Waterloo

Bernard Wong
bernard@uwaterloo.ca
University of Waterloo

ABSTRACT

This paper introduces Natto, a geo-distributed database system that supports transaction prioritization. Instead of having each shard process transactions in their arrival order, Natto leverages network measurements to estimate the transaction arrival time at each shard, and assigns a timestamp to the transaction based on its arrival time to the furthest shard. These timestamps establish a global ordering of transactions, and introduces opportunities to selectively abort pending low-priority transactions that conflict with a high-priority transaction, or even preempt transactions that are already partially prepared. Our experiments on both Microsoft Azure and a local cluster show that Natto's tail latency for high-priority transactions are significantly lower than the tail latencies of Carousel and TAPIR, which are the current state-of-the-art in geo-distributed transaction processing systems.

CCS CONCEPTS

• **Information systems → Distributed database transactions**.

KEYWORDS

transaction prioritization, geo-distributed transactions

ACM Reference Format:
Linguan Yang, Xinan Yan, and Bernard Wong. 2022. Natto: Providing Distributed Transaction Prioritization for High-Contention Workloads. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3526161

1 INTRODUCTION

Priority-based scheduling is critically important for database systems that process different classes of transactions. By assigning a high priority to a time-sensitive transaction, the transaction's completion time should largely be unaffected by concurrent low-priority batch transactions. In a single-server database system, priority scheduling is relatively straightforward to implement. Most implementations create a separate queue per priority level and process transactions starting from the highest priority queue. In-progress transactions can optionally be preempted to further reduce the wait time for a high-priority transaction.

However, this single-server approach for providing priority scheduling cannot be easily extended to distributed database systems, such as Spanner [17] and CockroachDB [15], that process geo-distributed data. In these systems, data is partitioned and replicated across datacenters in different geographic locations, and transactions are scheduled independently at each data partition. Without a global view of concurrent transactions, the effectiveness of priority scheduling is fairly limited. This is because the arrival order of transactions will often be different at each partition. A high-priority transaction may only be scheduled ahead of a conflicting low-priority transaction for some partitions, resulting in a potential distributed deadlock that has to be resolved by aborting one or both transactions. Preemption of partially-prepared transactions is also generally not supported in these systems due to both the complexity of performing distributed preemption, and the high latency to confirm that a transaction has successfully been preempted. A high-priority transaction must wait until conflicting partially-prepared transactions are complete before it can be processed.

Systems with a logically centralized transaction sequencer, such as Calvin [49] and FaunaDB [26], can be extended to support priority scheduling as their schedulers are given a complete global view of the transactions in the system by their sequencers. However, employing such a sequencer introduces an extra wide-area network round trip to process a transaction, which is unacceptable for some time-sensitive transactions. These systems also introduce other restrictions, such as requiring transactions to be non-interactive and deterministic, making them unsuitable for certain classes of transactions.

In this paper, we introduce Natto, a geo-distributed database system that can significantly reduce the tail latency of high-priority transactions through transaction prioritization. Natto builds on the Carousel [53] database system in which data is partitioned and stored at the datacenter where it will most frequently be used. Partitions are also replicated using Raft [41] to additional datacenters to provide fault tolerance. Similar to Carousel, Natto targets 2-round Fixed-set Interactive (2FI) transactions, where each transaction consists of a read round followed by a write round, the read and write sets are known at the start of the transaction but the write values can depend on the read results, and users can choose to abort their transactions after the first round.

Natto introduces or extends several techniques that build on each other to provide effective and efficient transaction prioritization. First, it uses network measurements from clients to servers to accurately estimate the arrival time of a transaction at the participating Natto servers. Each transaction is assigned a timestamp based on its estimated arrival time at the furthest participanting server, and the transaction is not processed on any server until that time. This approach was first introduced in Domino [52] for use

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526161

in state machine replication. Natto extends it to establish a global ordering of transactions without the need for a logically centralized sequencer or coordination messages between servers. Even though a transaction is not processed immediately on nearby servers, its completion time does not increase since the second round of the transaction cannot start until the client receives all of the results from the first round, including the result from the furthest server. Using this approach, Natto does not acquire resources prematurely, and its transactions cannot deadlock as the global ordering prevents cyclic dependencies.

Although the Natto servers agree on the ordering of transactions, each server only receives transactions that read and/or write to its partition. Therefore, they cannot unilaterally schedule a high-priority transaction ahead of a low-priority transaction without potentially introducing a distributed deadlock. However, for a given transaction, all servers other than the furthest one from the client will receive the transaction before its execution time. This creates an *abort window* at these servers during which time the transaction has been received but may still be aborted if it conflicts with a new high priority transaction. This abort window comes out of our use of arrival time-based execution timestamps, and gives Natto an opportunity to perform a *priority abort* of a low priority transactions that can interfere with high priority transactions without introducing any delays when there are no conflicts.

Additionally, each Natto transaction embeds its estimated transaction arrival time to all of the participating servers. Without this information, if a high priority transaction arrives at a Natto server that conflicts with an already prepared low priority transaction, the high priority transaction must wait until the low priority transaction completes even if the low priority transaction is eventually aborted. However, by having the arrival time information to all participating servers for its received transactions, a Natto server can in some cases know that a prepared low-priority transaction will likely be aborted. With this knowledge, it can send a *conditional prepare* message to the 2PC coordinator, which prepares the high-priority transaction before the server receives an abort acknowledgement for the conflicting transaction. This conditional prepare will be successful if the conflicting low-priority transaction is aborted. Otherwise, Natto will safely discard the conditional prepare results and prepare the high-priority transaction after the conflicting transaction completes successfully.

Finally, in the Carousel protocol, results from a committed transaction are not visible to other transactions until they have been replicated to a majority of the replicas. This is necessary since, in the event of a replication group failure, committed transactions that have not been replicated must be replayed, and a Carousel server cannot determine the ordering of multiple transactions in this state. With the availability of transaction timestamps, Natto can determine the order of transactions in the committed but not yet replicated state. This allows Natto to introduce *Early Committed State Forwarding (ECSF)* that allows a transaction to read committed results that have not been replicated. ECSF significantly reduces lock contention by reducing the amount of time that a transaction needs to hold a lock by one wide-area network roundtrip. This significantly reduces the latency of both high and low-priority transactions for high-contention workloads.

This paper makes three main contributions:

- We describe the use of network measurements to establish a global ordering of transactions based on transaction arrival time. We use this ordering to identify and selectively abort low-priority transactions that would delay the processing of a high-priority transaction.
- We introduce conditional prepare and early committed result forwarding, which allow Natto to further overlaps operations that are performed sequentially in previous systems.
- We evaluate Natto using the Smallbank [13, 20], Retwis [34, 54] and YCSB+T [19] on both an Microsoft Azure deployment and a local cluster. Our results show that Natto has significantly lower tail latency for high-priority transactions compared to Carousel [53] and TAPIR [54].

2 BACKGROUND

We design Natto on top of Carousel [53], a transaction processing system for globally distributed data. Unlike Carousel that processes transactions in their arrival order, Natto introduces a timestamp-based ordering to support transaction prioritization. Natto assigns a timestamp to each transaction, where the timestamp indicates when the transaction should have arrived at all participants. To estimate a transaction's arrival time at participants, Natto uses techniques introduced in Domino [52]. This section will first briefly review Carousel, and then describe the techniques that Natto borrows from Domino for estimating a transaction's arrival time at participants.

2.1 Carousel

Similar to many other geo-distributed database systems, such as Google Spanner [17] and CockroachDB [15], Carousel shards data into partitions to achieve scalability, and replicates each partition in different datacenters to tolerate datacenter-wide failures. Unlike other systems, Carousel targets a specific type of read-write transactions, 2-round Fixed-set Interactive (2FI) transactions [53].

A 2FI transaction consists of one round of reads followed by a round of writes. Both read and write keys are pre-defined at the start of a 2FI transaction. However, a client can decide the write values based on the read results and does not need to modify all of the keys in the write set. This interactive read-write pattern between clients and servers is preferred by many applications [42], especially in rapid development [10]. The 2FI model can directly implement common read-modify-write patterns in transactions, such as reading, incrementing, and updating one or more counter values. This allows many transactions to fit in the 2FI model, including transferring balance between user accounts and updating user profiles in web applications.

In order to have low transaction completion time, Carousel leverages the pre-defined read and write keys in 2FI transactions to overlap transaction processing (i.e., reads and writes) with 2PC and replication. Figure 1 shows an example of Carousel's basic protocol (known as Carousel Basic). In this example, a transaction accesses two data partitions in different datacenters. We only show the leader of each partition's replica group for clarity. The client starts executing the transaction by sending (①) read-and-prepare requests to the two partition leaders. A read-and-prepare request includes both the read and write keys that the transaction accesses on the partition. Since the read and write keys are known, a partition leader uses optimistic concurrency control (OCC) to serve reads and isolate conflicting transactions. While returning (②) read results back to the client, each partition leader prepares the transaction. At this point, the transaction processing and 2PC start in parallel.

After receiving the read results, the client generates write data and sends (④) the data together with its commit request to the transaction coordinator. In Carousel, the coordinator is typically in the same datacenter as the client, and it is also the leader of a replica group. Upon receiving the commit request, the coordinator will replicate (⑤) the write data to its replicas for fault tolerance. Once the replication completes, the coordinator marks the transaction processing as completed. To commit the transaction, it must also know the 2PC prepare results from all participants.

