Reducing the Latency of Dependent Operations in Large-Scale Geo-Distributed Systems

by

Xinan Yan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2021

© Xinan Yan 2021

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Ivan Beschastnikh Associate Professor Department of Computer Science University of British Columbia
Supervisor(s):	Bernard Wong Associate Professor David R. Cheriton School of Computer Science University of Waterloo
Internal Member:	Ali José Mashtizadeh Assistant Professor David R. Cheriton School of Computer Science University of Waterloo
Internal Member:	Kenneth Salem Professor David R. Cheriton School of Computer Science University of Waterloo
Internal-External Member:	Paul Ward Associate Professor Department of Electrical and Computer Engineering University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This dissertation includes first-authored and peer-reviewed materials that have appeared in conference proceedings published by the Association for Computing Machinery (ACM).

ACM's policy on the reuse of published materials in a dissertation is as follows:

> *"Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included."*

The following list serves as a declaration of the Versions of Record for works included in this dissertation:

Portions of Chapter 1 and 3:

Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. Carousel: Low-Latency Transaction Processing for Globally-Distributed Data. In Proceedings of the 2018 International Conference on Management of Data, SIGMOD'18, Houston, TX, USA. https://doi.org/10.1145/3183713.3196912

Portions of Chapter 1 and 4:

Xinan Yan, Linguan Yang, and Bernard Wong. 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, Barcelona, Spain. https://doi.org/10.1145/3386367.3431291

Portions of Chapter 1 and 5:

Xinan Yan, Arturo Pie Joa, Bernard Wong, Benjamin Cassell, Tyler Szepesi, Malek Naouach, and Disney Lam. SpecRPC: A General Framework for Performing Speculative Remote Procedure Calls. In Proceedings of the 19th International Middleware Conference, Middleware'18, Rennes, France. https://doi.org/10.1145/3274808.3274829

Abstract

Many applications rely on large-scale distributed systems for data management and computation. These distributed systems are complex and built from different networked services. Dependencies between these services can create a chain of dependent network I/O operations that have to be executed sequentially. This can result in high service latencies, especially when the chain consists of inter-datacenter operations.

To address the latency problem of executing dependent network I/O operations, this thesis introduces new approaches and techniques to reduce the required number of operations that have to be executed sequentially for three system types. First, it addresses the high transaction completion time in geo-distributed database systems that have data sharded and replicated across different geographical regions. For a single transaction, most existing systems sequentially execute reads, writes, 2PC, and a replication protocol because of dependencies between these parts. This thesis looks at using a more restrictive transaction model in order to break dependencies and allow different parts to execute in parallel.

Second, dependent network I/O operations also lead to high latency for performing leader-based state machine replication across a wide-area network. Fast Paxos introduces a fast path that bypasses the leader for request ordering. However, when concurrent requests arrive at replicas in different orders, the fast path may fail, and Fast Paxos has to fall back to a slow path. This thesis explores the use of network measurements to establish a global order for requests across replicas, allowing Fast Paxos to be effective for concurrent requests.

Finally, this thesis proposes a general solution to reduce the latency impact of dependent operations in distributed systems through the use of speculative execution. For many systems, domain knowledge can be used to predict an operation's result and speculatively execute subsequent operations, potentially allowing a chain of dependent operations to execute in parallel. This thesis introduces a framework that provides system-level support for performing speculative network I/O operations.

These three approaches reduce the number of sequentially performed network I/O operations in different domains. Our performance evaluation shows that they can significantly reduce the latency of critical infrastructure services, allowing these services to be used by latency-sensitive applications.

Acknowledgements

I would like to thank my supervisor, Bernard Wong, for his guidance and support throughout my Ph.D. His valuable feedback and cheerful attitudes have significantly benefited my work.

Furthermore, I extend my thanks as well to Kenneth Salem and Tim Brecht for their collaboration on Carousel. I would also like to thank my thesis external examiner, Ivan Beschastnikh, and my internal examiners, Ali José Mashtizadeh, Kenneth Salem (again), and Paul Ward, for their reviews and insightful comments.

In addition, I would like to thank all my friends and colleagues who have helped me, especially Linguan Yang, Hongbo Zhang, Xiayue Lin, Benjamin Cassell, Tyler Szepesi, and Malek Naouach, for their collaboration and support in my research projects. I am further grateful to Sajjad Rizvi for spending time with me on discussing problems and ideas. A sincere thank to Cong Guo for his support in my early days of Ph.D. His generous help makes me have a smooth transition to studying and living in Waterloo.

Finally, I would like to thank my family for their support throughout my Ph.D. It is their love and care that carry me through to the end, making this thesis possible. This thesis is dedicated to them.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Achieving low latency is critical to many distributed applications. For example, Google has published a study on the impact of latency in displaying search results to users and finds that half a second delay causes a 20% drop in its user traffic [79]. Amazon reports that for every 100 ms delay, it would lose 1% in sales[72], which is worth millions of dollars a year [55]. A study [58] also shows that a small increase in latency can have significant financial consequences for many web application providers.

However, it is challenging to reduce latency in geo-distributed systems. Large-scale distributed applications are complex and built from many networked services, where each service provides a different functionality. In serving a user request, many networked services have dependencies and have to execute in a serial order. This results in high latency, especially when the services are geo-distributed or access data in different datacenters. Microsoft reports that dependent geo-distributed services are common in enterprise applications, causing the sequential execution of multiple network I/O operations across different datacenters [100]. Likewise, Bahl et al. [12] and Natarajan et al. [87] also present complex dependencies among networked services in distributed applications. Furthermore, Facebook shows that many of their services have to access data in remote datacenters to serve user requests [8]. The sequential execution of inter-datacenter network I/O operations requires multiple wide-area network (WAN) roundtrips to complete, resulting in high request service times.

Providing additional resources, such as CPUs, memory, and network bandwidth, can reduce latency and increase throughput for applications that are resource bound. This is a good solution for these applications since the cost of adding computing resources is becoming increasingly affordable to many companies and organizations, especially with the

prevalence of cloud computing. However, simply adding resources is minimally effective at reducing the latency for a sequence of network I/O operations that have data dependencies because these dependent operations have to execute sequentially. Wide-area network propagation delay for inter-datacenter network I/O operations becomes a major source of latency, which is determined by physical distance and the speed of light.

This thesis explores different approaches to reduce the latency of performing a chain of dependent network I/O operations in three types of distributed systems. First, it addresses high transaction completion time in geo-distributed database systems that are a critical infrastructure for many applications. Such database systems shard data into partitions to achieve scalability, and rely on two-phase commit (2PC) to guarantee the atomicity of a transaction that involves multiple partitions. These systems also provide fault tolerance by replicating a data partition to several datacenters. To guarantee data consistency across replicas, they replicate both transactional data and 2PC states across datacenters before committing a transaction. When a transaction accesses partitions in different datacenters, it requires performing transaction processing (i.e., reads and writes), 2PC, and replication, where each part needs one or more wide-area network roundtrips to complete. As the three parts have dependencies, most existing systems execute them sequentially, resulting in high latency. In order to make the three different parts within a transaction execute in parallel, this thesis looks at breaking their dependencies by requiring additional transaction information that many applications can provide.

Another critical component in many applications is state machine replication (SMR), which is used to make replicas apply state changes in the same order. Most existing SMR protocols are leader-based, and they experience high latency in wide-area networks due to dependent network I/O operations. They rely on a leader to order requests and then forward these requests to replicas, requiring two network roundtrips to commit a request. To reduce the dependency on a leader and the number of required roundtrips, Fast Paxos [66] introduces a fast path, in which a client sends its request directly to every replica. However, the fast path may fail when concurrent requests arrive at replicas in different orders, e.g., due to network delays. Once the fast path fails, Fast Paxos has to perform additional coordination operations to commit the requests. To reduce the likelihood that Fast Paxos performs these dependent coordination operations for concurrent requests, this thesis explores the use of network measurements to establish a global order for requests across replicas, so that these requests can be committed via the fast path.

Besides designing protocols specifically for geo-distributed transactions and SMR, this thesis further explores a general solution for a class of distributed systems that execute a sequence of dependent network I/O operations in completing a task. Many distributed applications can leverage domain knowledge to predict an operation's result and use this

prediction to speculatively execute subsequent operations. This speculation opens up opportunities to execute these dependent operations across multiple nodes in parallel. However, it is challenging to correctly and efficiently track and update these operations' speculative states across nodes. There is no general solution and limited system-level support for implementing speculative execution in a distributed environment. Applications typically have to build their own solutions from scratch, which can be highly error-prone. To provide support for adopting speculation in distributed applications, this thesis introduces a framework for performing speculative network I/O operations.

In the next subsections, I will describe the three problems in details, providing the problem context. I will also outline the designs of my systems that address these problems.

1.1 Transaction Processing for Globally-Distributed Data

Many applications have global users generate data in different geographic datacenters and rely on geo-distributed database systems, such as Google Spanner [26] and CockroachDB [23], to manage the data. Such geo-distributed database systems shard data into partitions to achieve scalability and store each partition at a location where its data will most frequently be used. To meet applications' requirements on fault tolerance, these database systems also replicate each data partition to enough datacenters to tolerate datacenter-wide failures, e.g., due to power outages or natural disasters. Furthermore, they provide transactional support for applications to read and write multiple data records. This facilitates application development and makes it easy for developers to reason about application correctness and data consistency for concurrent read/write operations.

Although many transactions for geo-distributed database systems are designed to access data within a datacenter, it is common that applications have distributed transactions that involve multiple data partitions across geographic datacenters. For example, a flight ticket reservation system needs to access plane schedules and seat availability from airlines that operate in different regions across the world. Similarly, a social media platform has travelling users who will add someone from a different geographic region to their friend lists, and the user information is hosted in different datacenters.

To support distributed transactions that access multiple data partitions, many geo-distributed database systems, such as Spanner and CockroachDB, first perform transaction processing (i.e., reads and writes) by fetching the required data to a single site and buffering the write data. These systems will then use the 2PC protocol to ensure that transactions

are atomically committed or aborted. The write data is sent together with the first 2PC message to data servers and are applied if the transaction commits. They execute 2PC after transaction processing because the prepare phase of 2PC requires knowing the transaction's read and write keys to check conflicts with concurrent transactions.

An additional requirement for many distributed database systems is for them to remain available even in the event of a datacenter outage. To achieve this, Spanner and CockroachDB use a consensus protocol, such as Paxos [63] or Raft [91], to replicate both the updates to the database and the changes to the 2PC state machine for each transaction to servers across $2f+1$ datacenters, where f is the maximum number of simultaneous failures that the systems can tolerate.

The dependencies between transaction processing, 2PC, and replication result in the serial execution of these three parts for a multi-partition transaction. As each part requires one or more wide-area network roundtrips to complete, multi-partition geo-distributed transactions experience high transaction completion time.

To reduce the transaction completion time, this thesis introduces Carousel, a globally-distributed database system that provides low-latency transaction processing for multi-partition geo-distributed transactions. Carousel targets the same deployments as Spanner and CockroachDB, where data partitions are distributed and replicated across geographic datacenters. Much like the two systems, 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, e.g., adding friends in a social media application or redeeming points for gifts in a web application. By using the known read and write keys from this class of transactions to detect conflicts between concurrent transactions, Carousel can 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 [66] 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 to provide datacenter-wide fault tolerance, and the consensus group leader serves as the partition leader. During a transaction, instead of sending 2PC prepare requests only to the partition leaders, which would normally forward the requests to their followers, prepare requests are sent to every node in the participating partitions. Unlike state machine replication that uses Fast Paxos to order prepare requests, each member in a consensus group will independently prepare the transaction and agree on whether the transaction is prepared. Specifically, 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.

We have implemented a research prototype of Carousel and evaluated it using workloads from the Retwis [68] and YCSB+T [34] benchmarks on both an Amazon EC2 deployment and a local cluster. The experimental results show that Carousel has lower transaction completion time than TAPIR [118], a state-of-the-art distributed transaction protocol.

1.2 State Machine Replication in WANs

Similar to geo-distributed database systems, state machine replication (SMR) is a critical service in many applications, and it also has high latency in WANs due to dependencies between network I/O operations. SMR replicates application states across geo-distributed nodes, allowing applications to remain available even in the event of a region-wide system outage. A drawback of state replication is that a majority of the replicas must receive the same state update request and agree on its position in the request log before it can be committed, with most protocols relying on a leader to establish a request ordering. As a result, a distributed service that modifies its state from more than one location will typically have to wait two wide-area network roundtrips for each request: one roundtrip for a service node to send its request to the leader and receive a response, and a second roundtrip for the leader to disseminate the request to the other replicas.

Fast Paxos extends Paxos [63, 64] by introducing a fast path that can, in some cases, reduce the number of wide-area network roundtrips from two to one by having the service

nodes send their requests directly to all of the replicas. A simple approach for using Fast Paxos to implement SMR is to run a separate Fast Paxos instance for each position in the request log. Without a leader or some other mechanism to order the requests, each replica would have to independently decide which log position to accept each request. A service node knows its request will be committed if a supermajority[1] of the replicas accept the request at the same position. Unfortunately, replicas may accept concurrent requests at different log positions, forcing Fast Paxos to run a recovery protocol [66] (i.e., the slow path) to choose a request for each position that is under contention. This serial execution of the fast path first and then the slow path will cause high latency, especially in WANs.

Furthermore, Fast Paxos' fast path may have higher commit latency than a leader-based protocol depending on the geographic locations of clients and replicas. This is because a Fast Paxos client has to wait for the responses from at least a supermajority of replicas. This is more than the number of replicas (i.e., a majority) that are required for agreement in a leader-based protocol. The one network roundtrip time to a supermajority of replicas in Fast Paxos does not mean lower latency than the two network roundtrip time in a leader-based protocol. For example, when there are total three replicas, if a client is collocated with one replica in a datacenter but geographically far away from one of the other replicas, the client would experience higher latency by using Fast Paxos than Multi-Paxos [108] with the collocated replica being the leader. As clients and replicas are geographically distributed, neither Fast Paxos nor a leader-based protocol can always have the lowest latency for every client.

This thesis introduces Domino, a state machine replication protocol that uses network measurements to reduce commit latency in WANs. It supports both Fast Paxos-like consensus or leader-based consensus in different cycles of the same deployment. Clients perform periodic network latency measurements to the replicas, and the replicas also collect network latency data to each other and return those results to their clients. Each client uses their collected latency data to independently choose which consensus protocol to use for their requests. Given accurate network latency predictions, this approach allows clients to estimate the latency of both consensus protocols and use the one that has lower latency.

When clients use the Fast Paxos-like consensus in Domino, Domino leverages network measurements to establish a deterministic order for requests across replicas. To establish the request ordering, a Domino client uses its network measurement data to assign its request with a timestamp, indicating a future time when its request should have arrived at a supermajority of the replica servers. A request that arrives at a server after its timestamp

[1] A supermajority of $2f + 1$ replicas consist of at least $\lceil \frac{3}{2} f \rceil + 1$ replicas, and a common alternative is $2f + 1$ out of total $3f + 1$ replicas [66].

will not be counted, although the protocol may still choose to accept that request if enough other servers received the request before its timestamp. Domino deterministically maps a timestamp to a unique log entry. As long as requests have unique timestamps and arrive at a supermajority of servers on time, Domino can always complete its Fast Paxos-like consensus in a single roundtrip for these requests in the absence of failures.

Upon receiving a request that arrives on time, a Domino replica immediately accepts the request at the log position that is determined by the request's timestamp. Selecting any sufficiently large timestamp for a request would offer the same commit latency. However, a timestamp that is too far in the future would increase execution latency. This is because, from the current time to that future time, Domino may commit other requests that have smaller timestamps than the future time. As a SMR protocol, Domino must execute these requests in their log order, and thus Domino will only begin to execute a request after its timestamp. Although execution latency can be masked through application-level reordering, excessive execution latency should nevertheless be avoided as they can introduce user-perceptible artifacts. In a social media app, for example, a user uploads a picture to her album and then messages her friend who accesses a replica in a different region to look at the picture. It might affect user experience if the picture is not available in the region until hours later. To address unnecessarily excessive execution latency, when using Fast Paxos-like consensus, a future timestamp is chosen to represent the time when the last replica from the supermajority quorum should have received the request. Given accurately predicted arrival time at replicas, this approach ensures that the request is not rejected, while providing similar execution latency to other consensus protocols.

This approach introduces a number of challenges. It requires accurate latency predictions. We show, through extensive experiments performed on Microsoft Azure [81], that wide-area network latencies are relatively stable and can be predicted by keeping only a small history of previous network measurements. Our fine-grained timestamp-based log will also introduce many empty log entries, and it is expensive to have a dedicated replica propose no-op values for these entries. To reduce this overhead, Domino replicas optimistically accept no-ops without receiving no-op proposals once a log entry has expired.

