Carousel: Low-Latency Transaction Processing for Globally-Distributed Data

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

Linguan Yang
University of Waterloo
l69yang@uwaterloo.ca

Hongbo Zhang
University of Waterloo
hongbo.zhang@uwaterloo.ca

Xiayue Charles Lin
University of Waterloo
xy3lin@uwaterloo.ca

Bernard Wong
University of Waterloo
bernard@uwaterloo.ca

Kenneth Salem
University of Waterloo
kmsalem@uwaterloo.ca

Tim Brecht
University of Waterloo
brecht@uwaterloo.ca

ABSTRACT

The trend towards global applications and services has created an increasing demand for transaction processing on globally-distributed data. Many database systems, such as Spanner and CockroachDB, support distributed transactions but require a large number of wide-area network roundtrips to commit each transaction and ensure the transaction's state is durably replicated across multiple datacenters. This can significantly increase transaction completion time, resulting in developers replacing database-level transactions with their own error-prone application-level solutions.

This paper introduces Carousel, a distributed database system that provides low-latency transaction processing for multi-partition globally-distributed transactions. Carousel shortens transaction processing time by reducing the number of sequential wide-area network roundtrips required to commit a transaction and replicate its results while maintaining serializability. This is possible in part by using information about a transaction's potential write set to enable transaction processing, including any necessary remote read operations, to overlap with 2PC and state replication. Carousel further reduces transaction completion time by introducing a consensus protocol that can perform state replication in parallel with 2PC. For a multi-partition 2-round Fixed-set Interactive (2FI) transaction, Carousel requires at most two wide-area network roundtrips to commit the transaction when there are no failures, and only one roundtrip in the common case if local replicas are available.

CCS CONCEPTS

• **Information systems** → **Distributed database transactions**;
• **Computer systems organization** → *Dependable and fault-tolerant systems and networks*;

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196912

KEYWORDS

globally-distributed data; distributed transactions

ACM Reference Format:
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 2018 International Conference on Management of Data (SIGMOD'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3183713.3196912

1 INTRODUCTION

Geographically distributed database systems have become part of the critical infrastructure for organizations that operate in more than one geographic location. Two prominent examples of geographically distributed database systems are Spanner [11] and CockroachDB [8]. These systems partition their data and store each partition at the datacenter where it will most frequently be used. They also use a consensus protocol, such as Paxos [25] or Raft [40], to replicate each partition to enough additional datacenters to meet their users' fault tolerance requirements.

Although most transactions for these systems are designed to access data from just one partition, multi-partition transactions are often unavoidable for many applications. For example, a number of applications choose to partition their users' data based on their users' geographic locations. Using this partitioning key, a transaction to add someone to a user's friends list would only require access to a single partition in the common case since most members in a social group are from the same geographic region. However, this is not true for traveling users; the same transaction for them will typically require access to two partitions. Furthermore, application workloads often consist of multiple transactions that would benefit from data partitioning on different data attributes. Any one partitioning scheme would likely require some transactions to access multiple partitions. Application requirements also change over time, and new transactions based on updated requirements may not be well suited for an existing partitioning scheme.

To support multi-partition transactions, most geographically distributed database systems perform transaction processing by first fetching the required data to a single site, and then using the two-phase commit protocol (2PC) to ensure that transactions are

atomically committed or aborted. Updates are typically sent together with the first 2PC message and are applied if the transaction commits. Therefore, unless all of the required data is already available at a single site, separate wide-area network roundtrips are required to fetch the data and commit the transaction.

An additional requirement for many distributed database systems is for them to remain available even in the event of a datacenter outage. Spanner and CockroachDB achieve this by using a consensus protocol to replicate both the updates to the database and the changes to the 2PC state machine for each transaction to $2f + 1$ datacenters, where f is the maximum number of simultaneous failures that the systems can tolerate. However, simply layering 2PC on top of a consensus protocol can introduce additional wide-area network roundtrips that can significantly increase the completion time of multi-partition transactions.

In this paper, we introduce Carousel, a globally-distributed database system that provides low-latency transaction processing for multi-partition geo-distributed transactions. We are particularly interested in deployments where data is not fully replicated at every site. Much like in Spanner and CockroachDB, Carousel uses 2PC to ensure that transactions are committed atomically, and a consensus protocol to provide fault tolerance and high availability. However, instead of sequentially processing, committing, and replicating each transaction, Carousel introduces two techniques to parallelize these stages, enabling it to significantly reduce its transaction completion time compared to existing systems.

The first technique uses hints provided by the transaction to overlap transaction processing with the 2PC and consensus protocols. Carousel specifically targets 2-round Fixed-set Interactive (2FI) transactions, where each transaction consists of a round of reads followed by a round of writes with read and write keys that are known at the start of the transaction. Unlike one-shot transactions, the write values of a 2FI transaction can depend on the read results from multiple data partitions. The client can also choose to abort the transaction after receiving the read values. Such transactions are quite common in many applications. Carousel uses properties from this class of transactions to safely initiate 2PC at the start of the transaction, and execute most of the 2PC and consensus protocols independently of the transaction processing. This enables Carousel to return the result of a 2FI transaction to the client after at most two wide-area network roundtrips when there are no failures.

The second technique borrows ideas from Fast Paxos [28] to parallelize 2PC with consensus. In Carousel, each database is divided into multiple partitions, and each partition is stored by a consensus group of servers. The servers in the same consensus group are in different datacenters, and the consensus group leader serves as the partition leader. During a transaction, instead of sending 2PC prepare requests only to the partition leaders, who would normally forward the requests to their followers, prepare requests are sent to every node in the participating partitions. Each node responds with a prepare result using only its local information. If the coordinator receives the same result from a supermajority ($\lceil \frac{3}{2} f \rceil + 1$) of the nodes from a partition, it can safely use that result for the partition. This technique enables Carousel to reduce transaction completion time and complete a 2FI transaction in one wide-area network roundtrip in the common case if local replicas are available.

This paper makes three main contributions:

- We describe the design of Carousel and show that by targeting 2FI transactions, Carousel can return the result of a transaction to the client after at most two wide-area network roundtrips when there are no failures.
- We incorporate ideas from Fast Paxos to parallelize 2PC with consensus, which further reduces transaction completion time in the common case.
- We evaluate Carousel using workloads from the Retwis [30] and YCSB+T [15] benchmarks on both an Amazon EC2 deployment and a local cluster. Our results show that Carousel has lower transaction completion time than TAPIR [50], a state-of-the-art transaction protocol.

2 BACKGROUND AND RELATED WORK

Modern geo-distributed storage systems shard and replicate data across datacenters to provide scalability, high availability, and fault tolerance. There are many methods for managing replicas and supporting distributed transactions across data partitions. In this section, we first briefly review different replication techniques. We then describe how existing storage systems support transactions spanning multiple data partitions.

2.1 Replication Management

Many geo-distributed storage systems (e.g., [4, 8, 11]) replicate transaction states and data by using a consensus protocol to tolerate failures. One of the most widely used consensus protocols is Paxos [25, 26], which needs $2f + 1$ replicas to tolerate f simultaneous failures. In the common case, Paxos uses one roundtrip to choose one proposer's proposal and requires another roundtrip to enforce consensus on the proposed value. Multi-Paxos [48] adopts a long-lived leader to be the sole proposer, which eliminates Paxos' first roundtrip in the absence of a leader failure. Fast Paxos [28] reduces the end-to-end latency by sending a client's request to every member in a consensus group instead of only to the leader. If at least $\lceil \frac{3}{2} f \rceil + 1$ members agree on the request (i.e., a supermajority), the client can learn the consensus result in one roundtrip. If a supermajority cannot be achieved because of concurrent requests, Fast Paxos will fall back to a slow path that requires a leader to coordinate the consensus process. Generalized Paxos [27] further reduces the latency of achieving consensus on non-conflicting concurrent requests by leveraging the commutativity between operations.