2PC executes in parallel with the transaction processing. Each participant leader replicates (③) its prepare result to its followers. After the replication completes, a partition leader sends (⑥) its prepare result to the transaction coordinator. When the coordinator receives a prepared result from all participants and has completed its own processing, it will commit (⑦) the transaction. It will notify the client of its commit decision and asynchronously send a commit message with write data to every participant. Upon receiving the commit message, a partition leader will apply the updates after replicating the write data to its followers.

Carousel also has a fast protocol (known as Carousel Fast) to reduce transaction completion time by having a client send read-and-prepare requests directly to every replica of each participant partition. Although Carousel Fast has a lower average transaction completion time than Carousel Basic for most low-contention workloads, its tail latency for high-contention workloads is no better than Carousel Basic. This is because both protocols experience high abort rates for high-contention workloads, and retrying aborted transactions dominates their tail latencies. Without transaction prioritization support, Carousel is just as likely to abort a high-priority transaction as it is to abort a low-priority transaction. In contrast, Natto offers transaction prioritization to reduce the tail latency of high-priority transactions for high-contention workloads.

2.2 Estimating Request Arrival Time

Natto uses techniques from Domino [52] to estimate a transaction's arrival time at a Natto server. Domino has shown that network delays in private WANs are relatively stable, such as the one connecting Microsoft Azure data centers, and network delay information can be used to accurately estimate a request's arrival time at a server in a different datacenter.

Domino is a state machine replication (SMR) protocol that is designed for WANs. In order to account for both network delays and clock skews, each Domino client periodically probes the Domino servers, and each server returns its local time back to the client. The client calculates the one-way delay to a server as the difference between the sending time of its probe and the timestamp in the response from the server. To reduce the probability of underestimating its request's arrival time at the server, the client will estimate the arrival time by using the 95th percentile value from its collected delay information over a period of time.

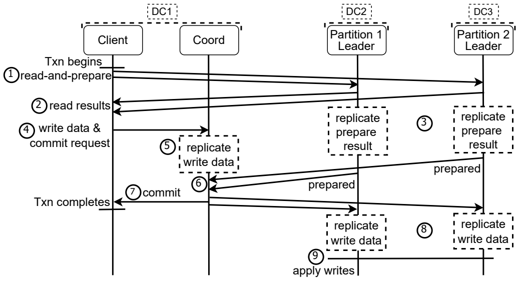


Figure 1: An example of Carousel's basic protocol.

Natto uses the same technique as Domino to estimate a transaction's arrival time at a participant server. While Domino targets SMR, Natto uses this estimation to support atomic commits and transaction prioritization. Unlike Domino that assigns a request with the estimated arrival time at a quorum of replicas, Natto's transaction timestamp indicates when the transaction should arrive at all of the servers that hold keys in the transaction's read and write sets.. Furthermore, Domino commits requests out of timestamp order, but Natto orders transactions across servers based on their timestamps and processes them in timestamp order to support transaction prioritization. Natto also differs from Domino in how it handles requests that arrive at a server later than the estimated arrival time. While Domino rejects these requests, Natto will only abort such a transaction if the transaction violates timestamp ordering with a conflicting transaction.

3 NATTO

Natto is a geo-distributed transaction processing system that provides transaction prioritization support for applications. It allows applications to assign a transaction with a high or low priority at runtime, and aims to lower the tail latency for high-priority transactions. Natto supports transaction prioritization in a distributed way by leveraging both network and transactional information. This section will first present Natto's system model and then describe Natto's transaction ordering and prioritization support.

3.1 System Model

We design Natto on top of Carousel's basic protocol [53]. We chose the basic protocol over the fast protocol because our focus is on reducing the tail latency for high contention workloads, and the fast protocol generally performs worse than the basic protocol in this scenario. Similar to Carousel, data in Natto are sharded into partitions, and data partitions can be distributed across geographically different datacenters. Each partition is further replicated at different datacenters for fault tolerance. For each partition, a replica is selected as the partition leader. Natto clients are application servers that also run in the same datacenters as Natto data servers. Following from Carousel, Natto targets the fail-stop failure model and support 2FI transactions.

In Natto, the leader of a participant partition in a transaction is called a participant leader, and other replicas of the partition are participant followers. Each transaction has a coordinator co-located with the client in a datacenter, and the coordinator's state is replicated to other datacenters to tolerate coordinator failures.

Natto transactions have two priority levels, low-priority and high-priority. This follows from previous work [37, 38] that argues that two priority levels are sufficient for many applications. For example, a web application can reserve high-priority transactions for high-value users to provide them with better performance [37]. In a trading platform, a large institutional trader may be willing to pay a premium to run its transactions at a high priority [38]. None of the techniques introduced in Natto is specific to having just two priority levels. Part of our future work is to extend Natto to support additional priority levels.

Natto clients issue transactions through a Natto client-side library that assigns each transaction a unique ID. The transaction ID can be derived from a client's unique ID and a monotonically increasing counter. A Natto client can specify the priority at the beginning of a transaction at runtime. This provides applications with flexible priority configurations for transactions.

The client-side library assigns each transaction a timestamp in order to provide support for transaction prioritization, which we will describe later in Section 3.2. As Natto leverages network measurements to assign transaction timestamps, it makes similar assumptions to Domino [52] about the network and deployment. Natto assumes that clients and servers are connected within a private wide-area network, which is common when they are deployed at a cloud provider. A private wide-area network can provide relatively stable network delays between datacenters, e.g., on Azure as shown by previous work [52]. Natto also assumes loosely synchronized clocks between clients and servers, which can be implemented by using a clock synchronization protocols, such as NTP.

3.2 Basic Timestamp-Based Prioritization

In geo-distributed database systems, a high-priority transaction may arrive at participant data servers in different orders relative to other transactions. By processing transactions in their arrival order, the servers would have different commit decisions for the high-priority transaction due to conflicts with other high-priority or low-priority transactions. This would cause the system to abort and retry the high-priority transaction, resulting in high latency. To address this issue, Natto introduces a timestamp-based ordering for transactions across data servers. This ordering enables Natto data servers to process transactions in the same order and handle high-priority and low-priority transactions with different concurrency control mechanisms to reduce the latency for high-priority transactions.

Observing that the critical path in 2PC is dominated by the furthest participant, there is no need for the other participants to process a received transaction immediately. Natto uses the same network measurement technique as Domino [52] to estimate a transaction's arrival time at participant servers (as described in Section 2.2). It then uses this time information to make participant servers process a transaction at the same time. Specifically, a Natto client will assign its transaction with a timestamp, indicating a future time when the transaction (i.e., its read-and-prepare request as described in Section 2.1) should have arrived at all participant leaders. The client will piggyback this transaction timestamp on its read-and-prepare requests to every participant leader.

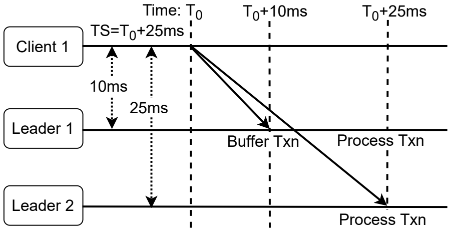


Figure 2: An example of Natto's transaction ordering.

Upon receiving the read-and-prepare request, a participant leader will not process the request until its clock time passes the transaction timestamp. Figure 2 shows an example where a transaction accesses two partitions. In this example, the client estimates that its request will arrive at participant leader L_1 and L_2 after 10 ms and 25 ms, respectively. It will set its transaction timestamp to be T_0 + 25 ms, where T_0 is its current time. After 10 ms, the transaction arrives at L_1 first, but L_1 will not process the transaction. When it is time T_0 + 25 ms, the transaction has arrived at L_2, and both L_1 and L_2 will independently process the transaction.

As there will be many concurrent transactions, a Natto server will buffer received transactions in a *transaction queue* in their timestamp order. If two transactions have the same timestamp, the server orders them based on the transaction ID. The server will process a transaction if its current time passes the transaction timestamp and the transaction is at the head of the queue. This timestamp-based ordering enables Natto servers to process conflicting transactions in the same order without a centralized scheduler.

To further support transaction prioritization, Natto uses different concurrency control mechanisms to prepare high-priority and low-priority transactions. When it is time to process a low-priority transaction, Natto simply follows Carousel's basic protocol to use OCC to serve reads and prepare the transaction. For high-priority transactions, a Natto server prepares them by using a locking-based mechanism to avoid unnecessarily aborting and retrying a high-priority transaction due to conflicts with concurrent transactions.

When it is time to process a high-priority transaction, Natto will first check whether the transaction can acquire locks on all its read and write keys (which are known at the start of the transaction for 2FI transactions). A lock on a key is available only if there is no prepared transaction (either high or low priority) that accesses the key. If the locks for the transaction's keys are all available, Natto will prepare the transaction. If any lock is unavailable, the transaction will not acquire any locks, and Natto will buffer the transaction until all its required locks are available. Buffered high-priority transactions are waiting for locks also in their timestamp order. In this locking-based mechanism, a high-priority transaction waits for conflicting transactions that have smaller timestamps to complete. When only a small number of conflicting transactions are queued up waiting for a lock, the waiting time is less than the time required to retry a transaction under high transaction contention, where a transaction may need to be retried multiple times before it is committed.