I have implemented a research prototype of Domino. Experiments on Microsoft Azure show that Domino can have significantly lower commit latency than Multi-Paxos [108], Mencius [78], and EPaxos [84], which require a replica to forward a request to other replicas.

1.3 A Framework for Performing Speculative RPCs

Both Carousel and Domino target a specific system domain. Their approaches might not be applicable to many other distributed applications that execute a chain of dependent networked services to complete a task. This thesis further looks at a general solution based on speculative execution (SE) to addressing the latency problem in such distributed applications.

SE has been used to reduce latency in operating systems [22, 88, 89, 113, 112], Byzantine fault tolerance protocols [114, 59], and a number of other specialized applications [28, 67, 80, 110]. These systems take advantage of domain knowledge to determine when the result of an expensive operation can be accurately predicted. This predicted result can be used to speculatively execute the dependent operations, allowing them to be executed concurrently as long as the prediction is eventually shown to be correct.

Although SE is a powerful technique for reducing latency, it can introduce a significant amount of complexity to an application. Currently, adding SE support to a new production-quality application requires a significant investment in time and resources. For example, tracking dependencies between speculative and non-speculative RPCs requires state management across nodes. Also, preventing side-effects from incorrect speculation is error-prone. Therefore, previous work has only considered using SE for the most performance-critical applications, and SE has only been implemented in research prototypes or with limited amount of speculation.

This thesis introduces SpecRPC, an RPC framework for performing SE. Using the framework, an application can provide its prediction for the result of an RPC, and speculatively execute the next operation, which can be a local function or another RPC, based on the predicted result. SpecRPC aims to simplify the integration of speculative techniques in distributed applications and to support sophisticated forms of SE, allowing for the pervasive use of speculation to reduce application latency. SpecRPC facilitates development of applications that leverage speculation, but it is up to developers to predict RPC results as accurate prediction requires domain-specific knowledge.

To support SE in an RPC framework, each operation (i.e., a local function or an RPC) in SpecRPC is associated with a state that characterizes its current status. Speculatively executing an operation based on the predicted result of a pending RPC causes this operation to enter a *speculative* state, and creates a dependency between this operation and the pending RPC. An operation in a *speculative* state will transitively cause subsequent dependent operations to also be in a *speculative* state to form a dependency tree where the branching factor depends on the number of predictions made per operation. The de-

pendency tree helps SpecRPC isolate the different branches of SE, and determine which branches are still valid as actual execution results become available.

Dependency tracking and the re-execution of computation based on incorrect speculation are done automatically by the SpecRPC framework. Programmers can focus on leveraging domain knowledge to make accurate predictions for performing SE. In addition to predicting an RPC's result on the caller side, SpecRPC allows an RPC to return speculative results. This enables a programmer to take advantage of speculation when only partial information is available on the callee side of an RPC. To facilitate application development, SpecRPC provides a simple abstraction that should be familiar to any programmer who has used RPCs and callbacks.

I have implemented a prototype of SpecRPC, which can fully parallelize complex communication patterns involving multiple sequential RPC calls from the same client, or an RPC call chain where an RPC function calls another RPC function. I have evaluated the effectiveness of SpecRPC by using it to add speculative execution support to Replicated Commit (RC) [77], a distributed transaction processing protocol. The evaluation results show that SpecRPC reduces the transaction completion time of RC by 58% compared to the sequential execution of dependent operations.

1.4 Outline

Many large-scale distributed systems perform a sequence of networked services to complete a task. These services often have dependencies, requiring the sequential execution of multiple network I/O operations. This can result in high latency. This thesis explores approaches to address the high latency problem of executing dependent network I/O operations in distributed systems, especially in a wide-area network environment. It has three main contributions:

- First, it introduces techniques that leverage application-provided transactional hints to break dependencies between transaction processing, 2PC, and replication for geo-distributed transactions. This enables these different parts within a transaction to execute in parallel and reduces latency compared to many existing geo-distributed database systems.

- Second, this thesis explores the use of network measurement to reduce the required number of sequential network I/O operations in SMR. This thesis presents techniques

that can leverage recent network measurement data to establish a global order for requests across replicas in wide-area networks. This can reduce the likelihood that Fast Paxos uses additional coordination operations between replicas to commit concurrent requests.

- Third, this thesis addresses the barriers of using speculation to parallelize a sequence of network I/O operations in a class of distributed applications. It presents an RPC framework that facilitates the implementation of speculative execution in a distributed environment. The framework tracks dependencies among non-speculative and speculative operations across nodes and ensures that incorrect speculations do not affect the correctness of applications that follow its suggested design pattern.

The following chapters will describe the contributions of this thesis in detail. Chapter 2 will provide a summary of the related work that this thesis discusses. Chapter 3, 4, and 5 will present the design and evaluation results of Carousel, Domino, and SpecRPC, respectively. Finally, I will discuss about future research directions and conclude this thesis in Chapter 6.

Chapter 2

Related Work

This chapter will first describe the related work on replication techniques that are commonly used in distributed systems. Thereafter, it will summarize previous work on distributed transactions. Finally, this chapter will review the related work on speculative execution.

2.1 Replication Management

Many applications replicate their states and data in different datacenters in order to tolerate datacenter-wide failures, such as due to power outages or natural disasters. This section will describe different protocols for performing replication management.

2.1.1 General Replication Techniques

A common replication technique is to use a consensus protocol so that replicas can agree on the same sate change even under failures. One of the most widely used consensus protocols is Paxos [63, 64], which needs $2f + 1$ replicas to tolerate f simultaneous failures. In the common case, a Paxos instance uses one network roundtrip to choose one proposer's proposal and requires another network roundtrip to enforce consensus on the proposed value. Since each proposal requires at least two network roundtrips to be accepted, Paxos suffers from high latency.

To reduce the high latency in Paxos, Multi-Paxos [63, 108] adopts a long-lived leader to be the sole proposer. This eliminates Paxos' first roundtrip in the common case. In Multi-Paxos, a client sends its request only to the leader. The leader will forward the request to other replicas in the consensus group, and it will send an acceptance notification back to the client after receiving agreement from a majority of replicas. The end-to-end latency consists of two network roundtrips. Raft [91] explicitly separates the consensus process into leader election and log replication in order to improve the understandability of the protocol and to simplify the implementation.

In addition to consensus protocols, there are also replication protocols, which can provide similar guarantees to Paxos. Viewstamped Replication [74, 90] (VR) has a primary replica to order clients' requests and to forward the requests to other backup replicas. Once a majority of the replicas finish replicating the requests, the primary will notify the clients that the requests are committed, and the replication group will asynchronously commit and execute the requests. Unlike Multi-Paxos or Raft, when the primary fails, VR does not use leader election to elect a new primary replica. Instead, VR pre-defines which replica to be the new primary. Also, the new primary has to collect the log from a majority of replicas to maintain an up-to-date log, which may consume large amount of network bandwidth.

Atomic broadcast is another way to achieve replication. One implementation is Zab [51], which is designed for primary-backup replication systems. Zab has a primary replica to receive clients' requests and forward state changes to backup replicas. Zab guarantees that all of the replicas in the system will make the same state change, or none of the replicas applies a state change. Unlike Paxos that may execute uncommitted requests out of order after a leader failure, Zab will apply uncommitted state changes in the same order as the failed primary. Zab achieves this by introducing a synchronization phase after electing a new primary, which will determine the ordering of uncommitted state changes by merging the logs from at least a majority of replicas. Renesse et al. [109] provide a summary that details the differences among Zab, VR and Multi-Paxos.

2.1.2 Low-Latency Consensus Protocols

Fast Paxos [66] reduces the end-to-end latency in Multi-Paxos 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 network roundtrip, which is called a fast path. 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 like in Multi-Paxos. Furthermore,

if more than $\lfloor \frac{f}{2} \rfloor$ replicas fail, there will not be enough replicas to form a supermajority for the fast path, and Fast Paxos will only use the slow path to make progress. Another configuration of Fast Paxos is to use $3f + 1$ replicas, where the supermajority consists of $2f + 1$ replicas. This configuration allows the fast path to continue even if there are f failures.

Generalized Paxos [65] leverages the commutativity between operations to make a consensus instance accept a group of concurrent requests that do not have conflicts with each other. This approach reduces the latency of achieving consensus on non-conflicting concurrent requests. However, this approach requires applications to define the request interference before committing requests, which may not be applicable in many applications.

2.1.3 High-Throughput Consensus Protocols

A single-leader consensus protocol, like Multi-Paxos and Raft, has low throughput since all client requests have to go to the leader. Mencius [78] pre-partitions a sequence of consensus instances and assigns each partition to a replica in a consensus group, where the replica will be the leader for the corresponding consensus instances. Each replica will accept clients' requests to its pre-defined consensus instances, but it cannot commit an instance until all previous instances are committed. As replicas may receive requests in different rates, a replica can choose to skip its instances by using a no-op command if it does not receive any request. Compared with single-leader consensus protocols, such as Multi-Paxos and Raft, Mencius provides load balance among servers in a consensus group, which increases its throughput.

EPaxos [84] also allows a client to submit its request to any replica in a consensus group, but it dynamically orders clients' requests. After a replica (acting as a coordinator) proposes its received request to other replicas, each replica will reply the dependency between the request and other concurrent requests. If the request does not have conflicts with other requests at a quorum of replicas, the coordinator will commit the request. Otherwise, the coordinator will combine its received dependency information and replicate them to at least a majority of the replicas. Once the replication completes, the request is committed. To execute the request, the coordinator will use the dependency information to build a dependency graph, and it will not execute a request until it has executed previous requests in the graph. For cycles in the graph, the coordinator will deterministically sort the requests based on their sequence numbers, where EPaxos assigns each request with a unique sequence number. Since each server has the partial orders from a majority of replicas, servers will have the same dependency graph and the same order of execution.

One limitation of this approach is the requirement of defining conflicts between requests in advance, which may not be applicable for many applications.

Similar to EPaxos, ATLAS [35] orders concurrent requests by tracking their dependencies across replicas. It also has a fast path to commit a request by requiring the agreement from a fast quorum of replicas. The fast quorum requires only a majority of replicas compared to EPaxos that typically requires more than a majority. By contacting fewer replicas to commit a request, ATLAS can have lower latency than EPaxos, especially in a geo-distributed deployment. To achieve this, however, ATLAS tolerates fewer simultaneous replica failures than EPaxos.

CAESAR [10] is a multi-leader Generalized Consensus protocol for wide-area networks. To establish a request ordering, CAESAR requires a quorum of nodes to agree on the delivery timestamp of an operation. However, if a node receives concurrent conflicting operations out of timestamp order, it has to wait until it finalizes the decisions of larger timestamp operations, which results in higher latency.

2.1.4 Network-Aware Consensus Protocols

There are also systems that use dedicated network hardware to improve the performance of achieving consensus. SpecPaxos [94] makes a client send requests to every replica in a consensus group, and each replica speculatively executes the request and replies the result to the client. Same as Fast Paxos, if a client receives the same result from a supermajority of replicas, the client considers its request committed. The replicas will periodically synchronize with each other to commit the speculative execution results. SpecPaxos uses software-defined network to implement a network-level multicast mechanism that ensures multiple receivers can receive multicast messages from different senders in the same order in most cases. As a result, SpecPaxos can commit most of requests just in one network roundtrip in the common case. However, if there is inconsistency between replicas' states, e.g., because of message lost or reordering, all replicas have to stop processing new requests and start a reconciliation protocol to solve the inconsistency, where the leader will collect all replicas' logs and merge them into one log.

NOPaxos [70] also uses software-defined networking to order multicast messages. NOPaxos implements a centralized network sequencer for each consensus group, and a client's request sent to the group will be assigned a sequence number by the sequencer. With the sequence number, each replica can order its received requests. Upon receiving a client's request, each replica will reply its order information about the request to the client. Each consensus group in NOPaxos has a leader. If the client receives the same

order information from a majority of the replicas including the leader, it considers the request as committed.

NetPaxos [31, 30] proposes to implement Paxos in a software-defined network with P4 switches. It also introduces an architecture to achieve agreement in Fast Paxos by requiring network messages to arrive at replicas in the same order. NetPaxos, NOPaxos, and SpecPaxos rely on software-defined networking and specialized network switches to order clients' requests packets, which may not be applicable to applications in a wide-area network environment.

2.2 Distributed Transactions

Geo-distributed database systems shard data into partitions in order to achieve scalability. To support distributed transactions that involve multiple partitions, many storage systems (e.g., Megastore [15], Spanner [26], and CockroachDB [23]) use a concurrency control mechanism, like two-phase locking (2PL) or optimistic concurrency control (OCC), to provide transaction isolation on each partition and apply two-phase commit (2PC) to guarantee a transaction's atomicity across partitions. To provide fault tolerance, these systems also use a consensus protocol to replicate transactional states and data.

One design is to layer the transaction management on top of a consensus protocol, where the system will sequentially execute different parts in a transaction [15, 26, 23]. In Spanner, for example, a client first reads data from the leader of each participant partition's consensus group. To commit the transaction, the client initiates 2PC among the leaders. Each leader prepares the transaction by using 2PL to detect conflicts with concurrent transactions. The leader will replicate its prepare result to its consensus group before exposing the result outside of the consensus group. This approach facilitates reasoning about the system's correctness and allows for a relatively straightforward implementation. However, it 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.

2.2.1 Improving Concurrency Control

Calvin [106] introduces a distributed transaction scheduler that can deterministically order all of the input transactions, and each data server will execute the transactions in the

same order. This deterministic ordering reduces the coordination overhead for conflicting transactions, so that it can significantly increase the system's throughput. However, Calvin still sequentially executes transaction management and replication, which requires multiple wide-area network roundtrips to complete a transaction in a geo-distributed environment. Furthermore, unlike Spanner, Calvin does not support transactions that require interactivity between clients and servers, which are common in practice [92].

Rococo [85] also orders transactions before executing them in order to increase the system's throughput and avoid aborting transactions. Rococo makes a server track the dependency among concurrent transactions. Before executing a transaction, a server will first send the dependency information to the transaction coordinator. With the dependency information from all of the participant servers in a transaction, the coordinator will determine the order of executing conflicting transactions. The coordinator will send its ordering result to each server, and the server will execute the transaction. To tolerate failures, Rococo uses Paxos to replicate the transactional data and states on the coordinator and data servers.

CLOCC [5, 73] moves transaction execution to the client side by using caches. CLOCC caches data on the client side and makes servers track which data set each client has cached. A client executes a transaction locally by using the cached data, and it initiates 2PC to commit the transaction. If the transaction can be committed, CLOCC will invalidate other clients' cache on the modified data, which is very expensive with many clients. Although CLOCC uses client cache to reduce number of messages required for transaction execution, this increases the storage cost, and a client's cache might need to be very large to obtain a good performance for some workloads.

Lynx [119] constructs a transaction as a chain of tasks across participant servers, where each hop in the chain executes one task as a local transaction. Although the chain executes sequentially to complete the whole transaction, but the client can receive a commit or abort notification after the first hop. If the first hop commits, all of the other hops will commit the transaction. If a hop cannot commit the transaction because of conflicts, it will retry until the transaction is committed. Lynx statically analyzes a workload to determine which transactions can be constructed as a chain by using transaction chopping techniques [99]. If a transaction cannot be chopped into a chain, Lynx will execute the transaction as a distributed transaction via 2PC.

Calvin, Rococo, CLOCC, and Lynx can increase throughput and reduce transaction completion time by introducing different concurrency control mechanisms for distributed transactions, but these systems still need to sequentially perform transaction and replication management, resulting in high latency.

2.2.2 Performing 2PC within A Datacenter

Restricting 2PC to involve only nodes within a datacenter can reduce the number of wide-area network roundtrips that are required to commit a transaction, which will reduce the transaction completion time. One such an approach is Replicated Commit [77], which builds Paxos on top of 2PC and can commit a transaction in one wide-area network roundtrip. 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. Furthermore, since Replicated Commit uses single-decree Paxos to commit each transaction, one replica failure in a consensus group may cause that the remaining replicas cannot determine the outcome of a pending transaction. This will lead to aborting any future transactions that have conflicts with the pending transaction.

Consus [36] makes a client first execute a transaction in one datacenter. The datacenter will also forward the transaction to other datacenters, of which each will independently re-execute the transaction. Each datacenter will broadcast its execution result (i.e., commit or abort) to others. Finally, Consus uses Generalized Paxos [65] to make each datacenter to learn the final decision on the transaction. This approach requires three one-way messages across datacenters to complete a transaction in the common case. Compared with Replicated Commit, Consus avoids quorum reads during the execution of a transaction.