Compared with Paxos, Raft [40] explicitly separates the consensus process into leader election and log replication in order to improve the understandability and its implementation. Past work has also looked at addressing other issues with consensus protocols, such as load imbalance [35, 36], low throughput [42], and high replication cost [29].

In addition to consensus protocols, there are also replication protocols, such as Viewstamped Replication [39] and Zab [20], which can provide similar guarantees to Paxos. Renesse et al. [49] provide a summary that details the differences between these protocols.

2.2 Distributed Transactions with Replication

To support transactions where data is distributed and replicated in different geographical datacenters, many storage systems (e.g.,

Megastore [4], Spanner [11], and CockroachDB [8]) layer transaction management, such as concurrency control and two-phase commit (2PC), on top of a consensus protocol. Such an architecture facilitates reasoning about the system's correctness and allows for a relatively straightforward implementation. However, this approach incurs high latency to commit a distributed transaction because it sequentially executes the layered protocols, with each layer requiring one or more wide-area network roundtrips (WANRTs).

Past efforts to improve the performance of either transaction or replication management do not address the high latency of sequentially executing the protocols in a layered system architecture. For example, Calvin [46] introduces a deterministic transaction scheduler to increase throughput, and can use a consensus protocol to replicate data in order to provide fault tolerance. Because Calvin is unable to run its transaction scheduling protocol and data replication protocol in parallel, it requires multiple WANRTs to complete a transaction in a geo-distributed environment. Compared with Calvin, Carousel targets geo-distributed transactions and completes a transaction in at most two WANRTs when there are no failures. Furthermore, unlike Carousel, Calvin does not support transactions that require interactivity between clients and servers, which are common in practice [41]. Other systems, such as Rococo [37], CLOCC [1, 31], and Lynx [51], increase throughput and reduce transaction completion time by introducing different concurrency control mechanisms for distributed transactions. However, these systems still need to sequentially perform transaction and replication management.

Restricting 2PC to involve only nodes within a datacenter can reduce the number of WANRTs that are required to commit a transaction. One such an approach is Replicated Commit [34], which builds Paxos on top of 2PC and can commit a transaction in one WANRT. However, Replicated Commit requires reading data from a quorum of replicas. The number of roundtrips required by the reads can significantly increase the transaction completion time. Granola [13] and Microsoft's Cloud SQL Server [5] also layer replication management on top of transaction management. These systems focus on a single-datacenter deployment, and they are not geo-replicated datastores [2]. Consus [16] executes a transaction within each datacenter independently and achieves consensus on whether to commit the transaction among the datacenters. This method avoids quorum reads and requires three one-way messages across datacenters to complete a transaction in the common case. Both Consus and Replicated Commit require fully replicating all data in every datacenter. This requirement is not cost-effective for a deployment that consists of a moderate to large number of datacenters because the replication costs increase with the number of datacenters [51].

Another approach to reduce the latency of committing a transaction is to merge transaction management with replication management. MDCC [23] uses Generalized Paxos to commit concurrent transactions that have commutative writes. TAPIR [50] proposes an inconsistent replication protocol and resolves consistency issues among replicas in its transaction management systems. Both TAPIR and MDCC use clients as transaction coordinators and have a slow path to commit a transaction when there are conflicts, which may increase the tail latency. In this case, both TAPIR and MDCC require three or more WANRTs to complete a transaction when data replicas are not available in the client's datacenter. In contrast, Carousel completes a transaction in at most two WANRTs when there are no failures. To achieve this, Carousel piggybacks prepare requests on read requests to execute transaction processing in parallel with 2PC and consensus. Using the same approach in TAPIR will cause inconsistent data replicas when TAPIR's coordinators fail. This is because TAPIR's transaction coordinators are not fault-tolerant, and TAPIR's commit operations do not guarantee consistency among replicas.

Limiting the expressiveness of transactions is another method for reducing transaction completion time. For example, Janus [38] targets one-shot transactions [21] that consist of stored procedures. By imposing a restriction that a stored procedure can only access data from a local partition, Janus can complete a transaction in one WANRT. Although Janus can also avoid aborting transactions, it may require three WANRTs to commit conflicting transactions. Sinfonia's mini-transactions [3] require keys and write values to be pre-defined. This enables Sinfonia to process and commit a transaction in parallel in order to reduce transaction completion time. Both one-shot transactions and mini-transactions prevent clients from interactively performing read and write operations to servers. For comparison, Carousel's 2-round Fixed-set Interactive (2FI) transaction model (see Section 3.2) allows clients to perform a round of reads followed by a round of writes for a fixed set of read/write keys. Also, the 2FI model does not require write values to be pre-defined at the start of a transaction.

An alternative approach to achieve low latency in a distributed storage system is to adopt weak consistency or reduce transactions' isolation level, such as eventual consistency in Bayou [44], Dynamo [14], PNUTS [9], Cassandra [24], and TAO [6]; causal consistency in COPS [32] and Eiger [33]; and parallel snapshot isolation in Walter [43]. However, applications that require strong consistency have to build their own application-level solutions, which is error-prone and can introduce additional delays to complete a transaction. Carousel provides both serializability and low latency.

3 DESIGN OVERVIEW

In this section, we describe our assumptions regarding the design requirements for Carousel and the properties of its target workloads. We then outline Carousel's system architecture.

3.1 Assumptions

Our design requirements and usage model assumptions are largely based on published information on Spanner [11]. Carousel's design is influenced by the following assumptions:

Geo-distributed data generation and consumption. Many applications have global users to produce and consume data. We assume that our target application has multiple datacenters in geo-distributed locations to store user data and serve users from their regions. Carousel assumes that data servers are running within datacenters, and Carousel clients are application servers running in the same datacenters as the data servers.

Scalability, availability, and fault-tolerance. Modern distributed storage systems shard data into partitions to improve scalability, and each partition is replicated at multiple geo-distributed

sites to provide high availability and fault-tolerance. Carousel targets the fail-stop failure model and an asynchronous environment, where the communication delay between two servers can be unbounded. Therefore, it is necessary to use a consensus protocol to manage replicas. To tolerate f simultaneous failures, standard Paxos or Raft requires the presence of $2f + 1$ replicas. We also wish to keep the choice of replication factor independent of the total number of deployed sites, as it is not cost effective to replicate all partitions at every site in deployments with a large number of sites. Furthermore, as the number of sites increases, fully replicating data at every site will increase the quorum size required to achieve consistency, which may incur higher latency due to the wider differences in network latency among sites. As a result, Carousel targets deployments where data is not fully replicated at every site.

Replica locations. Because data is not fully replicated at every site, some transactions must access data in remote sites. There are two main types of transactions based on the locations of replicas:

- Local-Replica Transactions (LRTs): every partition that the transaction accesses has a replica at the client's site.
- Remote-Partition Transactions (RPTs): the transaction accesses at least one partition that does not have replicas at the client's site.

Compared with previous work (e.g., [23, 34, 38, 50]) that focuses on reducing transaction completion time for LRTs, Carousel aims to reduce transaction completion time for RPTs. However, Carousel still achieves latency that is as low as other systems' latencies for LRTs or transactions that only involve one partition.

Wide-area network latency. We assume that the processing time in geo-distributed transactions is low, so that the wide-area network latency dominates the transaction completion time because 2PC and consensus protocols may require multiple wide-area network roundtrips. Reads to remote sites, such as in RPTs, further increase the number of wide-area network roundtrips. Therefore, the goal of Carousel is to minimize the number of wide-area network roundtrips to complete a transaction.

Interactive transactions. As found by Baker et al. [4], many applications prefer interactive transactions involving both reads and writes, especially in order to support rapid development. Many existing geo-distributed systems (e.g., [4, 11, 23, 34, 50]) target interactive transactions. Carousel also targets interactive transactions by supporting 2FI transactions.