As long as data servers process conflicting high-priority transactions in the same timestamp order, there will be no deadlocks. To guarantee this timestamp order, a Natto server will check whether a high-priority transaction that arrives later than the expected time conflicts with any ongoing transaction (i.e., prepared or in the transaction queue) that has a smaller timestamp. If there are conflicts, instead of putting the high-priority transaction into the transaction queue, the server will abort the transaction to avoid deadlocks.

In Natto's basic timestamp-based prioritization approach, except for processing read-and-prepare requests, Natto executes transactions in the same way as Carousel's basic protocol (as described in 2.1). Natto provides the same guarantees offered by Carousel. Its timestamp-based ordering only provides advisory information for servers to help it decide when to process a transaction and operates on top of the transaction commit protocol. The ordering requires no communication between servers, and each server can independently prepare or abort transactions based on its local information.

Mispredicting a transaction's timestamp in Natto can reduce performance but does not affect the system's correctness. Overpredicting the timestamp would increase the transaction latency as even the furthest participant would wait to process a transaction after receiving it from the client. Underestimating the timestamp can cause a transaction to abort and retry, which can increase the overall completion time. Consistent underestimation of the timestamp can affect transaction liveness as it is possible for the same transaction to repeatedly abort. However, Natto leverages network measurements to dynamically adjust its timestamp estimation. This allows Natto's prediction accuracy to be robust under low and moderate network delay variance. Our experimental results in Section 5 show that Natto can have low tail latency for high-priority transactions even with significant network delay variance.

In Natto, clients are not end users but application servers running in datacenters. In most deployments, clients will be in the same administrative domain as Natto servers, and the system trusts applications to use the Natto client-library to determine transaction timestamps and to set transaction priorities following system-wide policies. However, Natto can be extended to be used in a shared environment where clients are not fully trusted. Instead of directly sending requests to Natto servers, clients must send transactions through a local trusted proxy server that runs a Natto client-side library to assign transaction timestamps. Clients can be given a quota of high-priority transactions based on their payment plan, and their high-priority transaction can be processed as a low-priority transaction if they go over their quota.

3.3 Priority-Based Prepare and Aborts

Using just basic timestamp-based prioritization, Natto servers process transactions in timestamp order. If a low-priority transaction has a smaller timestamp than a conflicting high-priority transaction, Natto will not process the high-priority transaction until the low-priority transaction completes, which introduces delays for the high-priority transaction.

However, Natto's timestamp-based ordering creates an *abort window* on participant servers other than the furthest one from the client to preemptively abort low-priority transactions and reduce the waiting time for high-priority transactions. While a Natto server queues up transactions in their timestamp order, it can abort queued low-priority transactions if the transactions would block a high-priority transaction. We call this optimization *priority abort*, which we will describe later in this section.

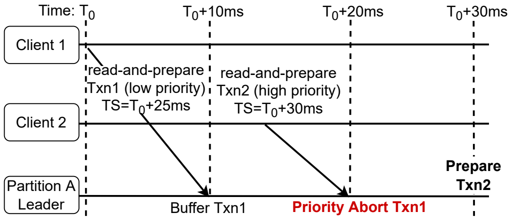


Figure 3: An example of priority abort.

Furthermore, if a server can estimate that a prepared low-priority transaction is going to abort before receiving the actual abort decision, it can optimistically prepare the subsequent conflicting high-priority transaction in order to reduce the waiting time for the high-priority transaction. We introduce *conditional prepare* to support such an optimization in Natto. In the rest of this section, we will describe priority abort and conditional prepare in detail.

3.3.1 Priority Abort.

As a transaction timestamp indicates when the transaction should have arrived at the furthest participant server, other participant servers may receive the transaction before the timestamp and buffer the transaction in the transaction queue until that timestamp. This buffering allows a Natto server to detect conflicts between low and high-priority transactions in the transaction queue before processing these transactions. If a low-priority transaction is ahead of a conflicting high-priority transaction in the queue, the server can abort the low-priority transaction immediately to avoid having the high-priority transaction wait for the completion of the low-priority transaction.

A Natto server can perform a priority abort for a low-priority transaction in its transaction queue any time before it prepares the transaction. When a high-priority transaction arrives at a server, the server puts the transaction in its transaction queue. It will abort any conflicting low-priority transactions in the queue. Similarly, a low-priority transaction can arrive at a server after a high-priority transaction but with a smaller timestamp. In this case, the server will also perform a priority abort for the low-priority transaction if there are conflicts between them.

Figure 3 shows an example of priority abort. In this example, a low-priority transaction Txn_1 and a high-priority transaction Txn_2 have conflicts on participant A. They also access other data partitions without conflicts, where we have omitted these partitions in the figure for clarity. When the current time is T_0, $client_1$ sends Txn_1 to every participant, estimating that the transaction will arrive at all participants after 25 ms. The transaction first arrives at participant A after 10 ms. Participant A will buffer the transaction until the time reaches the transaction's timestamp. However, before that happens, at time $T_0 + 20ms$, participant A receives Txn_2. Since Txn_2 has a larger timestamp than Txn_1, participant A will perform a priority abort for Txn_1 in order to prepare Txn_2 on time. This example shows that priority abort allows a high-priority transaction to commit even through there are conflicting low-priority transactions that have smaller timestamps.

In Natto, it is possible that a high-priority transaction has a large timestamp because it needs to access a distant remote data server. When the transaction arrives at a nearby server, the difference between its timestamp and a previous conflicting low-priority transaction in the queue is larger than the expected completion time of the low-priority transaction. In this case, it is not necessary to abort the low-priority transaction because the low-priority transaction should complete before the execution time of the high-priority transaction. This will reduce the abort rate for low-priority transactions. To achieve this, a Natto server needs to estimate a transaction's completion time. A server can leverage network measurements to estimate network delays to other participant servers and use the estimation to predict a transaction's completion time.

Priority abort leverages the timestamp-based ordering to abort low-priority transactions. It maintains the invariant that a server will not have a low-priority transaction ahead of a conflicting high-priority transaction in its transaction queue. Although priority abort reduces latency for high-priority transactions, it may lead to starvation in some workloads, where a low-priority transaction keeps being aborted. A number of existing approaches can be used to address the starvation. For example, a low-priority transaction can be promoted to high priority if it is aborted one or more times.

3.3.2 *Conditional Prepare.*

When a server performs a priority abort for a low-priority transaction due to the arrival of a conflicting high-priority transaction, it is possible that another participant server is preparing the low-priority transaction because it has yet to receive the high-priority transaction. The high-priority transaction may also have conflicts with the low-priority transaction at this participant server. However, this server cannot perform a priority abort for the low-priority transaction. It needs to wait for the abort notification from the coordinator. The abort notification may not arrive in time to process the high-priority transaction at its transaction timestamp, which can introduce delays. To reduce this delay, Natto introduces conditional prepare that allows the server to optimistically prepare the high-priority transaction even if the low-priority transaction is still in a prepared state. As the server does not know the low-priority transaction's state on other participant servers, preparing the high-priority transaction will succeed only if the low-priority transaction is finally aborted.

To perform conditional prepare for a high-priority transaction, a Natto server will estimate whether the transaction causes priority abort for conflicting low-priority transactions at the other participant servers. To support accurate predictions, each client piggybacks its estimated arrival time of a transaction at every participant server as well as all read/write keys on its read-and-prepare requests (as described in Section 2) to all participants. If a server estimates that priority abort happens on another participant server, it will conditionally prepare the high-priority transaction, where the condition is that the conflicting transaction will be priority aborted on another participant server. It will replicate its conditional prepare result and then send the result together with the condition to its transaction coordinator. The transaction coordinator will determine whether the conditional prepare is successful after receiving the prepare results from all participant servers.

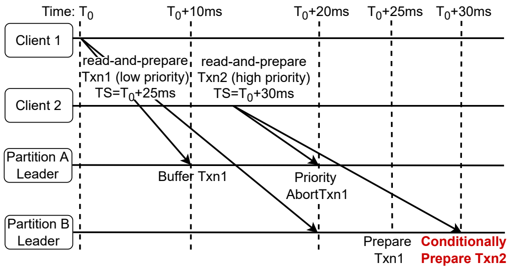


Figure 4: An example of conditional prepare.

Figure 4 shows an example of when a Natto server performs conditional prepare. This example is similar to our previous example on priority abort except that the high-priority transaction, Txn_2, has conflicts with the low-priority transaction, Txn_1, on both participant A and B. In this example, when participant B receives Txn_1, it has not seen Txn_2. It prepares Txn_1 when the time passes the transaction timestamp. At a later time, it receives Txn_2, and it uses the embedded timestamp and transaction information to estimate that participant A has performed a priority abort for Txn_1. It will then conditionally prepare Txn_2 without waiting for the abort acknowledgement on Txn_1. This will reduce the latency for the high-priority transaction.

When a server conditionally prepares a high-priority transaction, it also sends the read results to the client. The client executes the transaction based on the read results and sends the commit request to the transaction coordinator. However, the coordinator will not commit the transaction until it learns that the conditional prepare is successful. The conditional prepare could fail if the conflicting low-priority transaction is committed. In this case, Natto discards the conditional prepare result and processes the high-priority transaction using the normal path. The client also needs to re-execute the transaction based on the normal-path read results. To support this, Natto tracks whether reads are from conditional prepare, and the client needs to handle failed conditional prepare in its transaction execution. To reduce latency under the failure of conditional prepare, Natto executes the normal path and conditional path in parallel. If the conditional path successfully commits a high-priority transaction, Natto ignores the results from the normal path. Otherwise, Natto executes the transaction normally.