Both Consus and Replicated Commit require fully replicating all data in every datacenter in order to limit 2PC messages within a 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 [119].

Microsoft's Cloud SQL Server [16] avoids 2PC by forcing that a transaction can only access the data on one server. This approach is not scalable for all applications as the data size increases. Furthermore, this limitation requires a careful design for the data partitions, and the partition scheme may be invalid in the future as applications add new data schemes.

2.2.3 Parallelizing 2PC and Replication

Another approach to reduce the latency of committing a transaction is to merge transaction management with replication management. Instead of using a consensus protocol to replicate a transaction's prepare result (e.g., as done by Spanner), MDCC [60] uses Paxos instances per data record to accept or reject a write operation on the record. While executing the 2PC prepare phase, if all of the writes in a transaction are accepted by

the corresponding Paxos instances, the transaction coordinator determines to commit the transaction. MDCC uses Fast Paxos [66] to complete the prepare phase and the replication in one wide-area network roundtrip. It also leverages Generalized Paxos [65] to accept concurrent transactions' writes on the same key if the writes are commutative with each other. As MDCC runs a Paxos instance per update in a transaction, it requires more CPU cycles to complete a transaction compared to using a Paxos instance per data partition.

TAPIR [118] proposes an inconsistent replication protocol and resolves consistency issues among replicas in its transaction management systems. TAPIR makes a client send its transaction prepare request to every replica of a partition. Each replica independently prepares the transaction and sends its result to the client. If a supermajority of the replicas prepare the transaction, the client considers that the transaction is prepared on the partition. If a supermajority of agreement cannot be achieved, the client will force the replicas to agree on the same prepare result, which requires additional wide-area network roundtrips.

Both TAPIR and MDCC parallelize the execution of 2PC and replication and can commit a transaction in one wide-area network roundtrip in the common case. However, for concurrent transactions that have conflicts, TAPIR and MDCC have to fall back to a slow path to order the transactions, which may increase the tail latency. In this case, both TAPIR and MDCC require three or more wide-area network roundtrips to complete a transaction when data replicas are not available in the client's datacenter.

2.2.4 Limiting Transaction Expressiveness

Limiting the expressiveness of transactions is another method for reducing transaction completion time. Janus [86] targets one-shot transactions [52] 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 wide-area network roundtrip. Janus extends Rococo's concurrency control mechanism to make every replica of a data partition track the execution dependency of concurrent transactions that have conflicts. A client sends a transaction to every replica of each participant partition. Before executing a transaction, each replica sends the transaction's dependency information to the transaction coordinator. For each participant partition, if the coordinator receives the same dependency information from all of the corresponding replicas, the coordinator will notify every replica to execute the transaction following the current dependency. Once every partition has a replica finish executing the transaction, the coordinator will send a commit response to the client.

In Janus, if replicas have different dependencies for conflicting transactions, the coordinator will order the transactions and force all of the replicas to execute the transactions according to the new order. Although this re-ordering approach can avoid aborting transactions and increase throughput, it may require as many as three wide-area network roundtrips to commit the conflicting transactions.

Granola [27] introduces a special type of one-shot transactions, called independent transactions. For an independent transaction, each participant server can independently execute its part of the transaction, but all of the servers can come to the same decision to commit or abort the transaction. Granola makes servers run in two modes, timestamp mode and locking mode. A server runs in the locking mode when it processes general distributed transactions that require locking and 2PC. Otherwise, the server runs in the timestamp mode to process independent transactions. While receiving an independent transaction request, each server sends other participant servers a vote indicating whether it is in locking mode. A vote from a locking-mode server will cause every server to abort the transaction. Granola avoids 2PC to commit independent transactions, which reduces transaction completion time. However, Granola does not target geo-distributed settings, and it still sequentially executes transaction and replication management.

Sinfonia's mini-transactions [6] 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 mini-transactions and one-shot transactions prevent clients from interactively performing read and write operations to servers, which is commonly used in practice [92], especially in rapid development [15].

2.2.5 Relaxing Isolation Level

An alternative approach to achieve low latency in a distributed storage system is to reduce transactions' isolation level or adopt weak consistency. For example, Walter [101] is a geo-replicated key-value store that provides parallel snapshot isolation (PSI) among transactions. PSI guarantees that transactions within a site see a consistent snapshot of the data, and concurrent transactions cannot update the same data. Walter achieves PSI by allowing reads in local sites but forcing writes on a data record to go to the same site, called a preferred site. A transaction can be committed locally without coordinating with remote sites, if the local site is preferred by all of the writes. Otherwise, Walter uses 2PC to coordinate remote preferred sites to commit the transaction.

Instead of supporting a general read-write transactions, some distributed storage systems use a weak consistency model among replicas to provide high availability and low

latency for a single operation (e.g., a read or a write) on a single object. Bayou [104] provides eventual consistency by allowing a client to read and write to any replica. The client can get a response immediately without any coordination among replicas. Asynchronously, Bayou will propagate writes among all replicas, and Bayou uses a primary node to order the writes. Although a Bayou server can independently execute writes locally, the writes can be undone and re-executed, and they will not be finalized until they are executed following the primary's order. The client can query a Bayou server to know if its writes are finalized. Also, Bayou requires applications to specify how to detect and resolve write conflicts since applications have different semantics for conflicts. This increases the complexity of application development.

Dynamo [33] uses a quorum-like protocol to provide eventual consistency among replicas, in which the write quorum can be less than a majority of the replicas in order to achieve low latency. A client's request can go to any replica which will serve as a coordinator. The coordinator will forward the request to other replicas. For a write operation, the coordinator will assign it with a version number that is constructed by using vector clocks [62]. Upon receiving the write request from the coordinator, a replica will create a new data copy with the write version and respond to the coordinator. If the coordinator receives a response from a write quorum of replicas (including the coordinator itself), the write is successful. For a read operation, each replica returns its latest version of the data to the coordinator. After getting the reply from a read quorum of replicas, the coordinator returns all the versions it receives to the client.

PNUTS [24] proposes per-record timeline consistency, where all of the replicas execute writes in the same order for each data record. To achieve this, PNUTS apply a single master replica for every data record. All writes to a data record will be forwarded to the master. The master orders the writes and propagates them to other replicas that will execute the write operations in the same order. PNUTS allows a client to read from any replica, which may return stale data.

Facebook's Cassandra [61] uses a quorum-based protocol to perform read and write operations among replicas. Cassandra allows a client to read from any, a quorum, or all of replicas, depending the client's preferred consistency guarantee. Facebook's TAO [19] also provides eventual consistency. TAO forwards all writes to a primary replica but allows clients to read from any backup replica or a cache server.

COPS [75] provides causal consistency and converges concurrent writes that have conflicts in the same order across replicas. COPS allows a client to send a request to any replica, and a replica responses immediately after executing the request locally. Within one replica, COPS makes all of the operations be linearizable. For a write operation, the

replica will assign it with a timestamp and asynchronously propagate it to other replicas. Each replica will order writes based on the timestamp, so that concurrent writes on the same key are applied in the same order on every replica. An extension [76] of COPS can further support causal consistency for read-only and write-only transactions.

Although a storage system can provide low latency by using a low isolation level or weak consistency, applications that require serializability among transactions have to build their own application-level solutions, which is error-prone and can introduce additional delays to complete a transaction.

2.2.6 Avoiding Transaction Abort

When a transaction is aborted due to contention, such as read-write or write-write conflicts with concurrent transactions, retrying the transaction will introduce additional latency to commit the transaction. Avoiding transaction abort will prevent the latency due to retrying a transaction, and it will also increase the system's throughput.

RAMP [14] leverages multi-version to allow concurrent write operations not to block each other, where each write creates a new version of a data item. This avoids aborting write-only transactions due to write-write conflicts. RAMP also makes each write carry all the write keys in a transaction, so that a concurrent read-only transaction can be aware of potential read-write conflicts. If there is a read-write conflict, the read-only transaction will fetch the uncommitted but prepared write data instead of being aborted or blocked. RAMP can be extended to support read-write transactions, but it can not prevent anomalies caused by concurrent updates, such as two transactions increasing the same count number. RAMP only provides read atomic isolation, where none or all of the updates in a transaction is visible to other transactions.

ALOHA-KV [39] introduces an epoch-based concurrency control (ECC) mechanism to avoid transaction abort for read-only and write-only transactions. It avoids read-write conflicts by isolating the execution of read-only and write-only transactions into separate epochs. Like RAMP, it also prevents write-write conflicts by making each write create a new version of data. ALOHA-DB [38] extends ALOHA-KV to support read-write transactions by converting a transaction into a set of functors. ALOHA-DB uses one epoch to order transactions, and it executes the corresponding functors in the next epoch in the same order.

Although ECC avoids aborting transactions due to contention, it introduces additional latencies to the normal execution of a transaction even if there is no contention. This is because a transaction has to block until the system grants the corresponding epoch.

Furthermore, both ALOHA-KV and ALOHA-DB rely on a centralized epoch manager to grant epochs to every data server, and the whole system blocks during an epoch switch. This epoch switch mechanism is not scalable as the number of servers increases. It will also introduce significant latency in a geo-distributed setting.

As described in previous sections, Calvin [106] orders transactions through a distributed sequencer to avoid transaction abort. Rococo [85] and Janus [86] track the dependency among concurrent transactions and execute dependent transactions in the same order. Like ALOHA-DB, Calvin, Rococo, and Janus avoid transaction abort by deferring the processing of transactions and forcing transactions to execute in the same order on different partitions. This method can also increase throughput, but it introduces additional latency to every transaction, even if the transaction does not have conflicts with any other concurrent transactions.

Another approach to reduce transaction abort is to leverage applications' domain knowledge. Bailis et al. [13] propose a formal framework, invariant confluence, which can execute concurrent transactions without coordination by allowing an application to define data constraints. Invariant confluence executes transactions on different views of a database state, and it will merge the transactions' outcome. The system has to apply coordination for transactions that will invalid the application's constraints or cause diverge in database states. Invariant confluence can reduce transaction abort rate by avoiding the unnecessary conflicts detection according to applications' semantics. However, this approach requires developers to carefully examine the behavior of each type of transactions, and the developers have to re-define the data constraints as data scheme changes.

2.3 Speculation

Speculation [88, 22] is a latency-hiding technique that enables parallel execution of dependent operations. By predicting the result of an operation, an application can speculatively continue its execution that depends on the completion of the operation. If the prediction is correct, the executions of the dependent operations are overlapped, which reduces the overall execution time. Otherwise, the application discards its speculative execution and re-executes from where the incorrect speculation starts, which is equivalent to the sequential execution of the dependent operations.

Compared to the sequential execution, speculation requires additional computing resources. However, as computing resources become increasingly affordable, especially under the prevalence of cloud computing, many companies and organizations are able to provide additional computing resources for improving their systems' performance. Leveraging

speculation will facilitate applications to reduce latencies in cases of otherwise unavoidable or lengthy wait times for an operation to complete. This section will summarize previous works on using speculation to reduce applications' execution time.

2.3.1 OS-Level Speculation

There have been several systems that support speculative execution at the operating system (OS) level. Speculator [88] modifies the Linux kernel to allow a process to speculatively execute subsequent operations instead of blocking on a system call. To perform speculative execution, Speculator predicts the result of a system call, checkpoints the process's states, and allows the process to speculatively continue. If the prediction is correct, Speculator will commit its computation during speculative execution. Otherwise, it will rollback to the checkpoint to allow the process to re-execute, which is transparent to the user-level applications. Speculator also supports speculative execution across processes by propagating speculative states through inter-process communication (IPC). It has been used to reduce the latency of a distributed file system by allowing applications that would normally block during a network I/O operation to continue execution using a predicted result. The execution of the network I/O operations is overlapped with local computations, which will reduce the total execution time if speculation is correct.

Nightingale et al. [89] further use Speculator to implement a model for local file I/O operations, which provides the same durability guarantee as synchronous I/O but can achieve a low latency that is close to asynchronous I/O. AutoBash [102] also uses Speculator to facilitate system administrators and users to manage OS configurations.

To avoid side effects, Speculator buffers any external output during speculative execution until the speculation is committed. Since Speculator considers network I/O as external output, it can only allow speculative execution on the local machine, and it is unable to propagate speculation across machines in a distributed system. If an application has two dependent network I/O operations across different servers, these two operations still have to execute sequentially, which incurs high latency.

Furthermore, Speculator starts speculative execution through system calls. As system calls' interfaces are well defined, Speculator requires minimal changes to applications. However, system calls only have a final return, and they cannot return intermediate computation as a prediction. This does not fit applications that are unable to make a highly precise prediction until they have completed some (preliminary) work of the action that they want to predict. In this case, allowing using the intermediate computation result of a

predicted operation to start a speculative execution can increase the probability of correct speculation, which will effectively reduce latencies.

Also, because Speculator provides speculation as part of the operating system, it must checkpoint the state of the entire process in order to undo changes when speculative execution occurs on an incorrect prediction. As the whole process will rollback to a checkpoint, Speculator will waste the computation that is independent from the speculative execution in a multi-threaded process.

A recent study [112] proposes to use speculation to reduce the latency in OSes for devices that can complete I/O operations in microseconds. By analyzing pure I/O applications, intensive I/O applications, and computation intensive I/O applications, it finds that speculation is a promising technique to hide delay for low-latency storage I/O operations. The survey shows that the cost of checkpointing the states of the entire process is not trivial at the microsecond level. It also proposes hardware checkpointing and checkpoint-free speculation to provide a more efficient way of performing speculation.

OS-level speculation support is a generic approach, and it requires minimal modification to applications. However, it limits the power of speculative execution on latency reduction because it lacks the application's semantics [113]. Without knowing the behaviors of an application, it is difficult for an OS to determine which part of the applications is predictable. Also, the domain knowledge of an application usually provides more opportunities for speculative execution. For example, a storage system that is based on replicated state machines can use the result of the first responding replica as a prediction for the final result. It is not straightforward for the OS-level speculation to take advantage of such predictions. Furthermore, it is difficult for OS-level speculation support to propagate speculative states across machines. Without application-level semantics, an OS can not know if a network packet would cause side effects that could not be recovered on a different machine.

2.3.2 Application-Level Speculation

Compared with OS-level speculation support, allowing developers to define the scope of speculative execution at the application level can better utilize application semantics to make predictions and increase parallelism. For example, an application may have a sequence of user-level operations that can be executed in parallel by using speculation. This kind of application semantics is agnostic to the OS, but its latency may be significantly reduced by using speculation.

To provide application-level speculation support, Wester et al. [113] extend Speculator to allow applications to define customized speculation policies. The custom speculation policies are provided as specific system calls to developers. The developers can use the system calls to explicitly specify where speculative executions should start and end in an application. This allows developers to leverage application-level domain knowledge to make predictions for speculative execution in order to increase the parallelism of the system. However, as the implementation is based on Speculator which considers network I/O as external output, it cannot support propagating speculative states across machines to overlap the execution of dependent network I/O operations in a distributed system.

SpecHint [22] is a binary modification tool that can transform applications to use speculative execution to prefetch data from disks to memory in order to reduce the applications' total execution time. When an application performs a read I/O request and waits for the read result, SpecHint will create a thread that can speculatively continue the application's execution. Instead of actually performing read I/O operations, the speculative execution only generates hints for future disk read I/O operations. SpecHint uses these hints to prefetch data from disks into memory, which will reduce the application's data loading time in the actual execution. Although SpecHint does not need to modify applications' source code, SpecHint is limited to single-thread applications.

Fraser and Chang [40] further extend the user-level SpecHint [22] to implement kernel-level speculative execution for prefetching read data. This kernel-level SpecHint avoids modifying applications' binary code by dynamically detecting read I/O operations through system calls. It can also better handle memory swapping and limit the amount of resources required for speculation. However, both versions of SpecHint only reduce the latency for local read I/O operations. Furthermore, their speculative execution only aim to generate hints for prefetching read data. Any computation that is done through the speculation is discarded, and the normal non-speculative execution has to redo the computation.

To better leverage application's semantics, Fast Track [54] is a speculation system that allows developers to implement, or the compiler to generate, two versions of code for the same sequential tasks, fast track and slow track. The fast track is unsafe but optimized, and the slow track is safe and sequential code. In a multi-processor architecture, both the fast track and the slow track can execute in parallel. Given a set of sequential tasks, the output of one task on the fast track can be used to speculatively start the next task on the slow track, which will reduce the total execution time of these tasks. The actual slow track output is used to determine if the fast track output is correct in order to guarantee the correctness of the program. Fast Track does not allow speculative execution out of its own running process, such as propagating speculative states to another process via IPC.