3.2 2FI Transactions

In this paper, we introduce a new transaction model, which we call the 2-round Fixed-set Interactive (2FI) model. A 2FI transaction performs one or more keyed record read and write operations in two rounds: a read round, followed by a write round. In addition, *all read and write keys must be known in advance.*

One important property of 2FI transactions is that, while write keys must be known in advance, write *values* need not be known. Write values can depend on reads. This is important, because it means that 2FI transactions can directly implement common read-modify-write patterns in transactions. For example, a 2FI transaction can read a counter, increment its value, and write the updated value back to the counter, within the scope of a single transaction. In this sense, 2FI transactions are more expressive than other restricted transaction models, such as *mini-transactions* [3], which require write values to be known in advance.

Although 2FI transactions must have read and write keys specified in advance, there is no restriction on *which* keys are read and written. In particular, if the database is partitioned, there is no restriction limiting a 2FI transaction to a single partition. This distinguishes 2FI from models, such as the *one-shot* model[21, 38], which limit read and write operations to a single partition.[1]

Finally, an important property of 2FI transactions is that all read operations can be performed concurrently during the first round, since all read keys are known in advance. In the geo-distributed, partial replication setting targeted by Carousel, this property is particularly significant. Since data are only partially replicated, local reads may be impossible. The 2FI model ensures that all read operations can be performed with at most one wide-area network roundtrip, unless there are failures. However, the flip side of this restriction is 2FI transactions cannot perform *dependent* reads and writes. Dependent reads and writes are those for which the key to be read or written depends on the value of a previous read. This is the major restriction imposed by 2FI.

Dependent reads and writes do occur in real transactional workloads, although their frequency will of course be application specific. As noted by Thomson and Abadi [45], one situation that gives rise to dependent reads and writes is access through a secondary index. For example, in TPC-C, Payment transactions may identify the paying customer by customer ID (the key) or by customer name. In the latter case, the customer key is not known in advance. The transaction must first look up the customer ID by name (using a secondary index), and then access the customer record. This requires a sequence of two reads, the second dependent on the first, which is not permitted in a 2FI transaction.

Although the 2FI model prohibits such transactions, there is an application-level workaround that can be used to perform dependent reads and writes when necessary. The key idea is to eliminate the dependency by introducing a *reconnaissance transaction* [45]. In the TPC-C Payment example, the application would first perform a reconnaissance transaction that determines the customer ID by accessing a secondary index keyed by customer name. This is a 2FI transaction, since the name is known in advance. Then, the application issues a modified Payment transaction, using the customer ID returned by the reconnaissance transaction. The Payment transaction is modified to check that the customer's name matches the name used by the reconnaissance transaction. If it does not, the Payment transaction is aborted, and both transactions are retried. The modified Payment transaction is also 2FI, since the customer key (the ID) is known when the transaction starts, thanks to the reconnaissance transaction.

3.3 Architecture

Carousel provides a key-value store interface with transactional data access. It consists of two main components: a client-side library and Carousel data servers (CDSs) that manage data partitions. Carousel uses a directory service, such as Chubby [7] or

[1] 2FI transactions are neither stronger than nor weaker than one-shot transactions, since one-shot transactions do not require read and write keys to be known in advance.

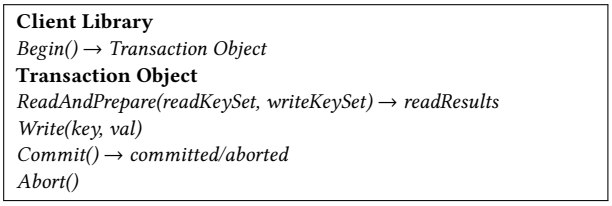

Figure 1: Carousel's client interface.

Zookeeper [19], to keep track of the locations of the partitions and their data servers. Carousel's client-side library caches the location information and infrequently contacts the directory service to update its cache. Carousel uses consistent hashing [22] to map keys to partitions.

Carousel's clients are application servers that run in the same datacenters as CDSs. Each client has a unique ID and a Carousel client-side library. The library provides a transactional interface as shown in Figure 1. To execute a transaction, the client first calls the *Begin()* function to create a transaction object that assigns the transaction a unique transaction ID (TID). A TID is a tuple consisting of the client ID and a transaction counter that is unique to the client. The client uses the transaction object to perform all reads by calling the *ReadAndPrepare()* function once. The client uses the *Write()* function to perform writes, and the write data is buffered by Carousel's client-side library until the client issues a commit or abort for the transaction. Furthermore, if a client does not specify write keys when calling the *ReadAndPrepare()* function, Carousel will execute the transaction as a read-only transaction.

To provide fault-tolerance, Carousel replicates data partitions in different datacenters. Each datacenter consists of a set of CDSs, and a CDS stores and manages one or more partitions. Carousel extends Raft to manage replicas, and the replicas of a partition together form a consensus group. A consensus group requires $2f + 1$ replicas to tolerate up to f simultaneous replica failures, and Carousel reliably stores transactional states and data on every member in the group.

When a transaction accesses (reads or writes) data from a partition, that partition becomes one of its *participant partitions*. The leader of a participant partition's consensus group is called a *participant leader*, and other replicas in the group are *participant followers*. For each transaction, Carousel selects one consensus group to serve as the *coordinating consensus group* for that transaction. The leader of the coordinating consensus group is referred to as the transaction *coordinator*.

The Carousel client always selects a *local* participant leader to serve as the transaction coordinator, if such a local leader exists. Otherwise, the Carousel client can choose any local consensus group leader to act as the transaction coordinator. Carousel expects that partitions are deployed such that each datacenter has at least one consensus group leader so that clients can always choose a local coordinator. It is also possible for Carousel to intentionally create consensus groups that are not CDSs to serve as coordinators. Unlike protocols that use clients as transaction coordinators, such as TAPIR [50], Carousel's coordinators are fault tolerant, as their states are reliably replicated to their consensus group members.

Carousel uses optimistic concurrency control (OCC) and 2PC to provide transactional serializability. Each data record in Carousel has a version number that monotonically increases with transactional writes, and our OCC implementation uses the version number to detect conflicting transactions.

4 PROTOCOL

In this section, we first describe Carousel's basic transaction protocol that takes advantage of the properties of 2FI transactions (see Section 3.2) to perform *early 2PC prepares*. We then introduce a consensus protocol that can safely perform state replication in parallel with 2PC, and describe how that is used in an improved version of Carousel. Finally, we introduce additional optimizations for Carousel to further reduce its transaction completion time.

4.1 Basic Carousel Protocol

Each Carousel transaction proceeds through a sequence of three execution phases. First is the *Read* phase, which begins with a `ReadAndPrepare` call from the client. During the Read phase, Carousel contacts the participant leaders to obtain values for all keys in the transaction's read set. In general, this phase may require one wide-area network roundtrip (WANRT), since some of the participant leaders may be remote from the client. Next is the *Commit* phase, which begins when the client calls `Commit`, supplying new values for some or all of the keys in the transaction's write set. During this phase, the client contacts the transaction coordinator to commit the transaction. The coordinator replicates the transaction's writes to the coordinator's consensus group before acknowledging the commit to the client. The Commit phase requires one WANRT to replicate the transaction's writes. After committing, the transaction enters the *Writeback* phase, during which the participant leaders are informed of the commit decision. This phase requires additional WANRTs. However, the Writeback phase is fully asynchronous with respect to the client.

In addition to the Read, Commit, and Writeback phases, which occur sequentially, the Carousel protocol includes a fourth phase, called *Prepare* which runs *concurrently* with the Read and Commit phases. This concurrent *Prepare* phase is a distinctive feature of Carousel. The purpose of the Prepare phase is for each participant leader to inform the coordinator whether it will be able to commit the transaction within its partition.