Conditional prepare introduces an optimistic path to prepare a high-priority transaction earlier. The system maintains the invariant that it cannot commit the high-priority transaction based on the conditional prepare result if the condition is not satisfied.

3.4 Early Committed State Forwarding

Priority abort and conditional prepare can reduce the latency of a high-priority transaction by preempting the conflicting low-priority transactions. However, when a high-priority transaction conflicts with transactions that cannot be preempted, like other high-priority transactions, the high-priority transaction needs to wait until the conflicting transactions complete. In Carousel, a transaction's updates will not be visible until after a participant leader replicates the updates to its followers. If we use this approach in Natto, then a high-priority transaction can block on a committed conflicting

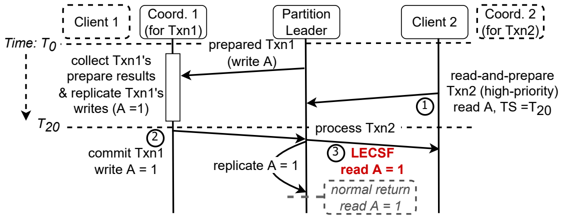


Figure 5: An example of LECSF.

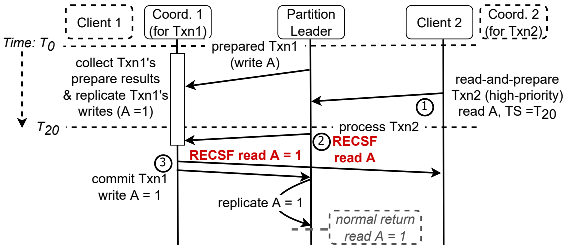


Figure 6: An example of RECSF.

transaction until the conflicting transaction's updates are applied by participant leaders.

To reduce this waiting time for high-priority transactions, we introduce *early committed state forwarding (ECSF)* in Natto to allow a committed transaction's updates to be visible to the subsequent conflicting transactions (in Natto's timestamp order) as early as possible. In ECSF, once a transaction's commit is fault-tolerant, its updates can be visible to the subsequent conflicting transaction before being applied by servers. If the subsequent conflicting transaction commits, Natto servers will apply the two transactions' updates in their timestamp order. We observe that, in Carousel, a transaction's commit and write data are first fault-tolerant at the transaction coordinator [53], and then propagated to participant leaders (as described in Section 2.1). Before participant leaders replicate the write data, Natto can perform ECSF at the participant leaders or the coordinator, which we call local ECSF or remote ECSF, respecitvely.

Local ECSF (LECSF). In LECSF, once a participant leader receives a committed transaction's update data from the transaction coordinator, it will make the data visible to the next conflicting transaction before replicating the data to its replication group. Figure 5 shows an example of LECSF. In this example, at time T_0, the participant leader has prepared Txn_1 that writes key A. At time T_{20}, it processes high-priority transaction Txn_2 but cannot serve its read on key A. Once it receives the commit result from $Txn_1's$ coordinator, it performs LECSF to serve $Txn_2's$ read. This reduces latency compared to serving the read after having the leader replicate and apply $Txn_1's$ writes. Natto uses LECSF for both high and low priority transactions, where a transaction can read the update data of a previous committed conflicting transaction before the updates are applied to the data store. This reduces the latency for both high-priority and low-priority transactions.

Remote ECSF (RECSF). A committed transaction's updates first become reliable on the transaction coordinator before they are forwarded to participant leaders. A participant leader has to wait for the updates to process any subsequent conflicting high-priority transactions in its transaction queue. RECSF aims to reduce this waiting time for high-priority transactions.

As a participant processes transactions in their timestamp order, it knows the immediate previous conflicting transaction for a queued high-priority transaction. The participant can forward the read requests of the high-priority transaction to the coordinator of the conflicting transaction. Figure 6 shows an example of RECSF. This example is similar to the previous LECSF example (in Figure 5). However, when the participant leader processes Txn_2, it forwards the read request to $Txn_1's$ coordinator instead of waiting for $Txn_1's$ write data. Once the coordinator commits Txn_1, it will serve $Txn_2's$ read. In RECSF, a high-priority transaction's read request is served by a different transaction's coordinator, and the coordinator may be in a different datacenter from the client of the high-priority transaction. Depending on the network delay between the coordinator and the client, the client may not receive the read results earlier compared to just using LECSF. To address this, Natto can run both RECSF and LECSF in parallel, and the client will use its first received read results. Due to the overhead of forwarding read results in RECSF, Natto only applies RECSF to high-priority transactions.

Both LECSF and RECSF make a transaction read the updated data of a previously committed conflicting transaction (in Natto's timestamp order) earlier than Natto's transaction commit protocol without changing Natto's transaction isolation for conflicting transactions. This is because the data that the transaction reads is deterministic, and is the same as the data the transaction would have read after the servers apply the previous conflicting transaction's updates. Also, with both LECSF and RECSF, participant servers apply conflicting transactions' updates in the order that the transactions are prepared, which guarantees the same execution order as Natto's commit protocol.

4 IMPLEMENTATION

As Natto is based on Carousel's basic protocol, we extend an implementation of Carousel to build a prototype of Natto. Our Natto prototype has all of the optimizations that we have introduced in this paper, including Priority Abort, Conditional Prepare, Local ECSF, and Remote ECSF. Our implementation consists of approximate 4k lines of code in the Go language. It uses gRPC [27] for network I/Os and extends an open-source implementation [22] of Raft [41] to manage replicas.

To reduce the overhead of network measurements, our Natto prototype uses one *proxy* per datacenter to periodically probe the replica leader of every data partition. A proxy uses its network measurement data to estimate network delays to servers on behalf of all clients in its datacenter. A client will fetch the delay information from the local proxy to determine a transaction timestamp. To reduce the load on the proxy, clients will buffer the delay information and periodically contact the proxy for updates.

Our prototype of Carousel includes both Carousel Basic and Carousel Fast. In addition to Carousel, we also implement a prototype of TAPIR [54] and a Spanner [17]-like system that uses 2PL+2PC to compare with Natto in our evaluation. For a fair comparison, our implementations of TAPIR and the 2PL+2PC system are also written in GO and uses gRPC for network I/O. Like Carousel,

	WA	PR	NSW	SG
VA	67	80	196	214
WA	-	136	175	163
PR	-	-	234	149
NSW	-	-	-	87

Table 1: Network roundtrip delays (ms)

TAPIR also has a fast path and a slow path to commit transactions. Compared with the open-source implementation [51] of TAPIR, our TAPIR prototype starts a slow path as soon as the fast path fails, instead of waiting for a 500 ms timeout. This reduces TAPIR's transaction completion time in our evaluation settings, where it typically takes less than 500 ms to know the failure of the fast path.

Our implementation of the 2PL+PC system uses wound-wait to prevent deadlocks. We further implement priority preemption in the 2PL+2PC system to support transaction prioritization, where a high-priority transaction will preempt conflicting low-priority transactions. To prevent deadlocks, the system will also preempt all low-priority transactions that have a smaller timestamp and wait for the lock. This is similar to placing high-priority transactions in a separate queue and always processing transactions from that queue first. Furthermore, we also implement preempt-on-wait (POW) [38] in the 2PL+2PC system as another mechanism to support transaction prioritization. Our prototypes of Carousel, TAPIR, and the 2PL+2PC systems do not implement fault recovery.

5 EVALUATION

Our evaluation compares Natto with Carousel [53], TAPIR [54] and a Spanner [17]-like (2PL+2PC) system. It includes experiments both in a local cluster (with emulated wide-area network delays) and on Microsoft Azure datacenters. In these experiments, we use three different workloads: YCSB+T [19] (a transactional extension on the YCSB key-value workload [16]), Retwis [34, 54] (a synthetic Twitter-like workload), and SmallBank [13, 20] (a workload that models banking applications).

5.1 Experimental Settings

In our experiments, data is sharded into 5 partitions, and each partition has 3 replicas, resulting in a deployment that spans 15 data servers. We run one server on each machine, and we select servers evenly across 5 Microsoft Azure datacenters: Virginia (VA), Washington (WA), Paris (PR), New South Wales (NSW) and Singapore (SG). Table 1 shows the average network roundtrip delays between these datacenters based on the network measurement data in [52]. Our deployment has one partition leader at each datacenter, and a datacenter has at most one replica for a partition.

We deploy two client machines at each datacenter. For each experiment, all clients generate transactions at the same rate, and we use the *transaction input rate* metric to represent the total number of new transactions (from all clients) that are submitted per time unit in a system. When the system aborts a transaction, the client will immediately retry the transaction. Retried transactions are not counted in the transaction input rate. If a transaction cannot commit after 100 retries, we consider the transaction to have failed, and its latency is not included in the experimental results. The latency of a committed transaction includes retries. By default, a transaction has a 10% (or 90%) probability of being assigned a high-priority (or low-priority), which is the same as the settings used in [37]. Unless specified otherwise, our data set consists of 1 million key-value pairs, where a key and a value are both 64 bytes in size. Transactions access keys by following a Zipfian distribution with a default coefficient of 0.65. We run each experiment for 60 seconds and exclude the transactions from first and last 10 seconds. We repeat each experiment 10 times and show the 95% confidence interval for data points using error bars.