Similar to Fast Track, Prospect [103] can also generate programs that will concurrently run a fast version and a slow version for the same sequential tasks. Prospect further supports speculatively performing system calls. Both Fast Track and Prospect allow developers to take advantage of application semantics to implement speculation, which is more flexible than just using the OS-level speculation support, such as in [88, 89].

Another application-level usage of speculation is to implement deterministic replay systems which are critical to system debugging. Respec [69] speculatively executes and logs a process between checkpoints. Respec will reply the log using another process. If the execution output diverges between the two processes, Respec will rollback the original process and redo the replying execution. To reduce the synchronization overhead between the original and replying processes, DoublePlay [111] introduces epochs during process execution. Within an epoch, DoublePlay forces threads running on different CPU cores in the original process to execute sequentially on one CPU in the replying process. As epochs access different copies of memory, DoublePlay parallelizes the execution of multiple epochs when replying. However, DoublePlay requires doubled number of CPU cores than Respec. Frost [110] extends DoublePlay to detect data-race bugs by changing the execution order of threads within an epoch in multiple replying processes.

All aforementioned application-level speculation techniques limits speculative execution on a single machine. They cannot propagate speculative states among different servers in a distributed environment. As a result, they do not directly support using speculation to overlap the execution of network I/O operations.

2.3.3 Thread-Level Speculation

Thread-level speculation is a parallelization technique that can shard a sequential program into multiple threads at compile-time and speculatively execute the threads in parallel on a multi-core and multi-processor architecture in order to reduce the total execution time.

LRPD [96] groups different iterations in a loop as multiple threads that will speculatively run in parallel. LRPD uses run-time tests to determine if the threads are safe to execute in parallel. R-LRPD [29] extends LRPD by using reduction parallelization [117]. R-LRPD analyzes the potential data dependence among iterations in a loop and schedule the iterations to speculatively execute in a way that possible non-dependent iterations run in parallel.

Privateer [50] provides thread-level speculation support for programs that use dynamic memory allocation. To achieve this, Privateer not only statically analyzes data dependence at compile-time but also uses profiling information to characterize the program's memory

access patterns. Another speculation possibility for a sequential program is to predict the execution control flow. Bhowmik and Franklin [18] use profiling information to predict the execution path of a sequential program, and each branch can execute speculatively as a thread. There are also other similar frameworks that provide automatic thread-level speculation through compiling, which are summarized in a survey [116].

The compiler-driven approach cannot extract all of the potential speculation opportunities in a program because the source code does not contain high-level domain knowledge which can facilitate performing more speculative executions. Prabhu and Olukotun [95] propose to manually change source code in order to allow more speculation to reduce a program's execution time.

Thread-level speculation techniques makes a single-thread program execute as multiple threads via speculation, which will increase the parallelism in order to reduce the program's execution time. However, most existing thread-level speculation techniques focus on numeric computation on a single machine, and they do not handle I/O latencies.

2.3.4 Speculation for Replicated State Machines

Wester et al. [114] propose to use speculation to tolerate latency in systems that use replicated state machines [97]. Without speculation, after issuing a request (e.g., a read or write operation) to a replicated state machine based system, a client has to wait for the reply from enough replicas, such as a majority, to know the final result before it can continue further execution. By using speculation, a client can use the reply from the first responding replica as a prediction for the final result in order to speculatively continue its execution. Leveraging this client-side speculation, Wester et al. [114] introduces PBFT-CS, which can reduce the latency in the replicated NFS that use the Practical Byzantine Fault Tolerance (PBFT) protocol [21].

In contrast to PBFT-CS, Zyzzyva [59] uses server-side speculation to reduce the latency in Byzantine Fault Tolerance design. Zyzzyva allows replicas to speculatively execute a client's request and respond to the client. If the client sees the same execution results from sufficient replicas, the client considers that the request has been committed, and the servers will guarantee the commit of the request even there are faulty servers. However, the speculation support in either PBFT-CS or Zyzzyva is application specific. They do not provide a general framework to facilitate developers to implement speculative execution on both client and server sides in a distributed system. Furthermore, none of the two works support propagating speculative states across multiple servers to overlap the execution of dependent network I/O operations.

Kim et al. [56] improve the performance of Byzantine Fault Tolerance replication systems by allowing the execution phase to concurrently process requests. This work uses operating system transactions [93] to allow multiple requests speculatively execute in parallel on a server. If the actual execution order of the requests is different from the pre-defined order, the execution of the requests are aborted as transaction abort. Although this work supports server-side speculative execution, the speculation is still limited to a single node. Furthermore, OS transactions mainly provide ACID for system states, and they cannot manage or rollback user states in a multi-threaded process. This limits the use of OS transactions to implement speculative execution in a general application.

Correctables [44] also targets replicated-object storage systems and provides incremental consistency guarantees for user applications. Correctables uses the return value from the first responding replica as a preliminary result (weak consistency) for applications. Once the final result (strong consistency) is available, Correctables will also return it to applications. Correctables can also be configured to return any intermediary views on the result between the preliminary and the final results. By continuously receiving different views on the result from weak consistency to strong consistency, applications can either speculatively continue their executions on each view or refine their following operations' results. In the former case, Correctables can reduce the application-level latency when the speculation is correct. In the latter case, Correctables may improve the interactivity of applications. In addition, some applications may use different consistent levels on the same operation as the system's state changes. For instance, weak consistency might be sufficient for a ticket selling system when the stock of tickets is above a threshold. When the stock is below the threshold, the system may require strong consistency.

By providing such incremental consistency guarantees, Correctables cost more network bandwidth and trades off throughput for low latency. Although Correctables provides an abstraction to decouple the application layer and the storage layer, Correctables leave the handling of incorrect speculations to the applications, which is very error prone and may increase the complexity of the application development. For example, some application may benefit from using speculative execution of a sequence of dependent operations. Using Correctables, the application still has to track the dependency between speculative and non-speculative execution by itself, which introduces a lot of complexity in the application's development. Furthermore, Correctables only targets replicated-object storage systems, and it does not benefit other applications that can use speculation techniques to reduce latency.

2.3.5 Speculation in Other Distributed Applications

Speculation has also been used by previous works to reduce latencies in distributed applications. Lange et al. [67] use speculation in remote display systems (i.e., VNC and Microsoft remote desktop protocol) to hide the network latency from end users in order to improve user experience, especially in a wide-area or wireless network environment. This work predicts screen update events by analyzing the history of user actions and screen events, and it speculatively updates the screen on the client side before the actual response arrives from the server side. Unlike many other speculation systems that prevent external output during speculative execution, this work allows a speculative screen update to be displayed to users. If there is an incorrect prediction, the system will correct the screen displays based on the actual responses from servers. This speculative remote display system only supports speculative execution on the client side. The implementation cannot be directly applied to other applications that prefer speculative execution on the server side.

Crom [80] is a JavaScript framework that allows web applications to speculatively prefetch web page content in order to reduce the page loading time. While an user is browsing a web page, a web application can predict the next event, such as clicking a button, and it will speculatively trigger the event in order to prefetch data from servers. If the event actually happens, the data will be available locally on the client side, and the data can be directly used to compute the page layout and rendering. Although Crom simplifies the work for developers to implement speculative execution, Crom limits the speculation support on the client side for web applications. Corm can not reduce the latency of data generation on the server side, which usually involves many network I/O operations in a large-scale application.

Time Warp [49] uses speculation to reduce the execution time for discrete event simulations. It allows each process in a simulator to speculatively proceed by using the messages in the process's current input queue. If a new message arrives and should be handled before some messages in the queue, Time Warp will rollback the process to the state when the new message should be the next message to consume. Time Warp only works for applications that use a global virtual time to implement synchronization. It cannot either directly provide speculation support for distributed systems, where messages can be out of order.

Remus [28] leverages speculative execution to provide high availability for virtual machines running on commodity hardware. Instead of blocking the primary's execution and synchronously replicating the primary's states to the secondary, Remus uses speculation to isolate the primary's system execution and the network I/O that is required to propagate the primary's states to the secondary. Remus makes the primary to speculatively execute and buffers any output that will be exposed to clients. Concurrently, Remus will check-

point the system states and asynchronously replicates the states to the secondary. Once the replication completes, Remus commits the speculative execution before the checkpoints and releases the buffered output to the clients. Remus's speculative execution is also limited on a single node.

Another usage of speculation is to prevent inconsistencies in distributed systems at runtime. CrystalBall [115] uses speculation to implement an immediate safety check for possible inconsistent states on a node in a deployed distributed system. Before a node actually makes state change based on input events, such as messages from other nodes, the node first speculatively processes the input events. If the speculative execution will result in an inconsistent state, the node will not make the actual state change. Instead, CrystalBall proposes that the system should be designed to automatically change its execution to avoid the predicted inconsistent states. CrystalBall only uses speculation as a pre-processing mechanism for safety check on a single node. Its speculative execution support does not deal with I/O latencies in distributed systems.

DEFINED [71] uses speculative execution to implement deterministic execution for network debugging. DEFINED makes every node in a network independently order the received messages via a deterministic function, so that each node will process the messages in a deterministic order. As a result, the system can repeat an execution instance deterministically in order to facilitate the debugging process. Instead of using the high-latency stop-and-wait approach to order the arrived messages, each node assumes that the messages follow the expected order, and it speculatively processes the messages, such as delivering the messages to application-level software. If the predicted order is incorrect, the node will rollback to the point where the order diverge first happens, and it will re-process the messages. This rollback procedure is also propagated to the nodes that have speculatively executed based on the incorrect order.

DEFINED supports the propagation of speculative execution across different nodes, and it can make all of the affected nodes rollback from an incorrect speculation. However, the implementation of the speculation support is specific to network message ordering. Also, the rollback mechanism on each node is similar to Speculator, which will recover the whole process. This may waste the computation that is independent from the speculation. Furthermore, DEFINED also checkpoints the entire process, and it could not leverage application-level semantics to facilitate speculation.

Chapter 3

Carousel

Geographically distributed database systems are becoming a key infrastructure for many applications. Such a system shards data into partitions to achieve scalability, and uses replication to provide fault tolerance. Executing a distributed transaction typically involves read and write operations, 2PC, and replication. When a transaction needs to access data in different datacenters, sequentially executing different parts in the transaction processing will result in a serial execution of network I/O operations across datacenters. For example, Google Spanner requires up to 4.5 wide-area network roundtrips to execute and commit a transaction, which will incur high latency.

In this chapter, I will present Carousel, a transaction processing system that can execute and commit a transaction within two wide-area network roundtrips in the absence of failures. By breaking the dependency among read operations, 2PC, and replication, Carousel executes the different parts in parallel to limit the number of sequential network I/O operations across datacenters. This will reduce the transaction completion time compared with the serial execution of the different parts. The rest of this chapter will describe Carousel's design and transaction protocols in details, and it will also provide the evaluation results in both Amazon EC2 and a private cluster.

3.1 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.

3.1.1 Assumptions

Our design requirements and usage model assumptions are largely based on published information on Spanner [26]. 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., [77, 60, 118, 86]) 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. [15], many applications prefer interactive transactions involving both reads and writes, especially in order to support rapid development. Many existing geo-distributed systems (e.g., [15, 26, 60, 118, 77]) target interactive transactions. Carousel also targets interactive transactions by supporting 2FI transactions.

3.1.2 2FI Transactions

This section introduces 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* [6], 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[52, 86], 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.

[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.

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 [105], 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* [105]. 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.

3.1.3 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 [20] or Zookeeper [47], 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 [53] to map keys to partitions.

Client Library
Begin() → *Transaction Object*
Transaction Object
ReadAndPrepare(readKeySet, writeKeySet) → *readResults*
Write(key, val)
Commit() → *committed/aborted*
Abort()

Figure 3.1: Carousel's client interface

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 3.1. 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 [118], 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.

3.2 Protocol

In this section, we first describe Carousel's basic transaction protocol that takes advantage of the properties of 2FI transactions (see Section 3.1.2) 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.

3.2.1 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, 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 wide-area network roundtrip 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 wide-area network roundtrips. 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 3.2 shows an example of the basic Carousel protocol when there are no failures. In the figure, solid and dashed arrows stand for intra-datacenter and inter-datacenter messages, respectively, and dashed rectangles represent replication operations. 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 3.2.

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 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.

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 wide-area network roundtrip. 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 3.2.1.

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

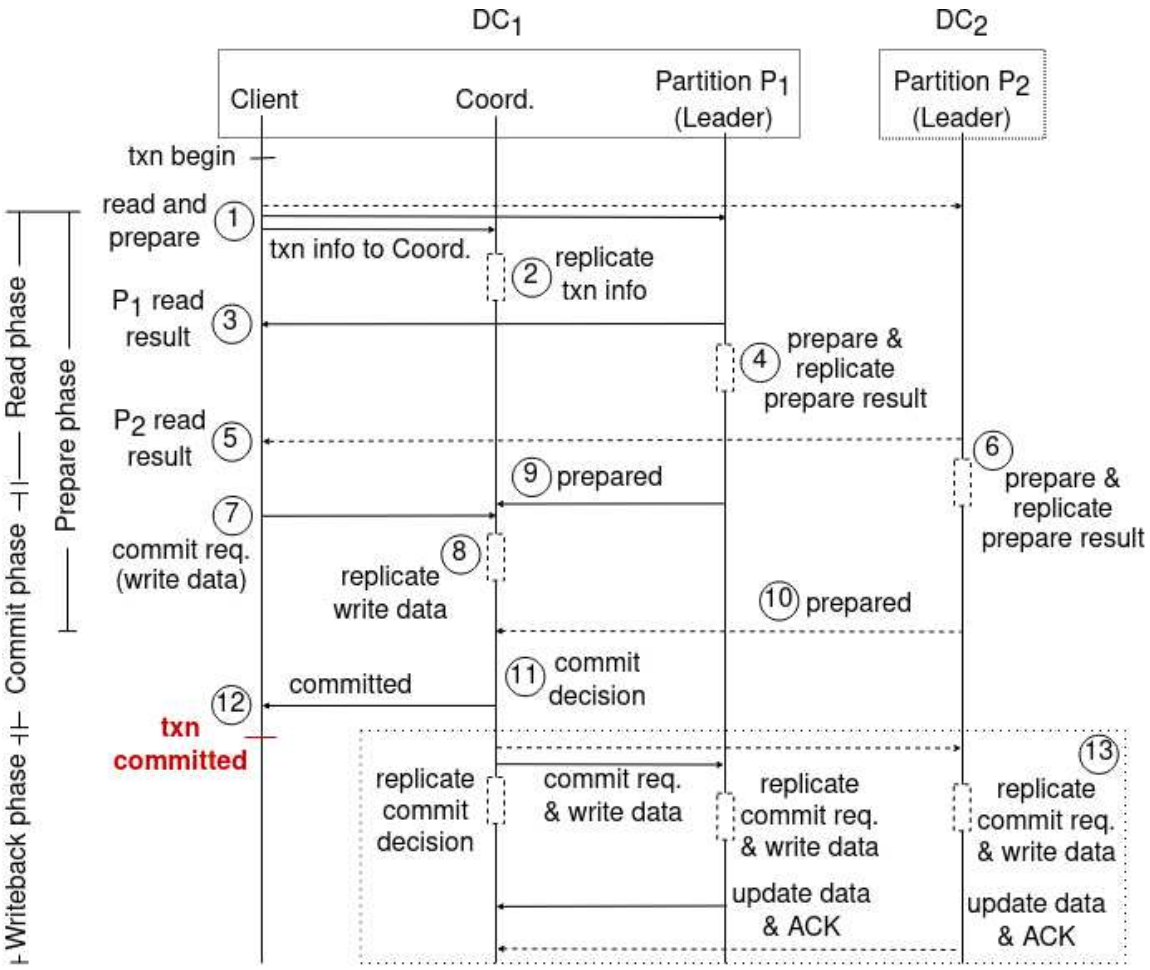


Figure 3.2: An example of Carousel's basic transaction protocol

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 3.2.3). 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.

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 wide-area network roundtrips. However, none of this latency is exposed to the Carousel client application.

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 wide-area network roundtrips. One wide-area network roundtrip 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 wide-area network roundtrip is required for each participant leader to replicate its prepare decision to its consensus group. However, since the Prepare phase runs concurrently with the Read and Commit phases, each of which requires one wide-area network roundtrip, the total number of wide-area network roundtrip delays observed by the client is at most two.