Figure 2 shows an example of the basic Carousel protocol when there are no failures. In this example, the client, the coordinator, and one participant leader are located in one datacenter (DC_1), and a second participant leader is located in a remote datacenter (DC_2). To simplify the diagram, the participant followers are not shown, and neither are the other members of the coordinator's consensus group. In the remainder of this section, we describe each of Carousel's execution phases in more detail. In our description, we use circled numbers (e.g., ①) to refer to the corresponding numbered points in the protocol shown in Figure 2.

4.1.1 Read Phase. During the read phase, a client sends (①) read requests to each participant leader, identifying the keys to be read from that partition. The participant leaders respond (③,⑤) to the client with the latest committed value of each read key. After reading, the client may update some or all of the keys in its write

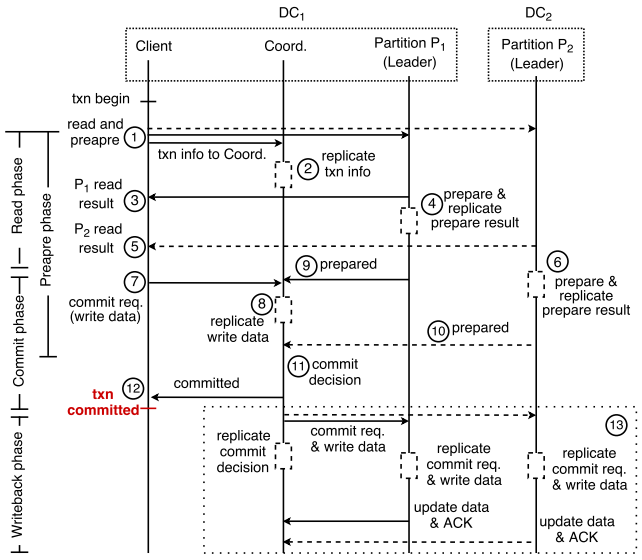

Figure 2: An example of Carousel's basic transaction protocol. Solid and dashed arrows stand for intra-datacenter and inter-datacenter messages, respectively, and dashed rectangles represent replication operations.

set by calling `Write`. Such updates are simply recorded locally by the Carousel client. The application finally calls either `Commit` (or `Abort`), which initiates Carousel's Commit phase.

4.1.2 Commit Phase. If the application decides to commit, the Carousel client initiates the Commit phase by sending (⑦) a `commit` request, including all updated keys and their new values, to the coordinator. Upon receiving `commit`, the coordinator replicates (⑧) the transaction's updates to its consensus group, which requires one WANRT. After replicating the updates, the coordinator must wait to receive (⑨,⑩) `prepared` messages from all participant leaders before it can commit the transaction. These `prepared` messages are generated as a result of the Prepare phase, which we will describe later in Section 4.1.4.

If all participant leaders successfully prepare, the coordinator decides (⑪) to commit the transaction and immediately sends (⑫) `committed` to the client. This is safe because the transaction's updates are replicated in the coordinator's consensus group, and prepare decisions have been replicated in all participant partitions during the Prepare phase. If there is a coordinator failure, Carousel can recover the transaction's data and state from the corresponding consensus groups (see Section 4.3). If any participant leader indicates that it has failed to prepare the transaction, then the coordinator aborts the transaction and replies to the client with `aborted`. In this case, the coordinator can reply immediately, without waiting for writes to be replicated and without waiting for messages from other participants. To ensure that the coordinator's response to the client is consistent with the actual outcome of the transaction, Carousel prohibits the coordinator from unilaterally aborting the transaction once it has replicated the transaction's write data. It may abort only once it learns that at least one participant leader failed to prepare.

If the application chooses to abort the transaction rather than commit it, the client sends `abort` to the coordinator. The coordinator may abort the transaction immediately, without waiting for `prepared` messages from participant leaders.

4.1.3 Writeback Phase. The purpose of Carousel's Writeback phase is to distribute the transaction's updates and commit decision to the participants. The coordinator initiates this phase by sending (⑬) a `commit` message to each participant leader. This message includes the transaction's commit decision and, if the transaction committed, its updates. Each participant leader then replicates this information to its consensus group and returns an acknowledgment to the coordinator. While the participants are updating their state, the coordinator replicates the transaction commit decision to its consensus group. This is not necessary to ensure that the transaction commits, but it simplifies recovery in the event of a coordinator failure. The entire Writeback phase requires two WANRTs. However, none of this latency is exposed to the Carousel client application.

4.1.4 Prepare Phase. Carousel's Prepare phase starts at the same time as the Read phase, and runs concurrently with Read and Commit. When the application calls `ReadAndPrepare`, the Carousel client piggybacks a `prepare` request on the `read` request that it sends to each participant leader. The `prepare` request to each participant leader includes the transaction's read and write set for that partition, and also identifies the transaction coordinator.

When a participant leader receives a `prepare` request, it uses the transaction's read and write set information to check for conflicts with concurrent transactions. To do this, each participant leader maintains a list of pending (prepared, but not yet committed or aborted) transactions, along with their read and write sets. The leader checks for read-write and write-write conflicts between the new transaction and pending transactions. If there are none, it adds the new transaction to its pending list, marks the new transaction as prepared, and replicates (④,⑥) the prepare decision, along with the new transaction's read set, write set, and read versions, to the participant followers in the partition's consensus group. Finally, the participant leader sends (⑨,⑩) a `prepared` message to the transaction coordinator. If the participant leader's conflict checks do detect a conflict, it will fail to prepare the transaction. In this case, it will replicate an abort decision to its consensus group, and then send an `abort` message to the coordinator.

When the client piggybacks its `prepare` messages to the participant leaders, it also sends a similar `prepare` message to the transaction coordinator. When it receives this message, the coordinator replicates (②) the transaction's read set and write set to its consensus group. This ensures that the coordinator is aware of all of the transaction's participants.

If there are no failures, the Prepare phase requires at most two WANRTs. One WANRT is required (in general) to send `prepare` requests from the client to the participant leaders, and to return the participant leaders' prepare decisions to the coordinator (which is located in the same datacenter as the client). The second WANRT is required for each participant leader to replicate its prepare decision to its consensus group. However, since the Prepare phase runs

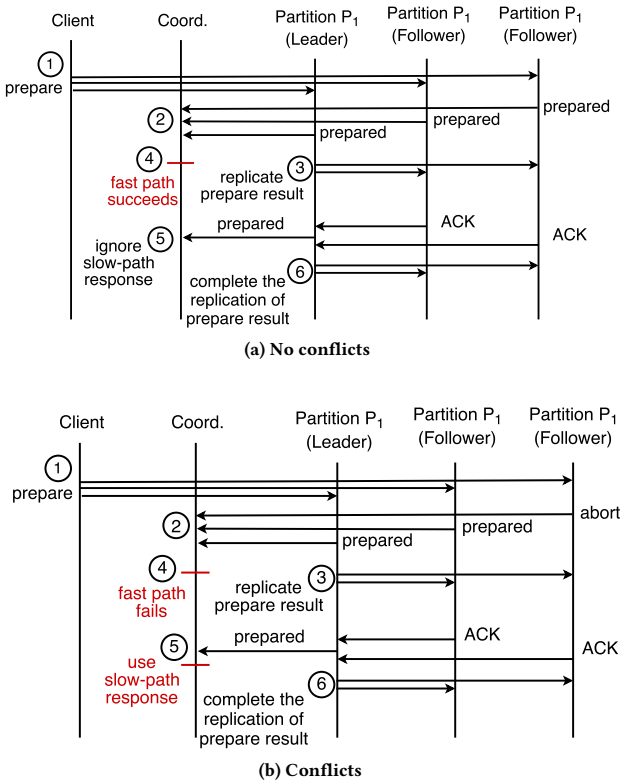

Figure 3: An example of CPC.

concurrently with the Read and Commit phases, each of which requires one WANRT, the total number of WANRT delays observed by the client is at most two.