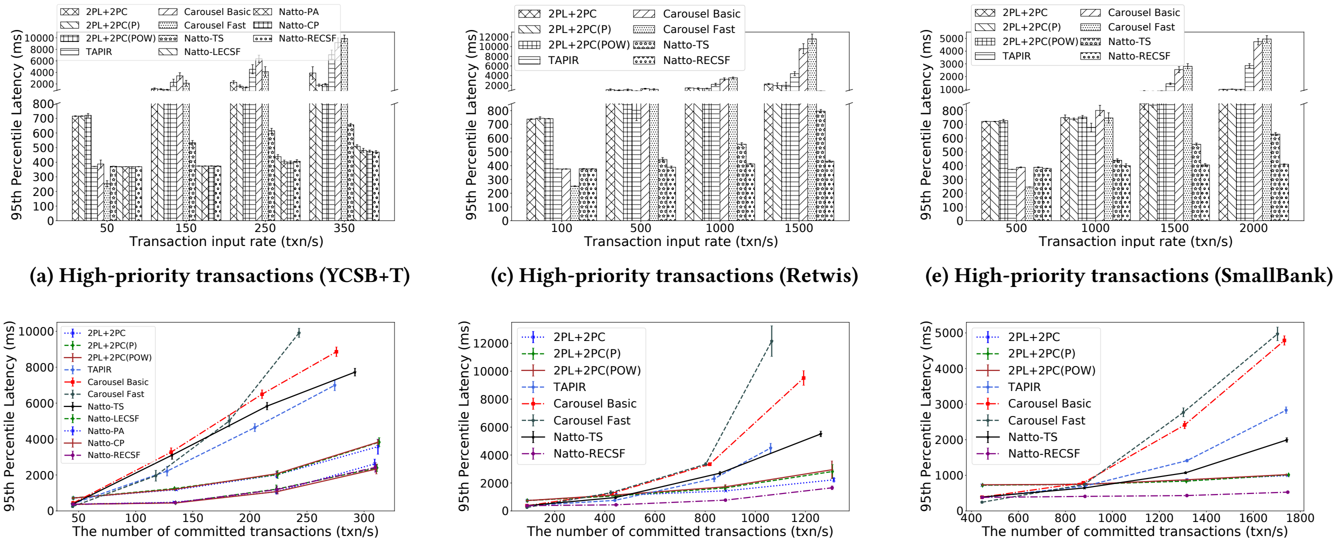
In our evaluation, a Natto proxy (as described in Section 4) probes partition leaders every 10 ms and uses the measurement data from the last second to estimate delays to the leaders. A client will contact the local proxy for the delay information every 100 ms. We evaluate Natto by using different combinations of its transaction prioritization mechanisms as follows: Natto-TS for Natto's basic timestamp-based transaction prioritization support (TS); Natto-LECSF for TS and Local ECSF (LECSF); Natto-PA for TS, LECSF, and Priority Abort (PA); Natto-CP for TS, LECSF, PA, and Conditional Prepare (CP); Natto-RECSF for TS, LECSF, PA, CP, and Remote ECSF (RECSF). We use Carousel Basic and Carousel Fast to represent Carousel's basic protocol and fast protocol, respectively. We use 2PL+2PC to represent the 2PL+2PC system, and 2PL+2PC(P) and 2PL+2PC(POW) to represent the system running with transaction preemption and POW [38], respectively.

In our local cluster, each machine has 12 CPU cores and 64 GB of memory. The machines are connected through a 1 Gbps network. We use the Linux traffic control utility to emulate wide-area network (WAN) delays between machines that belong to the different datacenters shown in Table 1. We also deploy the systems on Microsoft Azure by using the same five datacenters. Our deployment uses the Standard_D8s_v3 VM instance, which has 8 virtual CPU cores and 32 GB of memory.

5.2 Impact of Transaction Input Rate

We first evaluate the performance of the systems with different transaction input rates using three workloads: YCSB+T on our local cluster, and Retwis and SmallBank on Microsoft Azure.

5.2.1 YCSB+T Workload. In this experiment, each transaction consists of 6 read-modify-write operations accessing different keys. Figure 7 (a) shows the 95th percentile (95P) latency of high-priority transactions. When the input rate is low, i.e., at 50 transactions per second (txn/s), all versions of Natto have a similar latency to Carousel Basic. This is because few transactions have conflicts, and Natto processes transactions following their timestamp order without triggering its transaction prioritization mechanisms. Furthermore, although Natto waits until a transaction's timestamp to process the transaction, this waiting time has limited overhead compared to Carousel Basic in this experiment. This shows that Natto's transaction timestamp assignment is effective when network delays are relatively stable. In addition, Carousel Fast has the lowest latency among all of the systems since its fast path can commit a transaction in one WAN roundtrip while Carousel Basic needs two WAN roundtrips. TAPIR also has a fast path to commit transactions in one network roundtrip. However, it needs to read



(a) High-priority transactions (YCSB+T)

(c) High-priority transactions (Retwis)

(e) High-priority transactions (SmallBank)

(b) Low-priority transactions (YCSB+T)

(d) Low-priority transactions (Retwis)

(f) Low-priority transactions (SmallBank)

Figure 7: Impact of increasing transaction input rate.

data from remote datacenters first before committing a transaction, resulting in higher latency than Carousel Fast. Compared with Carousel and TAPIR, the three 2PL+2PC based systems sequentially execute reads, 2PC, and replication, resulting in significantly higher latency (approximate 715 ms).

Furthermore, Figure 7 (a) also shows that, as the transaction input rate increases from 50 to 350 txn/s, the 95P latency of high-priority transactions significantly increases for Carousel, TAPIR, and the three 2PL+2PC systems. In contrast, the latency increase for Natto is much smaller. For example, when the input rate is at 350 txn/s, both Carousel and TAPIR have over 5000 ms 95P latency compared to only 656 ms in Natto-TS. This is because contention between transactions increases with the input rate, and both Carousel and TAPIR abort and retry high-priority transactions, resulting in a significant increase in tail latency. Compared to Carousel and TAPIR, 2PL+2PC has lower latency since it decreases the number of aborts and retries for high-priority transactions. Although 2PL+2PC(P) and 2PL+2PC(POW) can further reduce the latency for high-priority transactions by preempting conflicting low-priority transactions, they still have much higher latency than Natto-TS. Furthermore, Natto-LECSF has 147 ms less latency than Natto-TS because a transaction can read the write data from a conflicting transaction earlier. Natto-PA, Natto-CA, and Natto-RECSF have similar latencies that are approximately 40 ms less than Natto-LECSF since there is only moderate transaction contention in this experiment.

Figure 7 (b) illustrates the 95P latency of low-priority transactions for the same experiment. To show how many low-priority transactions a system commits in this experiment, we change the x-axis from the input rate to the corresponding number of committed (low-priority) transactions per second. Even though it is prioritizing high-priority transactions, Natto still has a similar or even lower latency for low-priority transactions than Carousel and TAPIR under the same goodput. Natto-LECSF, Natto-PA, Natto-CP, and Natto-RECSF perform similarly for low-priority transactions because Natto's PA, CP, and RECSF optimizations are only effective for high-priority transactions. They have a lower latency than Natto-TS for low-priority transactions since the LECSF optimization allows transactions to read the write data from a committed but not replicated transaction, which reduces the latency of both low and high-priority transactions.

Natto-TS has lower latency for low-priority transactions than Carousel Basic as the goodput increases. This is because Carousel Basic will, in some cases, abort transactions that arrive at participants leaders in a different order relative to other transactions. When the input rate is low, transaction contention is low as well, and Carousel Fast and TAPIR have lower latencies than Natto-TS because of their fast path. However, as transaction contention increases with the input rate, Carousel Fast experiences higher latency than Natto-TS and Carousel Basic due to a higher abort rate from reading stale data from local replicas. In addition, 2PL+2PC, 2PL+2PC(P), 2PL+2PC(POW) and TAPIR's latencies remain lower than Natto-TS and Carousel Basic for all of the tested goodputs, but are significantly higher than the other variations of Natto.

Both Figure 7 (a) and Figure 7 (b) show that Natto provides effective transaction prioritization support for geo-distributed transactions. Compared to the other systems, Natto can significantly reduce the tail latency for high-priority transactions while having similar or even lower tail latency for low-priority transactions.

5.2.2 *Retwis workload.*

We use the same Retwis workload as in [54], which is a synthetic Twitter-like workload. Compared with YCSB+T, Retwis transactions read and write a different number of keys, representing workloads in more complicated applications. The transaction profile of Retwis includes 5% adding users (reads 1 key and writes 3 keys), 15% following users (reads and writes 2 keys), 30% posting tweets (reads 3 keys and writes 5 keys), and 50% loading timelines (reads a random number of keys between 1 and 10).