3.2.2 Parallelizing 2PC and Consensus

In the basic Carousel transaction protocol, 2PC and consensus together require two wide-area network roundtrips 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 wide-area network roundtrip in many situations.

CPC borrows ideas from both Fast Paxos [66] and MDCC [60] to introduce a fast path that can prepare a transaction in one wide-area network roundtrip 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 wide-area network roundtrips when there are no failures.

We use an example in Figure 3.3 (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.

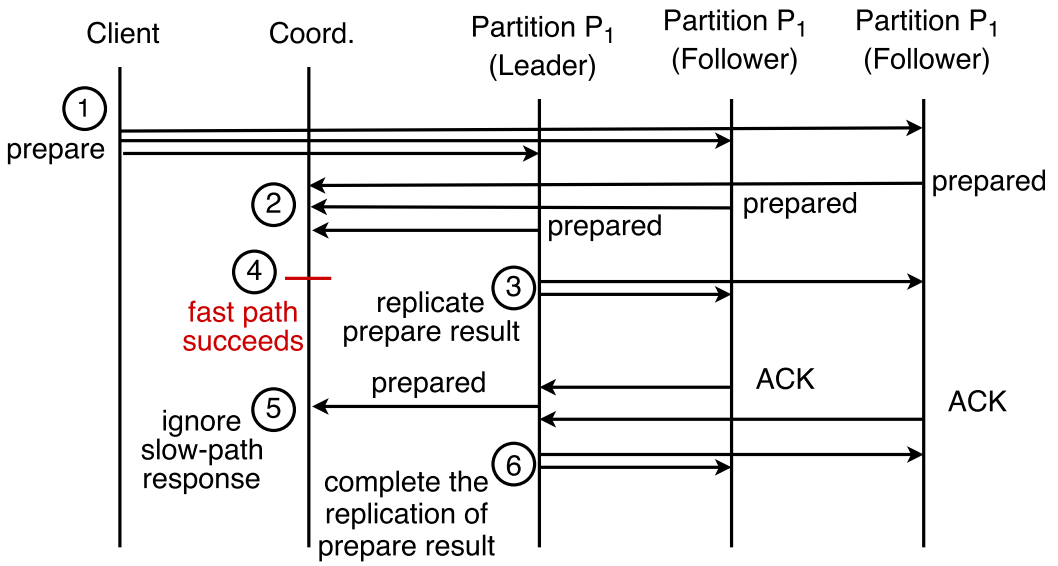
On the fast path, the coordinator can determine a participant partition's prepare decision for a transaction if both of the following conditions are satisfied:

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.

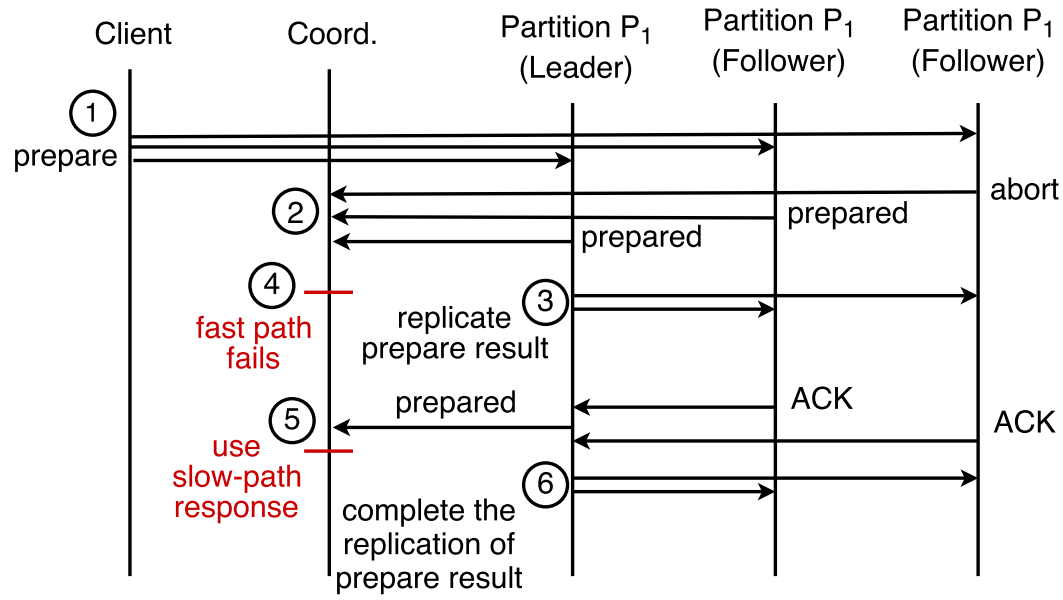
2. The participant leader must be part of the supermajority.

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 [91] 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 3.2.3. 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.

[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 [66].



(a) No conflicts



(b) Conflicts

Figure 3.3: An example of CPC

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 wide-area network roundtrip 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 3.3 (b) to illustrate the case when the fast path fails because of 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.

3.2.3 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.

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.

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.

Leader Failures

In Carousel's basic protocol, the state of each participant leader is replicated to its consensus group using Raft [91] 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 [66]. 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 3.2.2) in at least a majority of the $f+1$ lists. As we need to examine every transaction in the $f+1$ lists, the complexity of computing the potentially prepared transactions that use the fast path will be O(fk), where k is the number of transactions in the largest list. k is bounded by the system load, which typically ranges from tens of transactions to thousands of transactions per second. f is usually one (or two) since most applications use only three (or five) replicas.

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. The complexity of this process will be O(mn), where m and n are the number of potentially pre-

pared transactions using the fast path and the number of slow-path prepared transactions, respectively. Both m and n are bounded by the system load.

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.

Network Partitions

A network partition within a consensus group splits the group into two subsets, and a node in a subset cannot communicate with any node in the other subset. In Carousel, only the subset that has at least a majority of nodes can continue its service. This subset will consider nodes in other subsets as having failed, handling these failed nodes in the same way as we have described in follower and leader failures.

Network partitions between consensus groups are unlikely to happen in Carousel because the replicas of each partition are distributed across geographically different datacenters. If the whole consensus group of a partition is totally isolated from any other site in the system, any transactions that access the partition will stall until the network has recovered.

3.2.4 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.

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.

Read-only Transactions

In practice, read-only transactions are common. Carousel follows Spanner in using timestamps to complete read-only transactions in one network roundtrip. Instead of relying on Spanner's TrueTime API, Carousel keeps multiple versions of each data record, where the version number is based on a monotonically increasing timestamp. For a read-write transaction, each participant returns a timestamp that is larger than the timestamps of the data records accessed by the transaction as part of its prepare result to the coordinator. The coordinator uses the largest received timestamp as the commit timestamp of the transaction.

In contrast, for a read-only transaction, the client assigns the transaction timestamp and sends read requests directly to participant leaders. Upon receiving a read-only transaction,

a participant leader will read the version of the requested data with the largest timestamp that is smaller than the transaction's timestamp. If the transaction has conflicts with concurrent read-write transactions, or its timestamp is larger than the timestamp that the leader will assign for a future read-write transaction, the leader will abort the transaction. The client completes the transaction when it receives all the required data. The transaction is aborted if the client receives an abort from a participant leader.

3.3 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 [42] to implement the RPC functions for data servers. Although we extend an open-source implementation [37] of Raft [91] to manage replicas for each partition, we do not implement fault tolerance in our prototype.

Our evaluation (see Section 3.4) 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.

3.4 Evaluation

In this section, we evaluate Carousel Basic and Carousel Fast by comparing their performance with TAPIR [118], 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.

3.4.1 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

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

Table 3.1: Network roundtrip delays (ms) between different datacenters

(Sydney), and Asia (Tokyo). Table 3.1 shows the network roundtrip delays 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, 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 [107] 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.

3.4.2 Workloads

We evaluate our system using two different workloads. The first workload is Retwis [68], 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 [34], which extends the YCSB key-value store benchmark [25] 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 [118] 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 3.2.

For both workloads, we populate Carousel Basic, Carousel Fast, and TAPIR with 10

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 3.2: Transaction profile for Retwis from TAPIR [118]

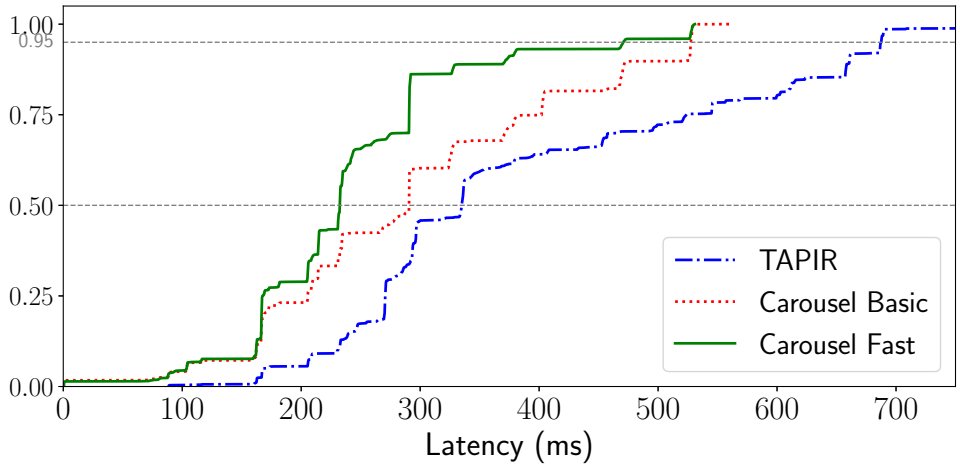


Figure 3.4: Latency CDF for the Retwis workload

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.

3.4.3 Retwis Amazon EC2 Experiments

We now evaluate the performance of the different systems using the Retwis workload. Figure 3.4 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. Furthermore, many applications only require a low to moderate throughput of transactions that access data partitions across different

regions. For example, most operations in Google F1 only access one partition [26]. We will later evaluate the performance of these systems under a heavy load in our throughput experiments in Section 3.4.4. 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:

- 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.

3.4.4 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

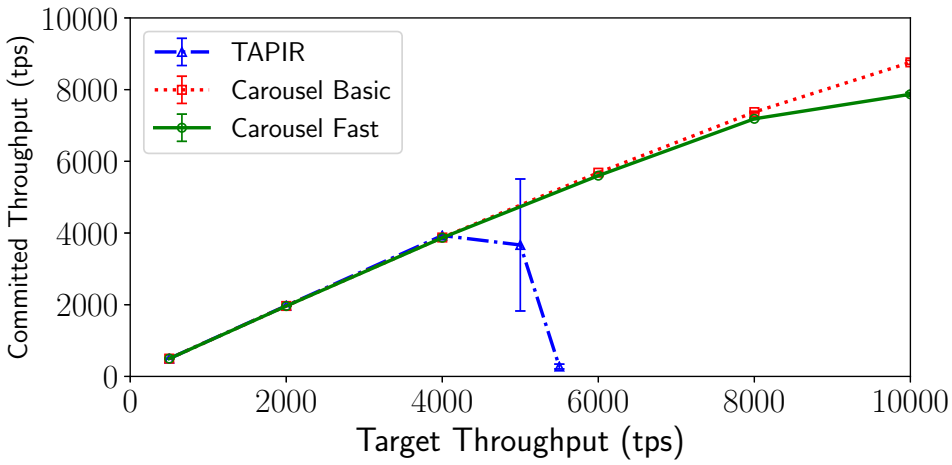


Figure 3.5: Committed throughput versus target throughput

simulate 5 geographically distributed datacenters by using the Linux traffic control utility 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 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.

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 3.5

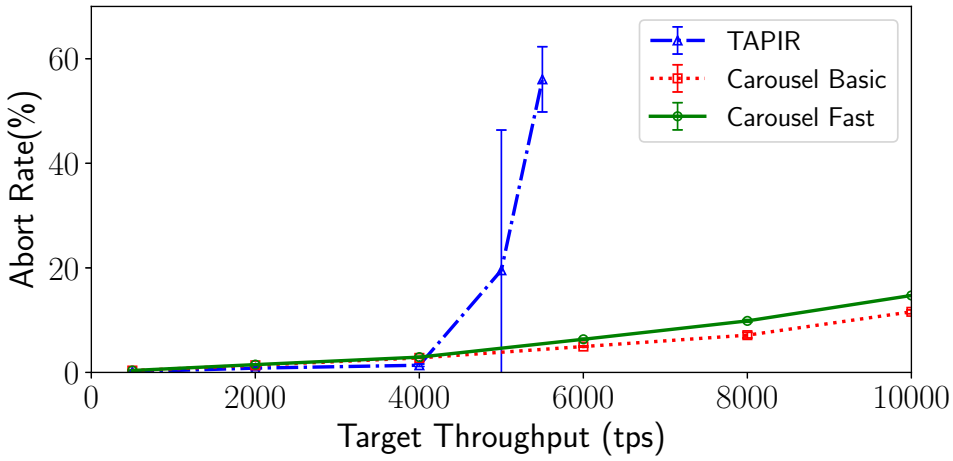


Figure 3.6: Abort rate versus target throughput

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. One reason is that the TAPIR implementation has less parallelism in processing concurrent transactions compared to Carousel. Furthermore, as the queuing delay results in higher transaction completion time, data contention at servers increases as well, causing more transactions to read stale data and get aborted.

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 target throughput earlier than Carousel Basic, leveling off at approximately 8000 tps. This is because Carousel Fast runs fast path and slow path in parallel, resulting in transaction coordinators (i.e., partition leaders) needing to process 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 3.6 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 3.6 also shows that Carousel Fast's abort rate is higher

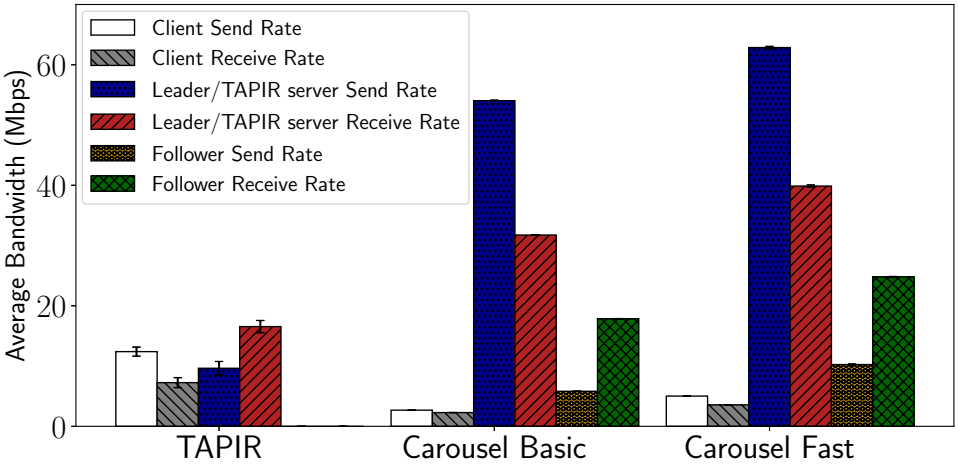


Figure 3.7: Bandwidth used at a target throughput of 5000 tps

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.

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 3.7 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. Also, in this experiment, a partition leader in Carousel serves as a transaction coordinator, which replicates the transaction's write data to its followers before disseminating the data to participants in the transaction. 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. In addition, Carousel Fast requires more bandwidth than Carousel Basic since Carousel Fast performs both fast path and slow path concurrently. This allows

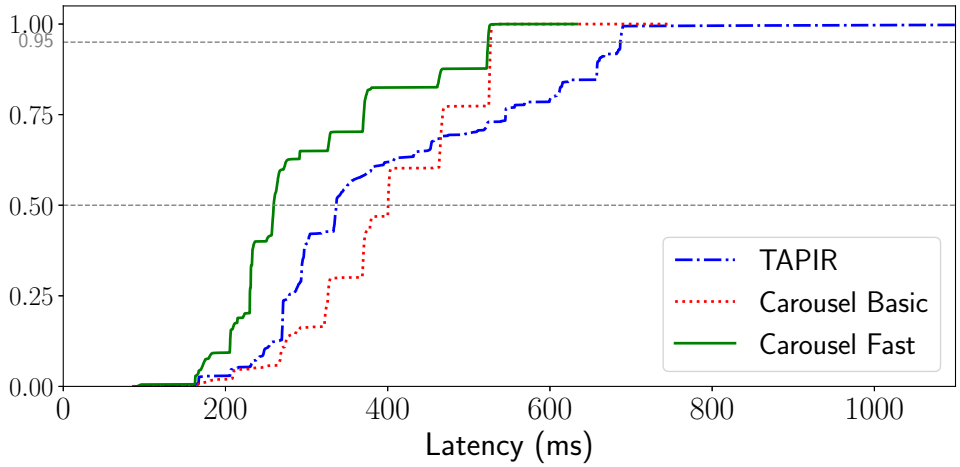


Figure 3.8: Latency CDF for the YCSB+T workload

Carousel Fast not to sequentially start the slow path after the fast path fails, resulting in lower transaction completion time.