4.2 Parallelizing 2PC and Consensus

In the basic Carousel transaction protocol, 2PC and consensus together require two wide-area network roundtrips (WANRTs) to complete Carousel's Prepare phase. In this section, we introduce Carousel's Prepare Consensus (CPC) protocol, which can safely run in parallel with 2PC while replicating the transaction's internal state. This allows Carousel's Prepare phase to complete after one WANRT in many situations.

CPC borrows ideas from both Fast Paxos [28] and MDCC [23] to introduce a fast path that can prepare a transaction in one WANRT if it succeeds. However, the fast path may not succeed if the transaction is being prepared concurrently with conflicting transactions. In this case, CPC must instead complete the Prepare phase using its slow path, which is just the Prepare phase in Carousel's basic transaction protocol. Unlike in Fast Paxos and MDCC where the slow path only starts after the fast path fails, CPC executes both paths in parallel. As a result, CPC can prepare a transaction in at most two WANRTs when there are no failures.

We use an example in Figure 3 (a) to illustrate CPC when there are no conflicting transactions. In CPC, a client sends (①) a prepare request to every participant leader and follower, which starts both the fast path and the slow path. Like in Carousel's basic transaction protocol, this request includes the transaction's read and write keys. Upon receiving the prepare request, on the fast path, each participant will independently prepare the transaction by checking read-write and write-write conflicts with concurrent transactions. To do this, each participant maintains a persistent list of pending transactions along with their read and write keys. This list is called a *pending-transaction list*. If there are no conflicts, a participant will send (②) a prepared message to the coordinator; otherwise, the participant will send an abort message. Meanwhile, on the slow path, the participant leader replicates (③) its prepare result to its consensus group.

On the fast path, the coordinator can determine a participant partition's prepare decision for a transaction if both of the following conditions are satisfied:

(1) It receives the same prepare decision from a supermajority [2] of the consensus group members, where every member in the supermajority is up-to-date.
(2) The participant leader must be part of the supermajority.

A group member is up-to-date if (a) it uses the same data versions to prepare the transaction as the participant leader; and (b) it is in the same *term* as the participant leader. A term is defined in Raft [40] as a period of time when a consensus group has the same leader, and it changes when a new leader is elected. The data versions and the term information are also stored in each participant's pending-transaction list. These requirements are needed to tolerate leader failures, which we will describe in Section 4.3. Furthermore, the need for the participant leader to be part of the supermajority stems from the requirement that CPC must safely run the fast path in parallel with the slow path. Specifically, CPC ensures that if the fast path succeeds, the fast path and the slow path will arrive at the same prepare decision, which is the participant leader's decision.

In the case where both of these conditions are satisfied for a participant partition, and the supermajority of the participants have chosen to prepare the transaction, the coordinator considers (④) the transaction to be prepared on the partition. Under the same two conditions, if the supermajority of the participants abort the transaction, the coordinator considers that the partition aborts the transaction. By sending prepare requests directly to every participant, CPC can determine if a transaction is prepared on a participant partition in one WANRT when both conditions are satisfied; that is, the fast path succeeds. For a partition where the fast path succeeds, the coordinator simply drops (⑤) the response from the slow path. Finally, on the slow path, the participant leader completes (⑥) replicating its prepare result to the participant followers, which can be done asynchronously and is not on the critical path.

For cases where multiple conflicting transactions are concurrently processed, it is possible for the transactions to not satisfy the two conditions. In this scenario, the fast path fails for the transactions, and the coordinator waits for the response from the slow path that executes in parallel with the fast path. We now use Figure 3 (b) to illustrate the case when the fast path fails because of

[2] A supermajority consists of $\lceil \frac{3}{2} f \rceil + 1$ members from a consensus group that has total $2f + 1$ members. This supermajority size is required for the consensus group to agree on an operation in one network roundtrip while tolerating up to f member failures [28].

conflicting transactions. The first three steps for this case are the same as the first three steps for the non-conflicting transaction case. However, in this example, the coordinator does not receive the same prepare decision from a supermajority of participants from the same partition. It must then wait for a response from the participant leader executing the slow path. Once it receives (⑤) the slow-path response, it uses the participant leader's prepare decision as the partition's decision. Just as in the non-conflicting case, the participant leader completes (⑥) replicating its prepare decision to its followers.

4.3 Handling Failures

To meet the fault tolerance and availability demands of large-scale distributed applications, Carousel must provide uninterrupted operations (with reduced performance) even with up to f simultaneous replica failures in a single partition. In this section, we describe in turn how Carousel handles client, follower, and leader failures.

4.3.1 Client Failures. While executing a transaction, the client sends periodic heartbeat messages to the coordinator of the transaction. Until the coordinator receives a commit message from the client, it will abort the transaction if it fails to receive h consecutive heartbeat messages from the client. After receiving the client's commit message, the coordinator will attempt to commit the transaction even if the client fails before the transaction completes.

4.3.2 Follower Failures. Carousel uses Raft to handle follower failures. Raft can operate without blocking with up to f follower failures. Therefore, Carousel can execute a transaction with up to f follower failures in a partition.

4.3.3 Leader Failures. In Carousel's basic protocol, the state of each participant leader is replicated to its consensus group using Raft [40] after each state change and before the state change has been made visible to the coordinator. As a result, in the event of a participant leader failure, Raft will elect a new participant leader for the partition, and the new participant leader has all of the necessary state information to continue processing its pending transactions.

Handling a participant leader failure during the Prepare phase of a transaction is more complicated when using Carousel's Prepare Consensus (CPC) protocol that overlaps consensus with 2PC. This is due to the need for a newly elected participant leader to arrive at the same prepare decisions that may have been exposed to the coordinator via the fast path. For example, the coordinator has determined a transaction to be prepared via the fast path, but the participant leader fails before starting to replicate its prepare result to its consensus group. In this case, the new participant leader must reliably replicate the same prepare result to its consensus group because the coordinator may have decided to commit the transaction and have notified the client. To achieve this, CPC introduces a failure-handling protocol that builds on both Raft's leader election protocol and the failure handling approach in Fast Paxos [28]. Specifically, the failure-handling protocol for a participant leader consists of the following steps:

1. Leader Election. To elect a new leader, Carousel extends Raft's leader election protocol by making each participant piggyback its pending-transaction list on its vote message. The new leader will use the lists to determine which transactions could have been prepared via the fast path. Specifically, a coordinator considers a transaction to be prepared on a partition if the fast path of CPC succeeds. The new leader will buffer requests from clients and coordinators until it completes the failure-handling protocol.

2. Completing replications. Before determining which transactions have been prepared via the fast path, the new leader first completes replicating any uncommitted log entries in its consensus log to its followers, which follows Raft's log replication procedure. Raft's leader election protocol guarantees that the new leader has the latest log entries. By replicating these log entries, the new leader ensures that its consensus group has reliably stored the prepare results of *slow-path prepared transactions*, which are transactions that have been already partially replicated by the failed leader to its consensus group using the slow path.

3. Examining pending-transaction lists. If a pending transaction has been prepared via the fast path, which we call a fast-path prepared transaction, the new leader must arrive at the same prepare decision. The new leader does not know for certain whether the fast path has succeeded for a transaction. Therefore, in order to determine if a transaction could have been prepared via the fast path, the new leader examines the pending-transaction lists taken from the vote messages that it received from a majority of participants, where each participant in the majority has voted for it during leader election. A fast-path prepared transaction must have been prepared on a supermajority ($\lceil \frac{3}{2} f \rceil + 1$) of the participants including the failed leader. With up to f participant failures, the transaction must be in at least a majority of $f + 1$ pending-transaction lists. As a result, the new leader only selects $f+1$ pending-transaction lists for further examination. A transaction could potentially be a fast-path prepared transaction if it is prepared with the same data versions and in the same term (see Section 4.2) in at least a majority of the $f + 1$ lists.