Figure 7 (c) shows the 95P latency of high-priority transactions in different systems as the transaction input rate increases. Like our YCSB+T experiments, Natto-RECSF and Natto-TS have significantly lower latency than TAPIR, Carousel, and 2PL+2PC systems when the input rate is high. For example, when the input rate is 1500 txn/s, Natto-RECSF has only 432 ms latency while 2PL+2PC (P) and TAPIR have 1922 ms and 4393 ms latency respectively. At this point, Carousel Basic and Carousel Fast have higher latency than TAPIR. This is because Carousel uses partition leaders to perform coordination operations (like replication and 2PC) for a transaction, and the leaders become a performance bottleneck due to the many retried transactions. TAPIR replicas are not saturated at this point since it offloads transaction coordination work to clients. The three 2PL+2PC systems have lower latency than TAPIR and Carousel because they have fewer aborts and retries with increasing transaction contention. However, with more effective transaction priority mechanisms, Natto has significantly lower latency compared with 2PL+2PC(P) and 2PL+2PC(POW). We also show the 95P latency of low-priority transactions and their good throughput in Figure 7 (d). Natto-RECSF has the lowest latency when the goodput is above 800 txn/s, showing that Natto-RECSF introduces little overhead to low-priority transactions in practical deployments.

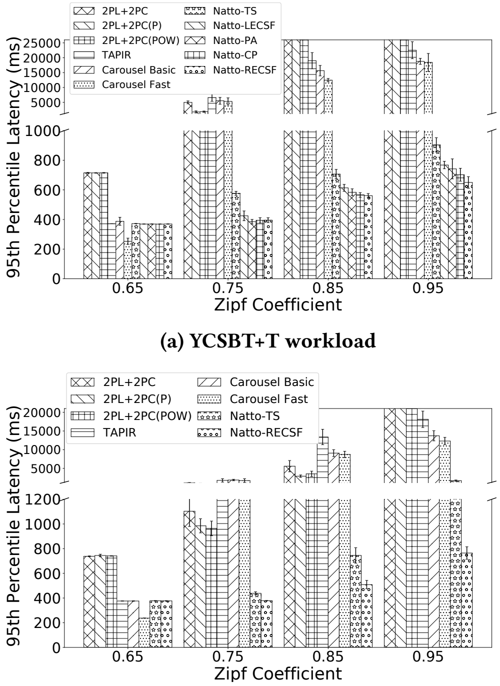
5.2.3 SmallBank workload. In addition to Retwis, we also compare the different systems on Microsoft Azure by using the SmallBank workload from OLTP-Bench [20]. SmallBank represents banking-like applications, where transactions only read and update one or two users' accounts. The OLTP-Bench SmallBank implementation extends the original SmallBank implementation [13] with money transfer transactions between user accounts. We configure the SmallBank workload with 1 million users. 1K users' accounts are hot data, and 90% of transactions will access the hot data.

Figure 7 shows the 95P latency of high-priority and low-priority transactions in different systems as the transaction input rate increases from 500 txn/s to 2000 txn/s. Similar to the results in our Retwis experiments, Figure 7 (e) shows that Natto-TS and Natto-RECSF have significantly lower latency for high-priority transactions than TAPIR and Carousel when the transaction input rate is high (e.g., at 1500 txn/s). Figure 7 (f) also shows that, under the same goodput, Natto-TS and Natto-RECSF have similar or even lower latency for low-priority transactions compared to Carousel and TAPIR. These experimental results demonstrate that Natto's transaction prioritization support is also effective for transactions that only access a small number of keys.

Both Retwis and SmallBank experiments show that Natto effectively reduces the latency for high-priority transactions for different types of workloads on Microsoft Azure. As a result, Natto's network-measurement-based techniques are a practical approach to provide transaction prioritization support for geo-distributed transactions.

5.3 Performance under High Contention

To further compare the different systems, we evaluate the performance of the systems under high transaction contention. We run the experiments on our local cluster with the YCSB+T workload to show the performance improvements of Natto's transaction prioritization mechanisms. We also perform experiments on Microsoft Azure using the Retwis workload.



(a) YCSBT+T workload

(b) Retwis workload

Figure 8: Impact of increasing the Zipfian coefficient.

5.3.1 YCSB+T Workload. Figure 8 (a) shows the 95P latency for high-priority transactions with various Zipfian coefficient values at 50 txn/s input rate. Under this relatively low input rate, as transaction contention increases by increasing the Zipfian coefficient from 0.65 to 0.95, Carousel and TAPIR experience an order of magnitude increase in latency (from under 400 ms to over 5000 ms) since they have to retry a transaction multiple times due to conflicts with concurrent transactions. The three 2PL+2PC systems have higher latency increase (from 700 ms to over 25000 ms) because of queuing due to contention. Natto-TS only has approximately 2.5 times higher latency (from 372 ms to 903 ms). This is because Natto-TS delays processing a high-priority transaction until a conflicting transaction with a smaller timestamp completes.

The figure also shows that when transaction contention is high, e.g., at 0.95 Zipfian coefficient, Natto-LECSF has approximately 136 ms lower latency than Natto-TS due to reading the write data of a conflicting transaction earlier. At the same point in the figure, Natto-PA has lower latency than Natto-LECSF. This is because Natto-PA can process a high-priority transaction earlier by aborting a conflicting low-priority transaction that has a smaller timestamp. In this scenario, Natto-CP can optimistically prepare the high-priority transaction across data partitions so that, as shown in the figure, Natto-CP has lower latency than Natto-PA. Natto-RECSF has the lowest latency out of the different systems because it allows a high-priority transaction to read the write data from a conflicting transaction as early as possible. This experiment shows that Natto's LECSF, PA, CP, and RECSF mechanisms effectively reduce the latency for high-priority transactions, especially when transaction contention is high.

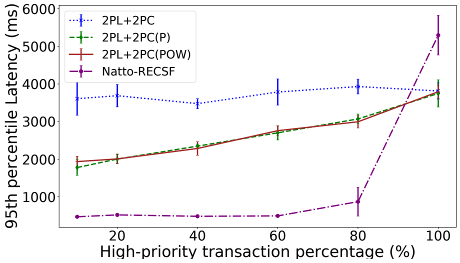


Figure 9: Impact of high-priority transaction percentage.

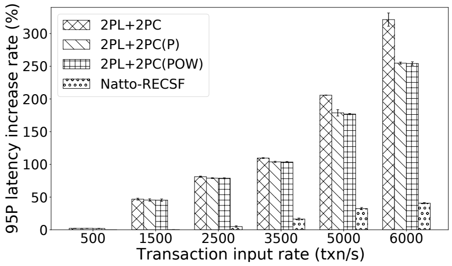


Figure 10: Modified SmallBank workload where sendPayment transactions are given a high priority.

5.3.2 *Retwis workload.*

Figure 8 (b) shows the 95P latency of high-priority transactions in the different systems with moderate to high transaction contention when the transaction input rate is 100 txn/s. When transaction contention is moderate, i.e., at 0.75 Zipfian coefficient, Natto-TS (438 ms) and Natto-RECSF (378 ms) have over 3 times lower latency than TAPIR (1794 ms), Carousel Basic (1892 ms), Carousel Fast (1745 ms), 2PL+2PC (1103 ms), 2PL+2PC(P) (984 ms), 2PL+2PC(POW) (953 ms). Although Natto-RECSF's latency increases with transaction contention, at 0.95 coefficient, it has 10 times lower latency than TAPIR, Carousel, and 2PL+2PC.

5.4 Impact of High-Priority Transaction Load

We also evaluate the performance of Natto by varying the percentage of high-priority transactions in the YCSB+T workload. With the transaction input rate at 350 txn/s, Figure 9 shows the 95P latency of high-priority transactions in different systems under various percentage of high-priority transactions. As Carousel, TAPIR, and 2PL+2PC process both low and high priority transactions identically, and 2PL+2PC has lower latency based on the result shown in Figure 7 (a), we only show 2PL+2PC, 2PL+2PC(P), and 2PL+2PC(POW). To further improve the readability of the graph, instead of showing every variation of Natto, we only show Natto-RECSF combining all transaction prioritization mechanisms in Figure 9.

As shown in the figure, when the high-priority transaction percentage increases from 10% to 100%, 2PL+2PC has a stable latency that slightly fluctuates at around 3700 ms. 2PL+2PC(P) and 2PL+2PC(POW) have latencies that increase from 1778 ms and 1951 ms to 3748 ms and 3762 ms, respectively, as fewer low-priority transactions can be preempted. Compared to these systems, Natto effectively reduces the latency of high-priority transactions to about 600 ms when the high-priority transaction percentage is lower than 60%, However, when the workload only contains high-priority transactions, Natto performs worse than 2PL+2PC. Natto is not designed for high-priority-transaction-only workloads. We believe the vast majority of applications that require priority transaction processing have a high priority transaction percentage that is significantly lower than 80%.

Furthermore, we compare the four different systems with the SmallBank workload by having only the *sendPayment* transaction as high-priority. This represents workloads that have multiple transaction types with one set as high-priority. Figure 10 shows the increase ratio of the 95P latency of high-priority transactions under various input rate compared to the latency at 100 txn/s. Note that Natto has a lower latency than the other systems when the input rate is 100 txn/s. The three 2PL+2PC systems have over 200% increase on the 95P latency for high-priority transactions when the input rate is 6000 txn/s. In contrast, Natto-RECSF has less than 50% increase. Our experiments also found that the latency growth of Natto's high priority transaction is significantly slower than that of its low priority transactions as more load is introduced.

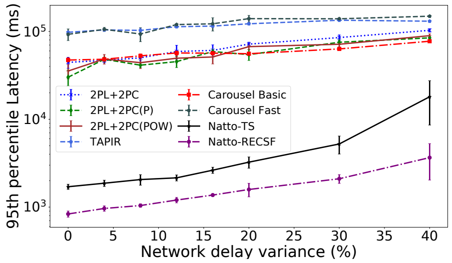


Figure 11: Impact of network delay variance.