3.4.5 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 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 3.8 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).

3.5 Chapter Summary

Many large-scale distributed applications service global users that produce and consume data. Geographically distributed database systems, like Spanner and CockroachDB, sequentially execute different parts in transaction processing. This requires multiple wide-area network roundtrips to commit a distributed transaction, which causes high latency. To reduce the latency, this thesis introduces 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.

Chapter 4

Domino

Many applications rely on state machine replication (SMR) to tolerate failures. Most SMR implementations use a consensus protocol to ensure that the replicas agree on the order of state updates. In a leader-based consensus protocol, a client has to wait for two network roundtrips to learn whether its request has been committed. This introduces significant delays, especially when the client is geographically distant from the leader. Fast Paxos was introduced to reduce the number of network roundtrips to just one by having clients directly contact every replica. However, concurrent client requests may arrive at replicas in different orders, requiring the protocol to fall back to a slow path.

This chapter introduces Domino, a low-latency SMR protocol for wide-area networks (WANs). Domino uses network measurements to predict the expected arrival time of a client request to each of its replicas, and assigns a future timestamp to the request indicating when the last replica from the supermajority quorum should have received the request. With accurate arrival time predictions and in the absence of failures, Domino can always commit a request in a single network roundtrip using a Fast Paxos-like protocol by ordering the requests based on their timestamps.

Additionally, depending on the network geometry between the client and replica servers, a leader-based consensus protocol can have a lower commit latency than Fast Paxos even without conflicting requests. Domino supports both leader-based consensus and Fast Paxos-like consensus in different cycles of the same deployment. Each Domino client can independently choose which to use based on recent network measurement data to minimize the commit latency for its requests.

Domino's performance depends on the stability of network delays between datacenters. The rest of this chapter will first describe our measurements of inter-datacenter network

delays on Microsoft Azure, showing that the delays are relatively stable and can be used to predict the current behaviour of the network. It will also present the impact of network geometry on the commit latency of different protocols, describe Domino's design in details, and provide our experimental results on Microsoft Azure.

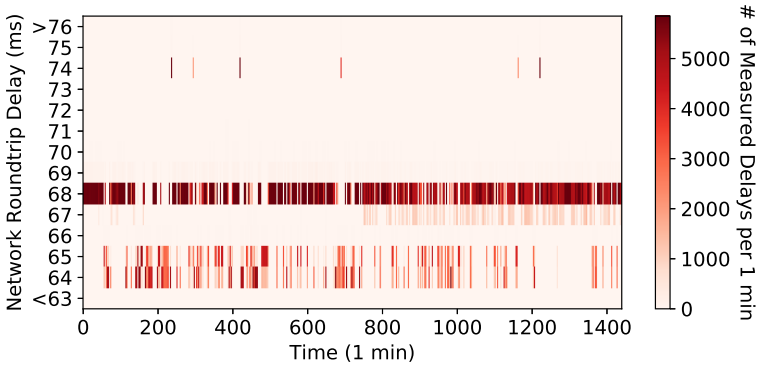
4.1 Inter-Datacenter Network Delays

Modern cloud providers, such as Microsoft Azure [81], AWS [7], and GCP [43], support private network peering between virtual machines in different datacenters. In such a private network, network traffic only traverses the cloud provider's backbone infrastructure instead of being handed off to the public Internet, which should reduce the variability of network delays. This section will describe our network delay measurements between datacenters on Azure, and show that recent network measurements can be used to estimate the one-way delays between datacenters.

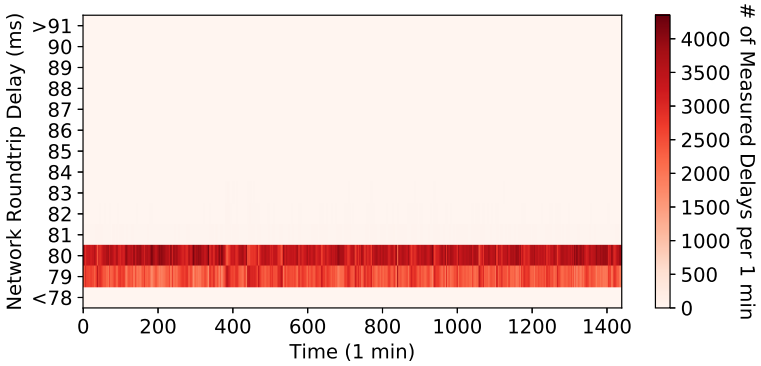
Our measurements include 6 Azure datacenters in different global locations, Washington (WA), Virginia (VA), Paris (PR), New South Wales (NSW), Singapore (SG), and Hong Kong (HK). We use a Standard_D4_v3 VM instance at each datacenter, and each instance runs a client and a server. We implement the client and server in the GO language, and we use gRPC [42] to provide communication between a client and server. Our measurements last for 24 hours. For every 10 ms, a client invokes an RPC request to the server in every other datacenter. The server returns its timestamp in its RPC response, and the client records the RPC completion time. The measured delay includes the network roundtrip delay between the client and server and the RPC processing delay. Since each VM instance is under light load in our measurements, the RPC processing delay is negligible compared to the network propagation delay between datacenters. In the rest of this section, we assume the measured delay is equivalent to the network roundtrip delay.

Figure 4.1 shows the network roundtrip delay from VA to WA, PR, and NSW, respectively. The variance of the network roundtrip delay is relatively small compared to the minimum measured delay which is dominated by the network propagation delay. The network roundtrip delays between other datacenters show a similar pattern in our measurements. We have also measured the network roundtrip delays between 9 datacenters at different locations in North America, and the results also show a small variance of network delays between these datacenters. The data traces and the scripts to parse the traces are publicly available online [3, 1, 2].

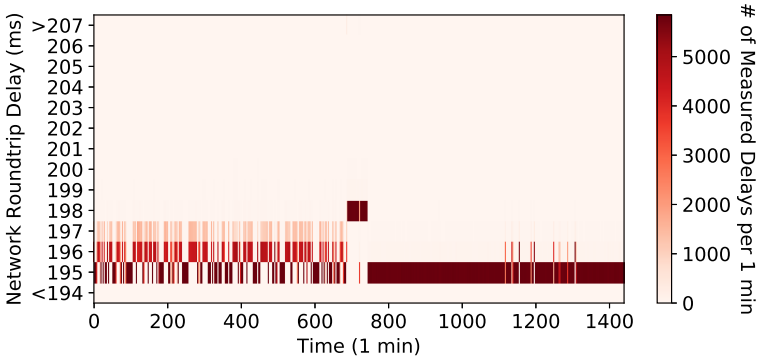
As shown by our measurements, the network roundtrip delay between datacenters is relatively stable in most cases. We next look more closely at whether recent network



(a) Washington (WA)



(b) Paris (PR)



(c) New South Wales (NSW)

Figure 4.1: Network roundtrip delays with the host datacenter in Virginia (VA)

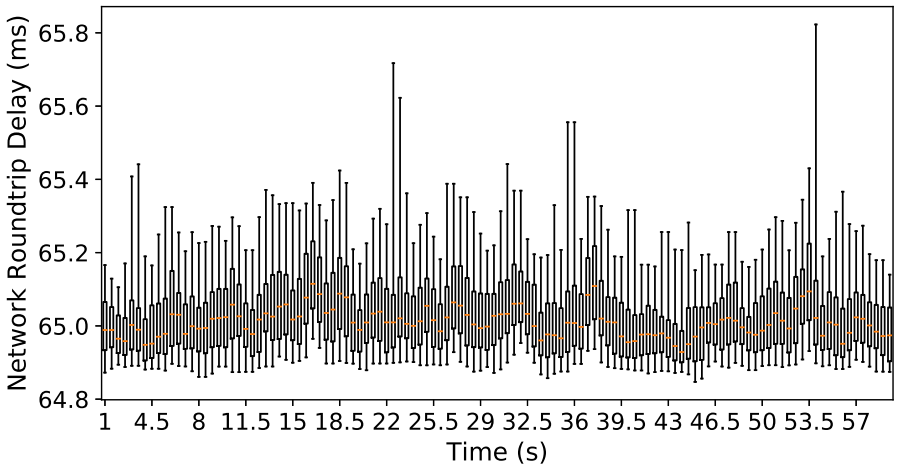


Figure 4.2: Network roundtrip delays between VA and WA

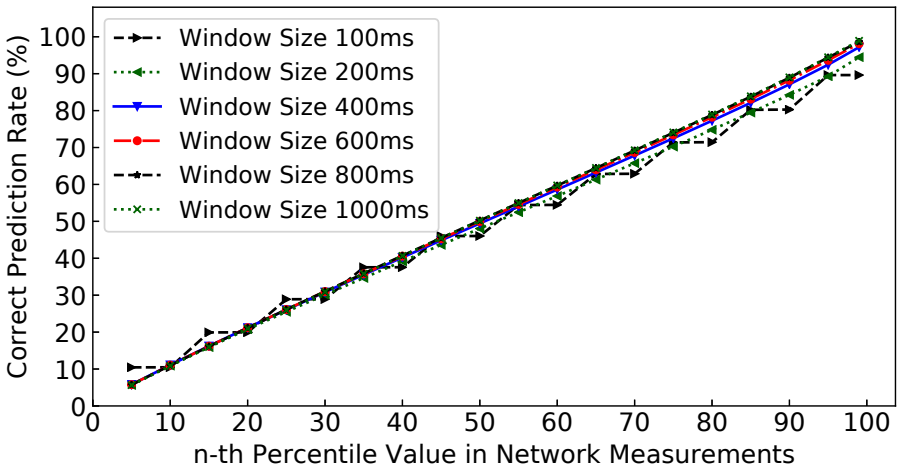


Figure 4.3: Correct prediction rate

delay information would be a good source to estimate the current network roundtrip delay. Figure 4.2 shows the distribution of our measured delays from VA to WA for a 1 min duration starting from the 12th hour in our measurements. Each box in the figure consists of the measured delays in the past one second, and two adjacent boxes have an overlap of half a second. The whiskers represent the 5th and 95th percentile network roundtrip delay. As shown in the figure, the variance of the network roundtrip delays is small during a short period of time. Our analysis demonstrates that the network roundtrip delay between datacenters is relatively stable, and it is possible to predict the current behaviour of the network based on recent network measurement data.

Stable network delays are important since Domino clients need to predict the arrival time of their requests at the replicas when they uses Domino's Fast Paxos-like consensus. Clients perform periodic one-way delay (OWD) measurements to the replicas, and use the n-th percentile value in the past time period (i.e., window size) to predict request arrival times. We will describe the details of our OWD measurement scheme in Section 4.3.4. As Domino only requires a request to arrive at a replica before the request's timestamp to achieve low commit latency, we consider an arrival-time prediction to be correct as long as the actual arrival time is equal to or smaller than the predicted timestamp.

For our data traces from VA to WA, Figure 4.3 shows the correct prediction rate using different window sizes and percentile values for estimating a request's arrival time. The figure shows that using the 95th percentile latency with a small window size of one second is sufficient to achieve a high prediction rate. We further analyze the effectiveness of our approach across different physical paths and latencies by breaking down the results by the geographical regions of the client and server. We find that the correct prediction rate ranges from 93.86% (NSW to PR) to 94.86% (SG to HK) and is largely independent of where the client and server are located.

4.2 Impact of Network Geometry

Depending on the network geometry, some clients may have lower commit latency by using Fast Paxos than a leader-based protocol, and some other clients may have the opposite results even in the absence of conflicting requests. This is because a Fast Paxos client must receive a response from a supermajority of replicas, whereas a Multi-Paxos leader only needs to receive a response from a majority of replicas. Figure 4.4 shows an example where Multi-Paxos has lower commit latency than Fast Paxos, even if Fast Paxos successfully uses its fast path to commit client requests. In this example, the commit latency for Multi-

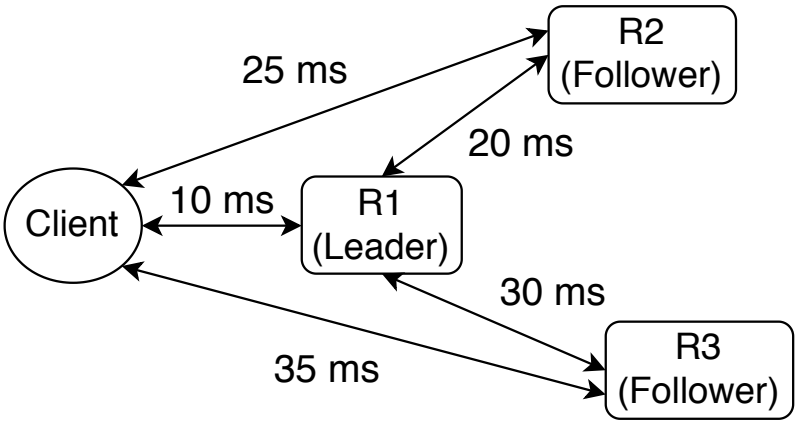


Figure 4.4: Multi-Paxos (30 ms) versus Fast Paxos (35 ms)

Paxos is only 30 ms as it only requires agreement from $R1$ and $R2$, while the commit latency for Fast Paxos is 35 ms as it requires agreement from all three replicas.

In a deployment where replicas are located at every datacenter, a client for a multi-leader consensus protocol (e.g., EPaxos and Mencius) can always send requests locally to a replica located in the same datacenter. These requests can be committed in one WAN roundtrip and experience lower commit latency than using Fast Paxos. However, as the number of datacenters increases, fully replicating data to every datacenter can be prohibitively expensive. Also, the latency to replicate requests would increase with the number of replicas since a quorum of replicas are needed to complete the replication. A study [8] from Facebook shows that many of its applications will maintain three or five data replicas while requests can originate from datacenters in other geographic regions.

Given our delay measurement data in Section 4.1, we analyze how often a client can have lower commit latency by using Fast Paxos than Multi-Paxos and Mencius. Our analysis consists of 6 Azure datacenters and uses the average network roundtrip delays between the datacenters, which are shown in Table 4.1.

In our analysis, we use three datacenters for the locations of three replicas, and we use one datacenter for a client. We iterate over all possible locations of the replicas and the client to compare the commit latency of the three protocols. We randomly select a replica to be the leader for Multi-Paxos, and we make the client send its request to the closest replica in Mencius. Our results show that Fast Paxos has lower commit latency than Mencius and Multi-Paxos for 32.5% and 70.8% of the cases, respectively.

These results show that the best consensus protocol to use depends on the network

	WA	PR	NSW	SG	HK
VA	67	80	196	214	196
WA	-	136	175	163	141
PR	-	-	234	149	185
NSW	-	-	-	87	117
SG	-	-	-	-	35

Table 4.1: Network roundtrip delays (ms) between datacenters

geometry. Instead of simply using one protocol, Domino adopts both Fast Paxos and a leader-based protocol, and allows a client to dynamically choose which protocol to use based on network measurement data.

4.3 Architecture

Domino is a protocol for state machine replication (SMR) in WANs. It aims to achieve low commit latency, where operations are replicated and ordered but may not yet have been executed. Commit latency is important to most use cases of SMR while execution latency is only important to some use cases. For example, a common use of SMR is to order and execute operations in logging systems that modify the state machine, but often have no return values. For such an operation, a client would only need to wait until it is committed. The execution of the operation can be performed asynchronously. Furthermore, execution latency can be masked in many ways, such as through application-level reordering.

To achieve low commit latency, Domino has two subsystems, Domino's Fast Paxos (DFP) and Domino's Mencius (DM), which execute in parallel. Each subsystem independently commits client requests, and Domino applies a global order for all of the committed requests in both DFP and DM. This section will first describe Domino's assumptions, and then it will give the details of the Domino protocol.

4.3.1 Assumptions

Domino has the following requirements and usage model assumptions.

Fault tolerance. Domino targets the crash failure model. It requires $2f + 1$ replicas to tolerate up to f simultaneous replica failures.

Inter-datacenter private network. Domino assumes that replicas and clients (i.e., application servers) are connected within an inter-datacenter private network. This is practical in modern datacenters. Microsoft Azure, for example, provides global virtual network peering [82] to connect virtual machines across datacenters in a private network, and the network traffic is always on Microsoft's backbone infrastructure instead of being handed off to the public Internet. Such a network provides more stable and predictable network delays between datacenters than the public Internet.