4. Detecting conflicts. If a transaction satisfies the condition in step 3, it still may not have been actually prepared via the fast path. One reason is that the failed leader may have decided not to prepare the transaction. Instead, it decided to prepare other conflicting transactions. Also, the pending-transaction lists include the data versions that the transaction depends on. If the versions are stale, the transaction must not have been prepared via the fast path because the leader always has the latest data versions. Therefore, the new leader should not only consider each transaction individually but also examine all pending transactions to exclude the transactions that conflict with the slow-path prepared transactions or are prepared based on stale data versions. For every potential fast-path prepared transaction in step 3, if the transaction does not conflict with the slow-path prepared transactions determined in step 2, and it is prepared based on the latest data versions, then the new leader considers the transaction to be a fast-path prepared transaction.

5. Replicating fast-path prepared transactions. For all of the fast-path prepared transactions in step 4, the new leader replicates their prepare results to its consensus group. Once the replication is finished, the failure-handling protocol completes. The new leader can now process requests from clients and coordinators, including those that were buffered previously.

Carousel also replicates the state of coordinators to their respective consensus groups using Raft. However, the coordinator reveals its commit decision to the client before it replicates its decision. This is because the coordinator's commit decision is based entirely on the client's commit request and write data, which it has already replicated to its consensus group members, and the participant leaders' prepare phase responses, which have been replicated to their respective consensus groups. In the event of a coordinator failure, the failed coordinator's consensus group will elect a new coordinator. The new coordinator will reacquire the prepare responses from the participant leaders. The prepare responses together with the saved write data allow the new coordinator to arrive at the same commit decision as the previous coordinator.

4.4 Optimizations

This section describes two additional optimizations for Carousel to reduce its transaction completion time. One optimization allows clients to read data from local replicas, and the other optimization targets reducing the completion time for read-only transactions.

4.4.1 Reading from Local Replicas. In practice, a participant leader may not be the closest replica to the client. A participant follower may be in the client's datacenter while the participant leader is in a different datacenter. Allowing a client to read data from a participant follower that is in the same datacenter will reduce the read latency by avoiding a wide-area network roundtrip to the participant leader.

To support reading data from a local replica, Carousel's client-side library will send a read request to the participant follower that is located in the client's datacenter while sending read and prepare requests to the remote participant leader. After receiving a read request, the participant follower returns its read data to the client. The client uses the first return value that it receives from the participant follower or the participant leader.

The data read from a participant follower may be stale. To guarantee serializability, the coordinator determines if the read data is stale. Specifically, the client's commit request to the coordinator will include the read versions received from the participant follower, and participant leaders carry their read versions on their prepare responses to the coordinator. The coordinator uses the read versions to determine whether the client has read stale data. If the client has read stale data, the coordinator will abort the transaction. Using the same approach, Carousel can also support reading from any replica, such as reading from the closest replica when there is no local replica.

By using Carousel's Prepare Consensus (CPC) protocol and reading data from local replicas, Carousel can complete a transaction in one wide-area network roundtrip if all of the participant partitions have replicas in the client's datacenter.

4.4.2 Read-only Transactions. In practice, read-only transactions are common. For a read-only transaction, Carousel's client-side library sends read requests to participant leaders, and there is no coordinator. After receiving the read request, each participant leader performs OCC validation to detect read-write conflicts. If there are no conflicts, the participant leader returns the read data to the client. Otherwise, the participant leader returns *aborted*. The client completes the transaction when it receives all the required read data. The transaction is aborted if the client receives an aborted response from a participant leader. With this optimization, Carousel can complete read-only transactions in one network roundtrip.

	US East	Euro	Asia	Australia
US West	73	166	102	161
US East	-	88	172	205
Euro	-	-	235	290
Asia	-	-	-	115

Table 1: Roundtrip network latencies between different datacenters (ms).

5 IMPLEMENTATION

We have implemented a prototype of Carousel's basic transaction protocol and Carousel's Prepare Consensus (CPC) protocol using the Go language. Our implementation also includes the optimizations for reading data from local replicas and read-only transactions. The implementation consists of about 3,500 lines of code for the protocols. Our prototype builds on an in-memory key-value store and uses gRPC [17] to implement the RPC functions for data servers. Although we extend an open-source implementation [12] of Raft [40] to manage replicas for each partition, we do not implement fault tolerance in our prototype.

Our evaluation (see Section 6) studies two versions of Carousel: *Carousel Basic*, which uses Carousel's basic transaction protocol, and *Carousel Fast*, which uses CPC and supports reading data from local replicas. Both Carousel Basic and Carousel Fast include the optimization for read-only transactions.

6 EVALUATION

In this section, we evaluate Carousel Basic and Carousel Fast by comparing their performance with TAPIR [50], which represents the current state-of-the-art in low-latency distributed transaction processing systems. Our experiments are primarily performed using our prototype implementation running on Amazon EC2. We also perform experiments on a local cluster to evaluate the throughput and network utilization of the three systems.

6.1 Experimental Setup

We deploy our prototype on Amazon EC2 instances across 5 datacenters in different geographical regions: US West (Oregon), US East (N. Virginia), Europe (Frankfurt), Australia (Sydney), and Asia (Tokyo). Table 1 shows the roundtrip network latencies between the different datacenters. Our Amazon EC2 deployment uses *c4.2xlarge* instances, each of which has 8 virtual CPU cores and 15 GB of memory. We configure the systems under evaluation to use 5 partitions with a replication factor of 3, resulting in deployments with a total of 15 servers. In our configuration, each datacenter contains at most one replica per partition. This ensures that a datacenter failure would cause partitions to lose at most one replica. Servers are uniformly distributed across the 5 datacenters so that each datacenter contains 3 partitions of data. One server in each datacenter is a partition leader to one of the partitions. To drive our workload,

Transaction Type	# gets	# puts	workload%
Add User	1	3	5%
Follow/Unfollow	2	2	15%
Post Tweet	3	5	30%
Load Timeline	rand(1, 10)	0	50%

Table 2: Transaction profile for Retwis from TAPIR [50].

we deploy 4 machines per datacenter (the same datacenters as the servers) running 5 clients per machine.

In order to evaluate the performance of TAPIR, we use the open-source implementation [47] provided by TAPIR's authors. We had to modify the implementation to allow TAPIR to issue multiple independent read requests concurrently from the same transaction. We have verified that our changes do not affect TAPIR's performance.

6.2 Workloads

We evaluate our system using two different workloads. The first workload is Retwis [30], which consists of transactions for a Twitter-like system. These transactions perform operations such as adding users, following users, getting timelines, and posting tweets, with each transaction touching an average of 4.5 keys. The second workload is YCSB+T [15], which extends the YCSB key-value store benchmark [10] to support transactions. In our evaluation, each YCSB+T transaction consists of 4 read-modify-write operations that access different keys. Both Retwis and YCSB+T were used by TAPIR [50] to evaluate their system. We configure the workloads based on TAPIR's published configurations. For Retwis, this includes using the distribution of transaction types from TAPIR; the distribution of transaction types is reproduced in Table 2.

For both workloads, we populate Carousel Basic, Carousel Fast, and TAPIR with 10 million keys. Each client can only have one outstanding transaction at a time. The popularity distribution of the keys follow a Zipfian distribution with a coefficient of 0.75. We run each experiment for 90 seconds and exclude the results from the first and last 30 seconds of the experiment. We repeat each experiment 10 times and show the 95% confidence intervals of the data points using error bars.