5.5 Impact of Network Delays

As Natto leverages network measurement to assign transaction timestamps, we further study how Natto responds to network delay variance. For network delays between two datacenters, we represent network delay variance by using the ratio of the standard deviation and the average network delay. By using the network measurement data in [52], we find that the network delay variance between the five Azure datacenters in our previous experiments is at most 0.1%. Since network delays between datacenters on Azure are relatively stable, we run experiments in our local cluster by emulating network delay variance between the five datacenters. In our emulation, network delays between datacenters follow a Pareto distribution with the same average network delays as in Table 1.

In this experiment, we use our YCSB+T workload with the transaction input rate at 350 txn/s. Figure 11 shows the 95P latency of high-priority transactions in different systems under various network delay variances. When the network delay variance is low and moderate (e.g., up to 15%), Natto-TS and Natto-RECSF have significantly lower latency than Carousel, TAPIR, and the three 2PL+2PC systems. The latency of Natto-TS and Natto-RECSF increase with the network delay variance. This is because a transaction is more likely to arrive at a server later than its timestamp under large network delay variance, and Natto may abort and retry the transaction

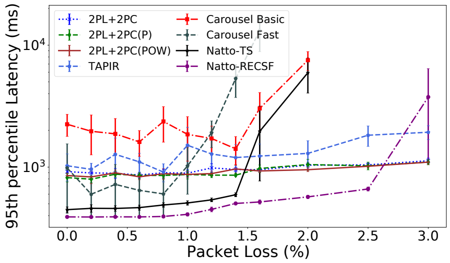


Figure 12: Impact of network packet loss.

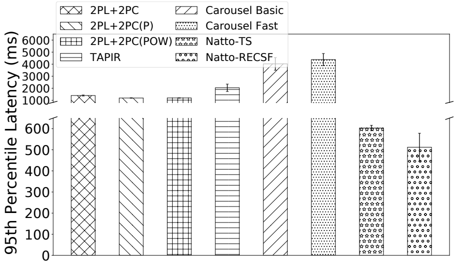


Figure 13: Latency under hybrid cloud (AWS and Azure).

under transaction contention. However, even when network delay variance is large, i.e., at 40% in the figure, Natto-TS and Natto-RECSF still have lower latency than the other systems that are under no emulated network delay variance. This shows that Natto's transaction prioritization support can be effective even under moderate to large network delay variance.

Furthermore, we evaluate the impact of network packet loss on Natto using the YCSB+T workload with transaction input rate at 100 txn/s. Figure 12 shows the 95P latency of high-priority transactions in different systems under various packet loss rates. When the packet loss is at 1.5%, Carousel Basic experiences significant latency increase because the TCP throughput becomes the bottleneck. The TCP throughput drops significantly as the packet loss rate increases. Carousel Basic is saturated earlier than TAPIR and the 2PL+2PC systems since it requires more network bandwidth as it must replicate transactional data twice. As Natto-TS is built on top of Carousel Basic, it has similar network bandwidth usage and is also saturated with 1.5% packet loss. Carousel Fast uses more network bandwidth than Carousel Basic and experiences significant latency increase earlier at 1% packet loss. The TCP throughput becomes a bottleneck for Natto-RECSF when the packet loss is above 2.5%. This is because Natto-RECSF completes a high-priority transaction earlier before participant leaders complete data replication, which mitigates the impact of packet loss on client perceived latency. While a high packet loss rate could cause high latency in Natto, the packet loss rate of most networks is typically low to moderate (e.g., less than 1%) in practice. At more typical packet loss rates, Natto has lower latency than the other systems.

Although Natto targets deployments in private WANs, we further evaluate the performance of Natto under a hybrid cloud network environment. In this experiment, we deploy the different systems on both Azure and AWS by replacing two Azure datacenters (i.e., VA and WA) in our default setting with the us-east and us-west datacenters on AWS. Our deployment on AWS uses the c4.2xlarge machine, which has 8 vCPUs and 15 GiB memory. Figure 13 shows the 95P latency of high-priority transactions in the different systems using the Retwis workload with 1000 txn/s input rate. In this hybrid cloud network setting, both Natto-TS and Natto-RECSF have significantly lower latency than the other systems. As a result, Natto's transaction prioritization mechanisms can be effective in a more general network environment.

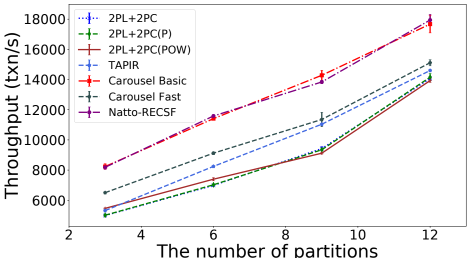


Figure 14: Throughput with different numbers of partitions.

5.6 Throughput

We further study the peak throughput of different systems running the experiments on our local cluster using the Retwis workload with an uniform key distribution. We simulate three datacenters, and the simulated roundtrip delays between datacenters are 4 ms, 6 ms, and 8 ms. Due to a lack of machine resources, we deploy one partition leader and two partitions followers on each machine and limited our experiments to 12 partitions.

Figure 14 illustrates that t he peak throughput of Carousel, TAPIR, 2PL+2PC, and Natto scale linearly with the number of partitions. Carousel Basic and Natto have similar peak throughput ranging from 8000 txn/s to 17500 txn/s. With 12 partitions, Carousel Fast has slightly higher throughput (15112 txn/s) than TAPIR (14598 txn/s), 2PL+2PC (14069 txn/s), and 2PL+2PC(P) (14156 txn/s).

6 RELATED WORK

This section will first summarize previous work on transaction processing for geo-distributed data and then briefly review past work on supporting transaction prioritization.

6.1 Geo-Distributed Transaction Processing

Geo-distributed storage systems shard data into partitions to achieve scalability and replicate data partitions to provide fault tolerance. To process distributed transactions that access multiple partitions, Megastore [10], Spanner [17], and CockroachDB [15], sequentially perform transaction processing (i.e., reads and writes), 2PC, and the replication of transactional data and states. This requires multiple wide-area network roundtrips to complete and result in high latency. To reduce the latency, MDCC [33] and TAPIR [54] introduce a fast path to execute 2PC and replication in parallel. By requiring read and write keys to be specified in the beginning of a transaction, Carousel [53] further overlaps the execution of transaction processing with 2PC and replication.

However, these systems have limited support for transaction prioritization. For example, Google's Cloud Spanner has recently added support for transaction priorities, although its priorities mainly affect resource scheduling [28]. CockroachDB [15] only considers transaction priority when handling deadlocks. In these systems, a high-priority transaction can be aborted and retried due to conflicts with low-priority transactions, significantly increasing the transaction's total completion time. For comparison, Natto's transaction prioritization support focuses on transaction contention, and aims to reduce latency for high-priority transactions.

Similar to Carousel and Natto, many systems limit transaction expressiveness in order to reduce latency and/or abort rate. For example, Sinfonia [7] also parallelizes transaction processing and commit but requires pre-defining both write keys and values in its mini-transaction model. Calvin [49], CalvinFS [48], and Q-Store [43] target deterministic transactions that also require pre-defined read and write sets, and they can avoid aborting transactions by deterministically ordering the transactions. By targeting one-shot transactions [32], Granola [18], Rococo [39], Janus [40], and Ocean Vista [24] can also avoid aborting transactions. RAMP [9] and ALOHA-KV [25] do not abort transactions either but only support read-only and write-only transactions. Although ALOHA-DB [23] extends ALOHA-KV to support read-write transactions, it needs to convert a transaction to a set of functors. Lynx [55] chops a transaction into a chain of tasks to execute sequentially across servers. Compared to Natto, most of these systems provide no support for transaction prioritization. Furthermore, while the transaction models in these systems have no support for interactive reads and writes between clients and servers, Natto targets Carousel's transaction model that supports interactive read-modify-write operations, which are preferred by many application developers [42], especially for rapid development [10]. A detailed comparison of these transaction models can be found in the Carousel paper [53].

Increasing data locality is an alternative approach to reduce transaction completion time in geo-distributed systems. CLOCC [6, 35] uses caches on the client side. However, some workloads need large caches, and it is expensive to keep caches consistent. Requiring data to be fully replicated at every datacenter can avoid the need to perform 2PC across datacenters, such as in Replicated Commit [36] and Consus [21], but the storage cost is significant in a deployment that consists of a moderate to large number of datacenters. Also, the replication latency increases with the number of replicas [8]. Microsoft's Cloud SQL Server [11] avoids 2PC by forcing that a transaction can only access the data on one server. By having a write site for each data object, Walter [47] can execute a transaction locally if the local site is the write site for all write objects in the transaction. Otherwise, Walter still needs to run 2PC across datacenters to commit a transaction. Akkio [8] moves data between datacenters as workloads change to increase data locality but it provides no transaction guarantees. Compared to Natto, most existing systems have little to no prioritization support for distributed transactions in a wide-area network environment.

6.2 Transaction Prioritization

Most previous work on supporting transaction prioritization in a database system focuses on transactions that only access data on a single data server instead of distributed transactions. For example, targeting a centralized database system that uses two-phase locking, McWherter et al. [37] propose to schedule high-priority transactions ahead of a queue of transactions that wait for a lock. This work also adopts priority inheritance and priority preemption to reduce the lock-waiting time for high-priority transactions. McWherter et al. [38] propose to only allow a high-priority transaction to preempt a low-priority transaction on a lock if the low-priority transaction waits for another lock.