Asynchronous FIFO network channels. Domino requires a FIFO network channel between replicas. To achieve this, Domino uses TCP as its network transport protocol.

Clock synchronization. Domino assumes that clients and replicas have loosely synchronized clocks. A network time protocol, like NTP [83], can achieve loosely clock synchronization in WANs. A severe clock skew will only affect Domino's performance instead of correctness.

4.3.2 Overview of Domino

A deployment of Domino consists of a set of replica servers (i.e., replicas) that are running in datacenters. Domino uses a request log to store client operations that it applies to the replicated state machine. At a log position, Domino runs one consensus instance to ensure that the same request is in the same log position across all replicas. Consensus instances will run in parallel, and they can use different consensus protocols.

Domino pre-classifies its log positions into two subsets, and it uses two different subsystems, Domino's Fast Paxos (DFP) and Domino's Mencius (DM) to manage the two subsets, respectively. DFP uses a Fast Paxos consensus instance (i.e., DFP instance) for each of its log positions, and DM uses an extension of Mencius for its log positions. Domino's log order defines a global order for DFP's and DM's consensus instances. DFP and DM can independently commit client requests in their log positions.

Domino's clients are application servers that are also running in datacenters, and a client can be in a different datacenter from the replicas. A client uses a Domino client library to propose a request. In the rest of this chapter, a client refers to a Domino client library unless specified otherwise.

In order to achieve low commit latency, Domino clients estimate the commit latency of using DFP and DM and select the one with the lower latency. A client periodically measures its network roundtrip delays and estimates its one-way delays to the replicas. It will use its measurements to estimate the commit latency of using DFP and DM. We will describe the details of choosing between DFP and DM later in Section 4.3.6.

4.3.3 Domino's Fast Paxos

In this section, we introduce Domino's Fast Paxos (DFP), a practical way of using Fast Paxos to implement state machine replication. In order to achieve low commit latency, DFP aims to increase the likelihood that DFP instances succeed in using the fast path to commit client requests. DFP makes a client estimate the arrival time of its request to a supermajority of replicas, and it assigns this future timestamp to the request. By ordering requests based on their timestamps, replicas will accept the requests in the same order, and DFP will commit the requests via the fast path.

Instead of dynamically ordering requests based on their timestamps, DFP pre-associates each of its log positions with a real-clock time (i.e., timestamp) in an ascending order. When a replica receives a client request, it will try to accept the request by using the consensus instance at the log position that has the request's timestamp. DFP by default uses nanosecond-level timestamps, where there will be one billion log positions within a single second. The probability that two concurrent requests have the same timestamp is low when the target throughput is only tens of thousands requests per second. This reduces the likelihood that two requests collide in one consensus instance, in which case DFP has to use the slow path to commit the requests. As a result, this increases the chances DFP commits requests via the fast path and achieves low commit latency.

Common Cases

When a client proposes a request, it will assign the request with a timestamp indicating a DFP's log position. The timestamp is the request's arrival time at a supermajority of the replicas. We will describe how a client uses network measurements to estimate its request's arrival time at replicas in Section 4.3.4.

The client will send its request and the assigned timestamp to every replica. Once a replica receives the request, it uses the timestamp to identify the log position for the request. It will serve as an *acceptor* in the consensus instance for that position to accept the request. DFP makes the client serve as a *learner* of the consensus instance. DFP also has a replica, the DFP coordinator, to be the learner of all of its consensus instances. The DFP coordinator is the replica that is required by Fast Paxos' coordinated recovery protocol [66] to handle request collisions. We will describe the collision handling later in this section.

As shown in Figure 4.5, we will use an example to describe the message flow of a DFP instance when the fast path succeeds. In our description, we use circled numbers (e.g., ①)

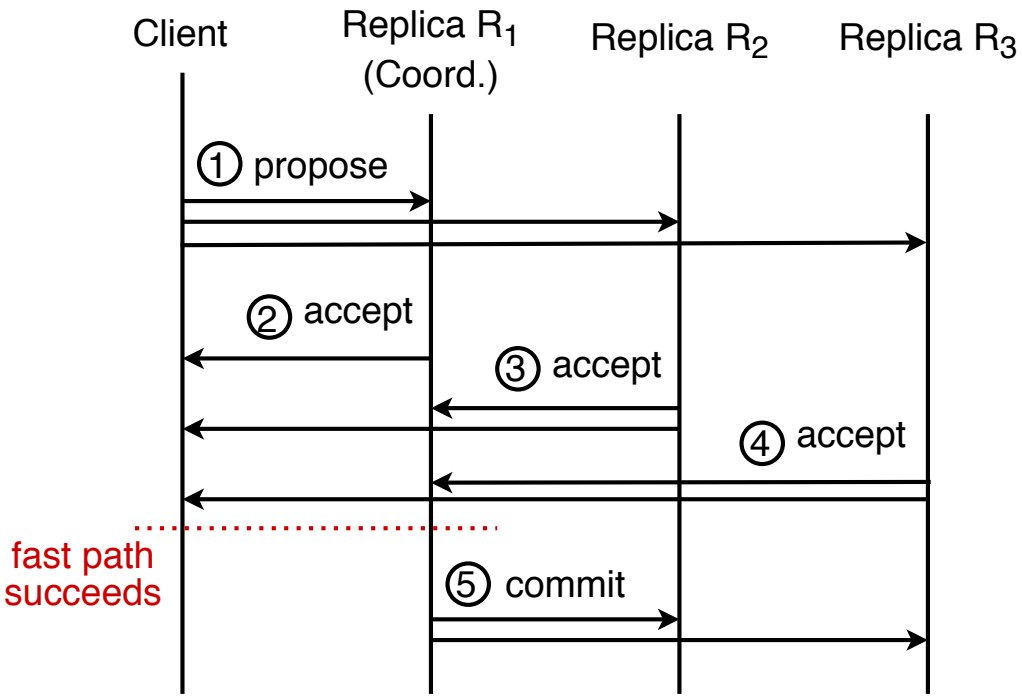


Figure 4.5: An example of DFP's fast path

to refer to the corresponding numbered points in the message flow illustrated in Figure 4.5. In the example, there are a total of three replicas. A client sends (①) its request to every replica.

Once a replica receives the request, it accepts the request and sends (② - ④) its acceptance decision to the client and the coordinator. When the client receives the acceptance decision from at least a supermajority of the replicas, it learns that DFP commits its request, which costs only one network roundtrip time.

When the coordinator learns the same result as the client, it commits (not executes) the request in its log. It also asynchronously notifies (⑤) the other replicas to commit the request. The replicas will commit the request once they receive the notification.

Filling Empty Log Positions

Since DFP makes clients choose log positions for their requests, there will be positions that no client will choose. DFP will fill these empty positions with a special operation, *no-op*, which has no effect on the state machine. One approach is to use a dedicated *proposer* to propose no-ops for unused log positions. However, it is challenging for the proposer to determine when to propose a no-op for a log position because the proposer might have an out-of-date log status, and the no-op may collide with a concurrent client's request. Also, it is expensive to propose a no-op for every empty log position.

To reduce the collisions between no-ops and client requests, DFP leverages clock time to fill no-ops at empty log positions that clients are unlikely to use. As a client always uses a future timestamp for its request, it is unlikely that a replica will receive a request that has timestamp smaller than its current clock time. When it is time T, a replica will accept no-ops for all empty positions that have a smaller timestamp than T without receiving a no-op proposal. This avoids the need to propose a no-op to every replica since the no-ops accepted by different replicas are identical.

It will be expensive for a replica to send an acceptance message to the DFP coordinator for each no-op since there could be many empty log positions. To reduce the number of messages for no-ops, DFP borrows ideas from Mencius. In Mencius, a replica uses a log index to indicate that it has accepted no-ops for all of the empty log positions until that index. By leveraging FIFO network channel, the replica only needs to send the index to other replicas.

In DFP, since each log position has a timestamp, a replica uses its current time, T, to indicate that it has accepted no-ops for all of the empty log positions that have a timestamp smaller than T. Because the network channel provides FIFO ordering, when the replica sends T to the DFP coordinator, it has already notified the coordinator about all of the accepted client requests at log positions that have a timestamp smaller than T. Also, the coordinator can use T to infer the log positions at which the replica has accepted no-ops.

Instead of using a special message for T, a replica can piggyback T on any message that it sends to the DFP coordinator. Additionally, each replica periodically sends the DFP coordinator a heartbeat message that includes its current time.

Handling Incorrect Estimations and Collisions

In DFP, it is possible that a client's request arrives at a replica after its estimated arrival time due to a number of factors, such as route changes, network congestion, packet loss,

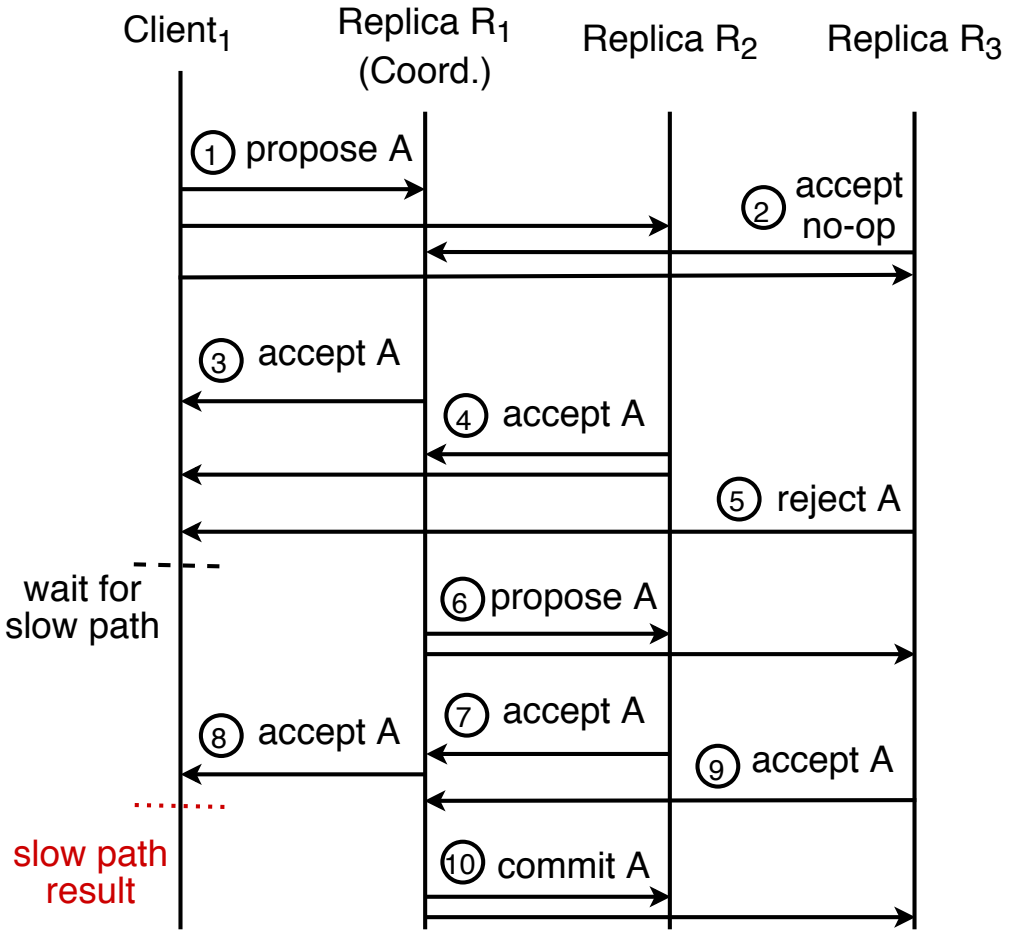


Figure 4.6: An example of DFP's slow path

and clock skew. When a request arrives at a replica later than its timestamp, the replica has already accepted a no-op at the request's target log position. This is equivalent to a collision of two concurrent requests at a log position. Such a collision may cause the fast path to fail, since the fast path requires a supermajority of replicas to agree on the same request for a log entry.

When the fast path fails due to request collisions, DFP uses the coordinated recovery protocol [66] to commit requests. Figure 4.6 shows an example in which a client request arrives at a replica later than its timestamp. In this example, the client sends ((1)) a

request to every replica. Before its request arrives at replica R_3, R_3 has passed the request's timestamp and accepts (②) a no-op at the request's target log position. Although R_1 and R_2 both accept (③, ④) the request, there is no supermajority formed because R_3 rejects (⑤) the request. In this case, the client waits for the slow-path response from the DFP coordinator.

By following the recovery protocol, the coordinator will use the slow path to accept (⑥, ⑦) the request. The coordinator will send (⑧) the slow-path result to the client, and it will asynchronously ask every replica to commit the request.

As DFP uses nanosecond-level timestamps to identify log positions, it is unlikely that two clients assign the same timestamp for their requests when the target throughput is tens of thousands of requests per second. If there is a fixed set of clients, pre-sharding timestamps among the clients can be used to completely avoid collisions between client requests. For example, with only one thousand clients, each client can replace the three least significant digits in its timestamps with its ID. In this case, each client can still send up to one million requests per second with unique timestamps, which will be far beyond the target throughput in many systems.

When the set of clients is dynamic, it is possible but rare that two clients assign the same timestamp for their requests, which will collide at a log position. In this case, DFP can only commit one of the requests at the position. If a supermajority of replicas have accepted a request, DFP will commit the request via the fast path. Otherwise, DFP will fall back to its slow path to commit one of the requests by following the recovery protocol. The DFP coordinator will propose the other request through Domino's Mencius.

4.3.4 Estimating Request Arrival Time for DFP

In DFP, a client assigns its request with a timestamp indicating a future time when the request should have arrived at a supermajority of the replicas. To ensure that most requests can arrive at replicas on time, one simple approach is to assign each request with a large timestamp. One problem with this approach is in determining how large the timestamp should be. An unnecessarily large timestamp can significantly delay the execution of a request since Domino executes requests in log order, which we will describe later in Section 4.3.7. One alternative way to set the timestamp is to add a constant delay value on the current time. However, using a constant delay value cannot adapt to changes in the system environment, such as a network routing change or clock skews, which may cause a client to underestimate its requests' arrival time. To address these problems, a Domino client will leverage network measurements to predict a request's arrival time at each replica

from\to	VA	WA	PR	HK	SG	NSW
VA	-	12.11	7.82	34.49	34.64	49.51
WA	5.36	-	4.89	29.15	27.87	38.05
PR	9.42	12.31	-	32.9	33.74	44.29
HK	9.76	3.08	2.34	-	4.52	22.22
SG	5.79	3.25	5.42	6.44	-	21.29
NSW	2343.97	700.7	105.86	48.7	20.38	-

Table 4.2: The 99th percentile misprediction value (ms) by using half-RTTs

from\to	VA	WA	PR	HK	SG	NSW
VA	-	5.26	5.42	5.12	6.24	5.72
WA	5.24	-	4.36	4.74	5.83	5.8
PR	5.5	4.58	-	5.03	6.15	5.45
HK	5.31	5.09	4.61	-	5.94	5.42
SG	5.86	5.62	5.71	5.91	-	5.9
NSW	5.25	4.51	4.31	4.97	5.87	-

Table 4.3: The 99th percentile misprediction value (ms) by using Domino's OWD measurement technique

as accurate as possible, and it will set its request's timestamp to be the qth smallest arrival time, where q is the number of replicas in a supermajority quorum.

To estimate a request's arrival time at a replica, a Domino client will add its current time and its predicted network one-way delay (OWD) to the replica. A naive approach to predict the OWD is to take half of a network roundtrip time (RTT). However, in networks where the forward and reverse paths between a client and replica are mostly disjoint, this approach may significantly under or over estimate the request arrival time at the replica. The request may also arrive at a replica later than its timestamp due to clock skew, or when a replica experiences a large request processing delay.

To improve the estimation accuracy of a request's arrival time at a replica, we propose to use both network measurements and replicas' time information to predict OWDs. Specifically, when a replica responds to a client's probing message, it piggybacks its current timestamp on the response. The client will calculate the OWD to the replica by taking the difference between the replica's timestamp and the probing message's sending timestamp. It will predict the request arrival time to the replica by taking the n-th percentile of the OWD values that it collected in the past time period.