6.3 Retwis Amazon EC2 Experiments

We now evaluate the performance of the different systems using the Retwis workload. Figure 4 shows the CDF of latencies for Carousel Basic, Carousel Fast, and TAPIR with each system receiving 200 transactions per second (tps). We use a relatively light transaction load to focus on the performance of the system when network latency, rather than resource contention, is the primary latency source. We will later evaluate the performance of these systems under a heavy load in our throughput experiments in Section 6.4. The CDF shows that both Carousel Fast and Carousel Basic have lower latencies than TAPIR over the entire distribution. TAPIR has a median latency of 334 ms compared to 232 ms for Carousel Fast and 290 ms for Carousel Basic. The performance gap widens at higher percentiles.

There are several reasons why Carousel Fast and Carousel Basic have lower latencies than TAPIR:

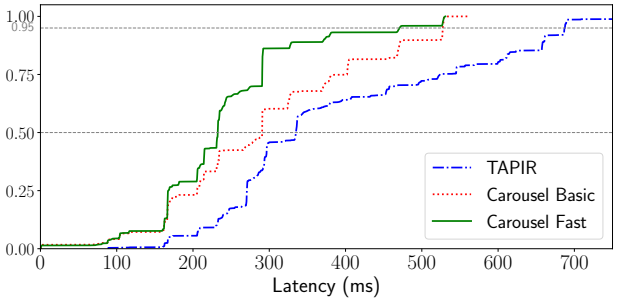

Figure 4: Latency CDF for the Retwis workload.

- Both versions of Carousel require a maximum of only two wide-area network roundtrips to complete a transaction in the absence of failures, while TAPIR can require as many as three wide-area network roundtrips.
- 50% of the Retwis workload consists of read-only transactions. Our read-only transaction optimization allows both versions of Carousel to complete a read-only transaction in just one wide-area network roundtrip.
- TAPIR waits for a fast path timeout before it begins its slow path to commit a transaction. This can result in long tail latencies.
- TAPIR does not allow a client to issue a transaction that potentially conflicts with its own previous transaction until the previous transaction has been fully committed on TAPIR servers. This increases the transaction completion time for a small number of transactions.

Carousel Fast has a lower latency than Carousel Basic due to its fast path, which allows it to complete its Prepare phase in one wide-area network roundtrip. This fast path benefits any transactions where the combined latency of the Read and Commit phases is lower than the latency of the Prepare phase using the slow path. This can occur when the wide-area network latencies from the client to the participant leaders are higher than the latencies between the coordinator and its consensus group followers. Furthermore, for transactions where local replicas are available for all of the keys in the transaction read set, the Read phase only has to perform local read operations. Therefore, Carousel Fast can complete each of these transactions in just one wide-area network roundtrip.

6.4 Retwis Local Cluster Experiments

In addition to running experiments on Amazon EC2, we conduct experiments on our local cluster to compare the throughput and network utilization of the different systems. We simulate 5 geographically distributed datacenters by using TC [18] to introduce network latencies between groups of machines. Our local cluster consists of 15 machines (3 per simulated datacenter) used for Carousel or TAPIR servers, and up to 40 machines (8 per simulated datacenter) used for clients to issue transactions. Each machine has 64 GB of memory, a 200 GB Intel S3700 SSD, and two Intel Xeon E5-2620 processors with a total of 12 cores running at 2.1 GHz. The machines are connected to a 1 Gbps Ethernet network. We use our local cluster for these experiments because experiments that

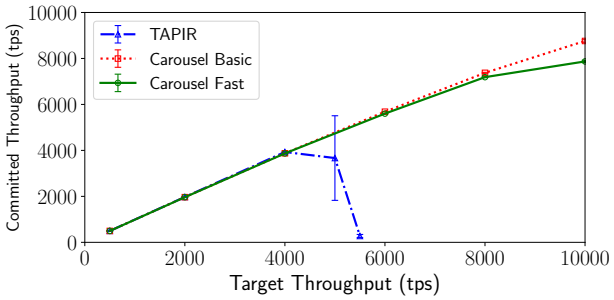

Figure 5: Committed throughput versus target throughput.

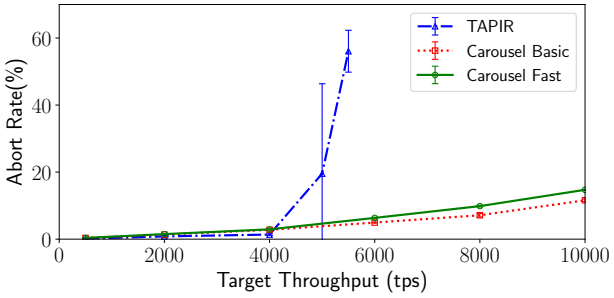

Figure 6: Abort rate versus target throughput.

require high throughput between a large number of geographically distributed servers are prohibitively expensive to run on Amazon EC2.

We use the same Retwis workload as in our Amazon EC2 experiments. We also configure Carousel Basic, Carousel Fast, and TAPIR to use the same system parameters as those used in our Amazon EC2 deployment. However, instead of using TC to introduce network latencies between datacenters based on Amazon EC2 latencies, we introduce a 5 ms latency between simulated datacenters. This choice of network latency allows us to reach the systems' peak throughput using the 40 available client machines.

6.4.1 Throughput. We examine the throughput of the systems under evaluation by increasing the target transaction rate (i.e., target throughput) of the clients, while measuring the number of committed transactions per second, which we call its committed throughput. Figure 5 shows that Carousel Basic, Carousel Fast, and TAPIR are all able to satisfy a target throughput of approximately 5000 tps. Past that point, TAPIR is unable to meet the target throughput. It experiences excessive queuing of pending transactions at the TAPIR servers, resulting in a precipitous drop in its committed throughput.

Carousel Basic's committed throughput only begins to drop below the target throughput at approximately 8000 tps. Its committed throughput continues to increase as we increase the target throughput to 10000 tps. Carousel Basic can achieve a higher committed throughput than TAPIR due to lower transaction latencies, which results in reduced data contention at the server for the same throughput. Carousel Fast's committed throughput falls below the

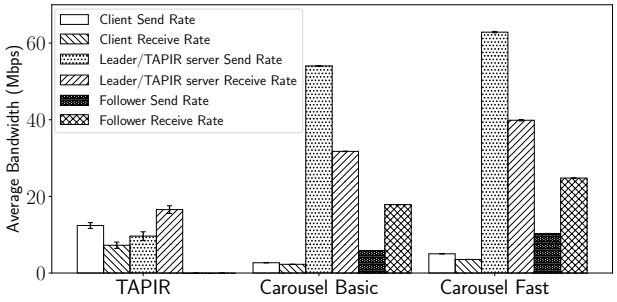

Figure 7: Bandwidth used at a target throughput of 5000 tps.

target throughput earlier than Carousel Basic, leveling off at approximately 8000 tps. This is because Carousel Fast needing to send more messages per transaction than Carousel Basic. Because the target throughput of 10000 tps required using all of our available machines, we were not able to test higher loads.

Figure 6 shows that TAPIR experiences a sharp increase in its abort rate when the target throughput is above 5000 tps, which is at the same rate when it sees a drop in its committed throughput. Figure 6 also shows that Carousel Fast's abort rate is higher than Carousel Basic's. At a target throughput of 8000 tps, Carousel Fast's and Carousel Basic's abort rate are 9% and 7%, respectively. This is due to Carousel Fast reading local replicas, which may read stale data and cause transactions to abort.

6.4.2 Network Utilization. For us to understand the network bandwidth requirement of the different systems, we measure their bandwidth usage at a target throughput of 5000 tps, which is approximately TAPIR's peak throughput. Figure 7 shows the average bandwidth usage of the three systems broken down into the send and receive rates of the clients and servers. For the two Carousel systems, we further distinguish the servers between leaders and followers. The results show that TAPIR clients require more network bandwidth than Carousel Basic and Fast clients. However, Carousel Basic and Fast servers, especially the leaders, require more network bandwidth than TAPIR servers. This is because Carousel Basic and Fast replicate both 2PC state and data to their consensus groups. As expected, Carousel Fast servers require more bandwidth than Carousel Basic servers since Carousel Fast performs both fast path and slow path concurrently.