There are also systems that schedule computing resources (i.e., CPU, memory, and disks) to support transaction prioritization on a single server. For example, Carey et al. [14] study multiple strategies of CPU scheduling, disk scheduling, and memory management in a database system to provide a best-effort service for high-priority transactions while minimizing the negative impact on low-priority transactions. Brown et al. [12] focus on memory management for transactions that have different requirements on the response time. These scheduling mechanisms are not effective for geo-distributed data, where network delays become a main source of latency.

EQMS [44, 45] introduces an external scheduler to limit the number of transactions in a system in order to provide quality of service. When the number of incoming transactions is over the limit, EQMS will buffer transactions and schedule them based on priorities. Using a centralized scheduler in geo-distributed database systems would increase transaction completion time since clients and servers can be in different datacenters from the scheduler.

In real-time database systems, there has been a large body of previous work on transaction scheduling, such as in [1–5, 29–31, 46, 50]. Some of these systems [2, 30, 31] also address problems in priority inheritance and priority preemption techniques. However, most of these systems aim to minimize deadline violations. In contrast, Natto aims to reduce latency for high-priority transactions over low-priority transactions.

7 CONCLUSION

Many applications require low tail latencies for their high-priority transactions. Natto addresses this requirement by providing support for transaction prioritization. It leverages network measurements to establish a global order of transactions based on their estimated arrival time at their furthest participating server. With accurate arrival time estimates, this ordering eliminates transaction aborts for high-priority transactions, and provides opportunities to selectively abort conflicting low-priority transactions. Natto further reduces the tail latency of high-priority transactions through conditional prepare and early committed state forwarding. Our experiments on Microsoft Azure and on a local cluster show that Natto has significantly lower tail latency for high-priority transactions than competing geo-distributed transaction processing systems.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Canada Foundation for Innovation, Ontario Research Fund, and a grant from the Waterloo-Huawei Joint Innovation Lab.

REFERENCES

[1] Robert Abbott and Hector Garcia-Molina. 1988. Scheduling Real-Time Transactions. *SIGMOD Record* 17, 1 (1988).

[2] Robert Abbott and Hector Garcia-Molina. 1988. Scheduling real-time transactions: A performance evaluation. In *VLDB*.

[3] Robert Abbott and Hector Garcia-Molina. 1989. Scheduling real-time transactions with disk resident data. In *VLDB*.

[4] Robert Abbott and Hector Garcia-Molina. 1990. Scheduling I/O requests with deadlines: A performance evaluation. In *Proceedings of the Real-Time Systems Symposium*.

[5] Robert K. Abbott and Hector Garcia-Molina. 1992. Scheduling Real-Time Transactions: A Performance Evaluation. *ACM Transactions on Database Systems* 17, 3 (1992).

[6] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. 1995. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *Proceedings of the International Conference on Management of Data (SIGMOD'95)*.

[7] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. 2007. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'07)*.

[8] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. 2018. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In *OSDI*.

[9] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. 2014. Scalable Atomic Visibility with RAMP Transactions. In *Proceedings of the International Conference on Management of Data (SIGMOD'14)*.

[10] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data System Research (CIDR'11)*.

[11] Philip A. Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B. Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. 2011. Adapting Microsoft SQL Server for Cloud Computing. In *Proceedings of the International Conference on Data Engineering (ICDE'11)*.

[12] Kurt P. Brown, Michael J. Carey, and Miron Livny. 1993. Managing Memory to Meet Multiclass Workload Response Time Goals. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'93)*.

[13] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2009. Serializable Isolation for Snapshot Databases. *ACM Transactions on Database Systems* 34, 4 (2009).

[14] Michael J. Carey, Rajiv Jauhari, and Miron Livny. 1989. Priority in DBMS Resource Scheduling. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'89)*.

[15] Cockroach Labs. 2021. CockroachDB. https://github.com/cockroachdb/cockroach.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the Symposium on Cloud Computing (SoCC'10)*.

[17] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. 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. 2012. Spanner: Google's Globally-Distributed Database. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*.

[18] James Cowling and Barbara Liskov. 2012. Granola: Low-overhead Distributed Transaction Coordination. In *Proceedings of the USENIX Conference on Annual Technical Conference (USENIX ATC'12)*.

[19] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Rohm. 2014. YCSB+T: Benchmarking Web-Scale Transactional Databases. In *Proceedings of the International Conference on Data Engineering Workshops (ICDEW'14)*.

[20] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *VLDB* 7, 4 (2013).

[21] Robert Escriva and Robbert Van Renesse. 2016. Consus: Taming the Paxi. *CoRR* abs/1612.03457 (2016).

[22] etcd. 2017. Raft Implementation. https://github.com/coreos/etcd/tree/master/raft.

[23] Hua Fan and Wojciech Golab. 2018. Scalable Transaction Processing Using Functors. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'18)*.

[24] Hua Fan and Wojciech Golab. 2019. Ocean Vista: Gossip-Based Visibility Control for Speedy Geo-Distributed Transactions. *Proc. VLDB Endow.* 12, 11 (2019).

[25] Hua Fan, Wojciech Golab, and Charles B. Morrey. 2017. ALOHA-KV: High Performance Read-Only and Write-Only Distributed Transactions. In *Proceedings of the Symposium on Cloud Computing (SoCC'17)*.

[26] Fauna Inc. 2021. FaunaDB. https://fauna.com/.

[27] Google. 2020. gRPC-go. https://github.com/grpc/grpc-go.

[28] Google. 2021. Introducing request priorities for Cloud Spanner APIs. https://cloud.google.com/blog/topics/developers-practitioners/introducing-request-priorities-cloud-spanner-apis.

[29] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. 1992. Data Access Scheduling in Firm Real-time Database Systems. *Real-Time Systems* 4, 3 (1992).

[30] Jiandong Huang, John A. Stankovic, Krithi Ramamritham, Don Towsley, and Bhaskar Purimetla. 1992. Priority Inheritance in Soft Real-Time Databases. *Real-Time Systems* 4, 3 (1992).

[31] Jiandong Huang, John A. Stankovic, Krithivasan Ramamritham, and Donald F. Towsley. 1991. On Using Priority Inheritance in Real-Time Databases. In *Proceedings of the Real-Time Systems Symposium*.

[32] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (2008).

[33] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-Data Center Consistency. In *Proceedings of the European Conference on Computer Systems (EuroSys'13)*.

[34] Costin Leau. 2013. Spring Data Redis - Retwis-J. https://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[35] Barbara Liskov, Miguel Castro, Liuba Shrira, and Atul Adya. 1999. Providing Persistent Objects in Distributed Systems. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'99)*.

[36] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. 2013. Low-latency Multi-datacenter Databases Using Replicated Commit. *Proc. VLDB Endow.* 6, 9 (2013).

[37] David T. McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. 2004. Priority Mechanisms for OLTP and Transactional Web Applications. In *Proceedings of the International Conference on Data Engineering (ICDE'04)*.

[38] David T. McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. 2005. Improving Preemptive Prioritization via Statistical Characterization of OLTP Locking. In *Proceedings of the International Conference on Data Engineering (ICDE'05)*.

[39] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*.

[40] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*.

[41] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the USENIX Conference on Annual Technical Conference (USENIX ATC'14)*.

[42] Andrew Pavlo. 2017. What Are We Doing With Our Lives?: Nobody Cares About Our Concurrency Control Research. In *Proceedings of the International Conference on Management of Data (SIGMOD'17)*.

[43] Thamir Qadah, Suyash Gupta, and Mohammad Sadoghi. 2020. Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication. In *EDBT*.

[44] Bianca Schroeder, Mor Harchol-Balter, Arun Iyengar, and Erich Nahum. 2006. Achieving Class-Based QoS for Transactional Workloads. In *Proceedings of the International Conference on Data Engineering (ICDE'06)*.

[45] Bianca Schroeder, Mor Harchol-Balter, Arun Iyengar, Erich Nahum, and Adam Wierman. 2006. How to Determine a Good Multi-Programming Level for External Scheduling. In *Proceedings of the International Conference on Data Engineering (ICDE'06)*.

[46] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. 1990. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Comput.* 39, 9 (1990).

[47] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional Storage for Geo-Replicated Systems. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'11)*.

[48] Alexander Thomson and Daniel J. Abadi. 2015. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*.

[49] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the International Conference on Management of Data (SIGMOD'12)*.

[50] Özgür Ulusoy and Geneva G. Belford. 1992. Concurrency Control in Real-time Database Systems. In *Proceedings of the ACM Annual Conference on Communications (CSC'92)*.

[51] UWSysLab. 2017. TAPIR Implementation. https://github.com/UWSysLab/tapir.

[52] Xinan Yan, Linguan Yang, and Bernard Wong. 2020. Domino: Using Network Measurements to Reduce State Machine Replication Latency in WANs. In *Proceedings of the 16th International Conference on emerging Networking EXperiments*

and Technologies (CoNEXT '20).

[53] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-Latency Transaction Processing for Globally-Distributed Data. In *Proceedings of the International Conference on Management of Data (SIGMOD'18).*

[54] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'15).*

[55] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. 2013. Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'13).*