Although both Carousel Basic and Fast require more bandwidth than TAPIR, at less than 70 Mbps, the network is not a resource bottleneck even when they are running at TAPIR's peak throughput, which is more than half of their own peak throughput. Not presented are additional results showing that network bandwidth usage increases linearly with the target throughput for both Carousel Basic and Carousel Fast.

6.5 YCSB+T Experiments

In the next set of experiments, we use the YCSB+T workload to evaluate the performance of the different systems. Similar to our previous experiments using the Retwis workload, we study the

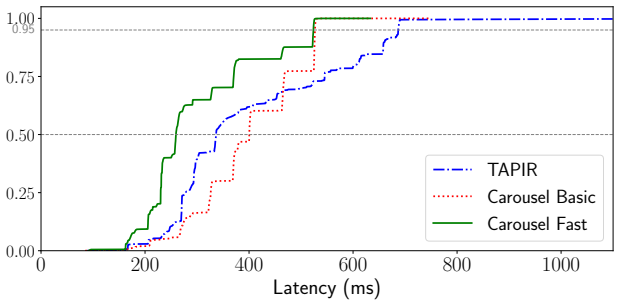

Figure 8: Latency CDF for the YCSB+T workload.

systems under a target throughput of 200 tps to focus on the performance of the system when wide-area network latencies are the dominant latency source. Figure 8 shows the CDF of the latencies for Carousel Basic, Carousel Fast, and TAPIR. Much like in the Retwis experiments, Carousel Fast has lower latencies than the other two systems across the entire distribution. This is due to its fast path that allows it to complete its Prepare phase in a single wide-area network roundtrip.

Carousel Basic's median latency when servicing the YCSB+T workload is 400 ms compared to just 290 ms in the Retwis workload. This shift in latency is mainly due to the difference in transaction types between the two workloads. YCSB+T consists of only read-modify-write transactions, whereas 50% of Retwis' transactions are read-only. Without read-only transactions, Carousel Basic does not benefit from its read-only transaction optimization, and always requires two wide-area network roundtrips to complete a transaction in the absence of failures.

TAPIR has a lower median latency than Carousel Basic because its fast path allows it to reduce its transaction completion time if local replicas are available for keys in its transaction set. This was not evident in the Retwis workload because Carousel Basic's read optimization was able to more than make up the difference. In the case where local replicas are available for all of a transaction's read set, TAPIR can complete the transaction in just one wide-area network roundtrip. However, when there is data contention and fast path execution is not possible, TAPIR must fall back to its slow path, resulting in transaction execution that requires three wide-area network roundtrips to complete. This explains TAPIR's longer tail latencies compared to those for Carousel Basic.

As can be seen from our experiments with both the Retwis and YCSB+T workloads, our Carousel Fast prototype offers significant latency reductions when compared with TAPIR. For the Retwis workload, TAPIR has a 44% higher median latency than Carousel Fast, where the latencies are 334 and 232 ms, respectively. For the YCSB+T workload, TAPIR has a 30% higher median latency than Carousel Fast (337 and 259 ms respectively).

7 CONCLUSION

Many large-scale distributed applications service global users that produce and consume data. Geographically distributed database systems, like Spanner and CockroachDB, require multiple wide-area network roundtrips to execute and commit a distributed transaction. In this paper, we introduce Carousel, a system that executes 2PC and consensus in parallel with reads and writes for 2FI transactions. Carousel's basic transaction protocol can execute and commit a transaction in at most two wide-area network roundtrips in the absence of failures.

Furthermore, Carousel introduces a prepare consensus protocol that can complete the prepare phase in one wide-area network roundtrip by parallelizing the 2PC and consensus. This enables Carousel to complete a transaction in one wide-area network roundtrip in the common case if the transaction only accesses data with replicas in the client's datacenter. Our experimental evaluation using Amazon EC2 demonstrates that in a geographically distributed environment spanning 5 regions, Carousel can achieve significantly lower latencies than TAPIR, a state-of-the-art transaction protocol.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. We would also like to thank Haibo Bian and Mikhail Kazhamiaka for their comments on Carousel's protocol design. This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and a grant from Huawei Technologies Co., Ltd. Finally, we wish to thank the Canada Foundation for Innovation and the Ontario Research Fund for funding the purchase of equipment used for this research.

REFERENCES

[1] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. 1995. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *SIGMOD*.
[2] Divy Agrawal, Amr El Abbadi, and Kenneth Salem. 2015. A Taxonomy of Partitioned Replicated Cloud-based Database Systems. *IEEE Data Eng. Bull.* 38, 1 (2015).
[3] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. 2007. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *SOSP*.
[4] 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)*.
[5] 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 *ICDE*.
[6] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX ATC*.
[7] Mike Burrows. 2006. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *OSDI*.
[8] Cockroach Labs. 2017. CockroachDB. https://github.com/cockroachdb/cockroach. (2017).
[9] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s Hosted Data Serving Platform. *VLDB* (2008).
[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*.
[11] 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 *OSDI*.
[12] CoreOS. 2017. Raft Implementation. https://github.com/coreos/etcd/tree/master. (2017).
[13] James Cowling and Barbara Liskov. 2012. Granola: Low-overhead Distributed Transaction Coordination. In *USENIX ATC*.

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*.

[15] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Rohm. 2014. YCSB+T: Benchmarking web-scale transactional databases. In *ICDEW*.

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

[17] Google. 2017. gRPC-go. https://github.com/grpc/grpc-go. (2017).

[18] Stephen Hemminger. 2005. Network Emulation with NetEm. In *Australia's 6th National Linux Conference*.

[19] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC*.

[20] Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks*.

[21] 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. *VLDB*.

[22] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*. ACM.

[23] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-data Center Consistency. In *EuroSys*.

[24] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (2010).

[25] Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998).

[26] Leslie Lamport. 2001. Paxos Made Simple. *Technical Report, Microsoft* (2001).

[27] Leslie Lamport. 2005. Generalized Consensus and Paxos. *Technical Report, Microsoft* (2005).

[28] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19 (October 2006).

[29] Leslie Lamport and Mike Massa. 2004. Cheap Paxos. *Technical Report, Microsoft* (2004).

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

[31] Barbara Liskov, Miguel Castro, Liuba Shrira, and Atul Adya. 1999. Providing Persistent Objects in Distributed Systems. In *ECOOP*.

[32] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don'T Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *SOSP*.

[33] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-latency Geo-replicated Storage. In *NSDI*.

[34] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. 2013. Low-latency Multi-datacenter Databases Using Replicated Commit. *VLDB*.

[35] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *OSDI*.

[36] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *SOSP*.

[37] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions. In *OSDI*.

[38] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *OSDI*.

[39] Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *PODC*.

[40] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *USENIX ATC*.

[41] Andrew Pavlo. 2017. What Are We Doing With Our Lives?: Nobody Cares About Our Concurrency Control Research. In *SIGMOD*.

[42] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *NSDI*.

[43] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional Storage for Geo-replicated Systems. In *SOSP*.

[44] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP*.

[45] Alexander Thomson and Daniel J. Abadi. 2010. The Case for Determinism in Database Systems. *Proc. VLDB Endowment* 3, 1-2 (2010), 70–80.

[46] 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 *SIGMOD*.

[47] UWSysLab. 2017. TAPIR Implementation. https://github.com/UWSysLab/tapir. (2017).

[48] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. *ACM Comput. Surv.* 47, 3 (2015).

[49] Robbert van Renesse, Nicolas Schiper, and Fred B. Schneider. 2015. Vive La Différence: Paxos vs. Viewstamped Replication vs. Zab. *IEEE Trans. Dependable Sec. Comput.* 12, 4 (2015).

[50] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *SOSP*.

[51] 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 *SOSP*.