By using the data traces from Azure that we described in Section 4.1, we compare the arrival-time prediction accuracy between using half-RTTs and our timestamp-based approach to estimate OWDs. In our analysis, we estimate a request's arrival time by using the 95th percentile value from the calculated OWDs in the last one second. We only consider requests that arrive at replicas after their timestamps since DFP may need to fall back to its slow path to commit these requests. We define a request's misprediction value as the difference between its estimated arrival time, and the actual arrival time at the replica. Table 4.2 and Table 4.3 show the 99th percentile misprediction value by using half-RTTs and our approach, respectively. The results are broken down by the geographical regions of the endpoints. Our approach has a 99th percentile misprediction value of up to 6.24 ms, compared to more than 2 s when using half-RTTs. Even discounting this possible outlier for the half-RTT approach, the 99th percentile misprediction value to NSW from any other region is still more than 21 ms.

To accurately predict a request's arrival time at a replica, we need to account for both the OWD and clock skew between the client and replica. However, instead of separating the two factors, our arrival time measurements include both network delays and clock skew between a client and replica, and we use our previous arrival time measurements to predict the current arrival time of a request at a replica. Therefore, stable clock skew should not affect our arrival time prediction accuracy. A sudden clock drift may cause a client to underestimate its requests' arrival time at replicas. However, our arrival time measurements will resolve the difference after a short period of time. This is because the client periodically fetches a replica's clock time and uses the difference between the replica's and its own clock time to estimate its request's arrival time. Furthermore, sudden clock drifts are unlikely to happen in practice [41].

In order to account for the misprediction values shown in Table 4.3 and other external factors, such as network congestion and packet loss, a client can artificially increase the request timestamps by a fixed amount. For example, a client can increase its request timestamp by 8 milliseconds to account for the maximum 99th percentile misprediction value (6.24 ms) found in our data trace. This will not increase DFP's commit latency since each replica accepts the request immediately after receiving the request. However, using a large additional delay will increase the execution latency, which we will describe later in Section 4.3.7.

Furthermore, if a client's requests are frequently committed via DFP's slow path for an extended period of time, the client can switch over to using Domino's Mencius (DM) to reduce the commit latency. Part of our future work is to design a feedback control system that monitors DFP's fast path success rate and have clients adaptively adjust their request timestamps or switch between DFP and DM.

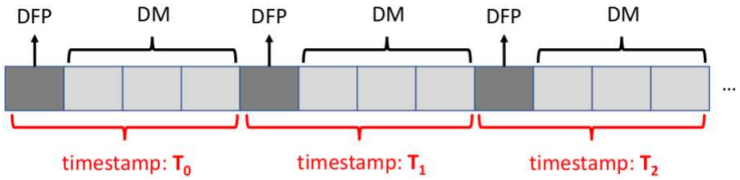


Figure 4.7: An example of Domino's request log

4.3.5 Domino's Mencius

Depending on the network geometry, some clients may have lower commit latency by using a leader-based protocol than using DFP. To achieve low commit latency for these clients, Domino introduces Domino's Mencius (DM), a multi-leader protocol that is a variant of Mencius. DM runs in parallel with DFP, and they manage different subsets of the log positions in Domino's request log. Domino makes a client choose to use DFP or DM in order to achieve low commit latency.

DM pre-shards its log positions to associate each replica with a different set of the positions. A replica will serve as the (DM) leader of the consensus instances for its associated log positions. When a client uses DM, it sends its request to a DM leader, which will accept the request to one of its associated empty log positions.

The log positions of DM and DFP are interleaved in Domino's request log. When few clients use DM, there will be empty DM positions between committed DFP positions. DM should also fill no-ops at empty log positions at the same rate as DFP. To achieve this, Domino arranges its log positions as follows. Between any two adjacent DFP log positions, there is a DM log position that is associated with each DM leader. Domino also pre-associates these DM log positions with the same timestamp as the DFP log position that is immediately before them. Figure 4.7 shows an example of Domino's request log when there are three replicas. In this example, after each DFP log position, there are three DM log positions that are associated with the same timestamp and managed by a different DM leader. When it is time T, a DM leader will fill no-ops at all of its associated log positions that have a timestamp smaller than T. Each DM will piggyback T on its periodic heartbeat message to every other replica.

In DM, a client simply sends its request to a DM leader without assigning a timestamp for the request. Instead, DM delays the timestamp assignment at the leader. When the leader receives the request, it assigns the request with a future time indicating when it should have replicated the request to a majority of the replicas. The leader will use network measurements to predict this replication latency, which we will describe in Sec-

tion 4.3.6. The leader will add its current time and its predicted replication latency to assign a timestamp to the request, and it will accept the request to the corresponding log position. The leader will ask other replicas to accept the request. Once a majority of the replicas (including the leader) accept the request, the leader will commit the request in the log and notify the client and the other replicas.

In Domino, it is possible to replace DM with another leader-based or leaderless consensus protocol, like EPaxos. This is because Domino pre-classifies its log positions into subsets that are managed by different consensus protocols. However, examining the compatibility of different consensus protocols in Domino is beyond the scope of this work.

4.3.6 Choosing between DFP and DM

In Domino, a client measures the roundtrip time to the replicas in order to estimate the commit latency of DFP and DM. It will then use the subsystem that has a lower estimated commit latency. A Domino client will periodically send probing messages to each replica to measure the network roundtrip time and estimate one-way delay. By default, it will use the 95th percentile roundtrip time from measurements collected within the last second to estimate commit latencies.

With the delays to each replica, the potential commit latency of using DFP will be the network roundtrip delay to the furthest replica in the closest supermajority of the replicas. We use q to denote the supermajority quorum number, which is $\lceil \frac{3}{2}f \rceil + 1$ out of total $2f+1$ replicas. The client sorts its roundtrip delays to the replicas, where D_i denotes the ith lowest roundtrip delay. Therefore, the estimated commit latency of using DFP, Lat_{DFP}, will be D_q.

To estimate DM's commit latency, the client needs to know L_r, which denotes the replication latency when replica r is the leader, for each replica. To predict L_r, replica r estimates its delay to every other replica in the same way as a client, and sets the network delay to itself to be zero. The replica sorts its roundtrip delays to every replica, and L_r will be D_m, where m is the majority quorum number (i.e., $f+1$). Each replica will piggyback its estimated latency for replication on the reply to the client's probing messages.

On the client side, the estimated commit latency of using DM, Lat_{DM}, will be $min\{E_r + L_r\}$, where E_r is the network roundtrip delay to replica r, and $r = 1, 2, ..., 2f+1$. The client will compare the estimated commit latency of using DFP and DM, and it will use the one with the lower commit latency. If the client decides to use DM, it will send its requests to the replica that achieves Lat_{DM}.

In Domino, probing messages can be piggybacked on any other messages between clients and replicas. However, by having each client independently measure network delays, the number of probing messages will increase with the number of clients. If there are many clients in one datacenter, we can reduce the number of probing messages by having one dedicated proxy to measure and estimate the network delays to replicas. A client (or a replica) in the datacenter can query the proxy for delay estimation. In this case, a proxy that is not co-located with a replica will send a total of $(2f + 1)R$ probing messages per second, where R is the probing rate. A proxy that is co-located with a replica will send total $2fR$ probing messages per second to the other replicas.

4.3.7 Executing Client Requests

Domino will execute committed requests in their log order. Although Domino can commit requests at different log positions in parallel, Domino will only execute a committed request once it has committed and executed requests (including no-ops) at all of the previous log positions.

As replicas follow wall clock time to fill no-ops at empty log positions, they might not be able to execute a committed request until the time passes the request's timestamp because of empty log positions before that timestamp. As a result, in DFP, if a client uses a large additional delay to increase its request timestamp, this delay will increase the execution latency of this request. However, using a small additional delay (e.g., a couple of ms) will only introduce negligible execution delays compared to the propagation delay in WANs. Also, this could effectively reduce the delays caused by DFP's slow path, which could be hundreds of ms.

Furthermore, in DFP, replicas wait for the coordinator's notification to commit a request at a log position. This may introduce delays for a replica to execute requests at the following DM log positions, which it has committed earlier. Making every replica be a learner in DFP will reduce this delay.

4.3.8 Handling Failures

Domino uses one consensus instance at each log position. The consensus protocol Domino uses for each instance ensures that it selects the same request across all working replicas even with up to f replica failures. As Domino statically partitions the log for DFP and DM, DFP and DM independently manage their log partitions and handle their failures.

The failure handling in DFP and DM is the same as the failure handing in Fast Paxos [66] and Mencius [78], respectively.

When there is a replica failure, by following the failure handling protocol in [78], DM will select one of the remaining replicas to manage the log positions that are associated with the failed replica. In DFP, when there are f replica failures, the number of remaining replicas is insufficient to form a supermajority. In this case, DFP cannot use the fast path to commit client requests although it can still continue to commit requests by falling back to the slow path [66]. Additionally, Domino clients will not receive replies from the failed replicas for their probing messages. The clients will predict large network delays to these replicas after a timeout, and will use DM instead of DFP to achieve lower commit latency.

4.4 Implementation

We have implemented a prototype of Domino in the GO language, which consists of approximate six thousand lines of code and is publicly available online [3]. We use gRPC [42] to implement the network I/O operations between clients and servers (i.e., replicas), including the network delay measurements. We do not implement fault tolerance in our prototype.

Since Domino has many empty log positions to store no-ops, this would cost significant amount of storage space. To reduce the storage overhead, we compress its continuous no-op log entries into one entry in both DFP and DM by using a binary tree data structure for uncommitted log entries. We use a list-based data structure to store the continuous committed log positions from the beginning of the log, and we remove the positions with no-ops to further reduce storage cost.

We have also implemented a state machine replication protocol that uses standard Fast Paxos under the same implementation framework of Domino. We will refer this protocol as Fast Paxos in our evaluation.

4.5 Evaluation

In our evaluation, we compare Domino, Fast Paxos [66], Mencius [78], EPaxos [84], and Multi-Paxos [108]. We use the open-source implementation of Mencius, EPaxos, and Multi-Paxos in [84, 4]. Our evaluation consists of three main parts:

1. Experiments on Microsoft Azure compare the commit latency and the execution latency of the different protocols.

2. Microbenchmark experiments demonstrate that Domino responds to network delay variance in order to achieve low commit latency.

3. Experiments within a local computer cluster compare the peak throughput of the different protocols.

4.5.1 Experimental Settings

Our evaluation mirrors the workload from EPaxos [84], where the state machine is a key-value store, and a client only performs write operations. Such a workload represents applications that only replicate operations that change the replicated state. An example would be a logging system that mostly processes write operations. Furthermore, many applications handle reads outside of the replication protocol by reading data directly from a replica instead of ordering reads together with writes. This optimization can improve read performance at the cost of potentially reading stale data. Therefore, from the perspective of the replication protocol, workloads from applications that employ this type of read optimization are effectively write-only.

Our experiments use the following default settings, unless specified otherwise. Our workload consists of one million key-value pairs. The size of a key or a value is 8 B, and a request's size will be 16 B, which is the same as the request size in [84]. In each experiment, replicas are selected from a fixed set of datacenters. A client is selected from the same fixed set of datacenters and does not have to be co-located with a replica. Each client sends 200 requests per second. The requests select keys based on a Zipfian distribution, where the alpha value is 0.75.

In Domino, a client (or a replica) periodically sends a probing request to every (other) replica for measuring network delays. The probing interval is 10 ms in our experiments. A replica also sends a heart beat to other replicas every 10 ms, which can be piggybacked on the probing messages. Furthermore, a client (or a replica) estimates its delay to a replica as the 95th percentile delay in its probing results within the last time period, i.e., the window size. The window size is 1 s by default. We have measured Domino's commit latency with different probing intervals (from 5 ms to 100 ms) and window sizes (from 0.1 s to 2.5 s). We find that Domino's commit latency is not sensitive to these parameters in our deployments on Azure. For example, a 5 ms probing interval has a marginally lower

	TX	CA	IA	WA	WY	IL	QC	TRT
VA	27	59	31	67	46	26	38	29
TX	-	33	22	42	23	30	51	43
CA	-	-	41	23	24	48	67	59
IA	-	-	-	36	14	8	32	22
WA	-	-	-	-	21	43	68	57
WY	-	-	-	-	-	24	46	36
IL	-	-	-	-	-	-	23	14
QC	-	-	-	-	-	-	-	11

Table 4.4: Network roundtrip delays (ms) between datacenters in North America

99th percentile commit latency than a 100 ms interval, but the median and 95th percentile commit latency for both probing intervals are nearly identical.

In our experiments, by default, each Domino client introduces no additional delay to increase its request timestamps. For Mencius and EPaxos, a client always sends its requests to the closest replica that is pre-configured based on our network delay measurements.

Our evaluation runs every experiment 10 times. Each experiment lasts 90 s, and we use the results in the middle 60 s. We combine the results from the 10 measurements for CDF and box-and-whisker figures. For figures that have error bars, we use the average result from the measurements, and the error bar is a 95% confidence interval.

4.5.2 Experiments on Microsoft Azure

We deploy Domino, Fast Paxos, Mencius, EPaxos, and Multi-Paxos on Microsoft Azure to evaluate their commit latency and execution latency. Our deployment uses the Standard_D4_v3 VM instance that has 4 vCPUs and 16 GB memory, and an instance runs one client or one server (i.e., one replica).

To evaluate the performance of the protocols under different locations of datacenters in WANs, our experiments consist of two settings, North America (NA) and Globe. NA has 9 datacenters in North America, which are Virginia (VA), Texas (TX), California (CA), Iowa (IA), Washington (WA), Wyoming (WY), Illinois (IL), Quebec City (QC), and Toronto (TRT). Globe has 6 datacenters that are globally distributed, which include VA, WA, Paris (PR), New South Wales (NSW), Singapore (SG), and Hong Kong (HK). Table 4.4 and Table 4.1 (in Section 4.2) show the average network roundtrip latency between datacenters for NA and Globe, respectively.

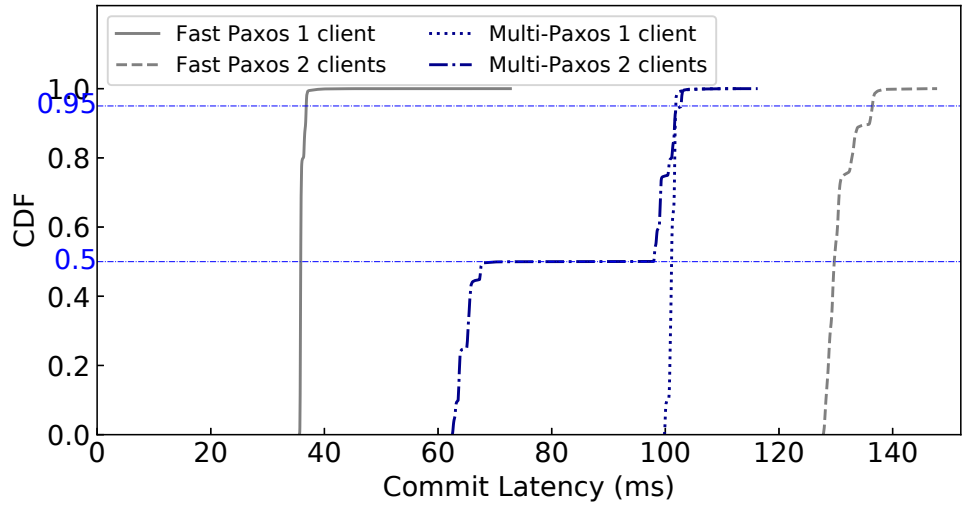


Figure 4.8: Fast Paxos versus Multi-Paxos

We first show that Fast Paxos [66] could experience high latency when there are only a small number of concurrent clients. Because of the high latency of Fast Paxos, we will focus on comparing Domino with Mencius [78], EPaxos [84], and Multi-Paxos [108] in the rest of our evaluation.

Fast Paxos Commit Latency

This section compares Fast Paxos and Multi-Paxos when there are a small number of clients. In this experiment, we use 4 datacenters, WA, VA, QC, and IA from our NA setting. We deploy 3 replicas in WA, VA, and QC, respectively. WA hosts the Fast Paxos coordinator (for the slow path) and the Multi-Paxos leader.

We first run one client in IA to evaluate the commit latency of Fast Paxos and Multi-Paxos. Figure 4.8 shows that Fast Paxos can achieve approximately 65 ms lower median commit latency than Multi-Paxos when there is only one client. This is because Fast Paxos always uses its fast path to commit requests when there are no concurrent clients. We also run two clients in IA and WA, respectively, to compare the two protocols. As shown in Figure 4.8, when there are two concurrent clients, Fast Paxos has higher commit latency than Multi-Paxos. In this experiment, the two clients' requests arrive at replicas in different orders, and Fast Paxos has to use its slow path to commit requests, which causes high latency. In Multi-Paxos, the two clients experience different commit latency because

they have different network delays to the leader. The client that is co-located with the leader in WA has an average of approximate 65 ms commit latency, while the other client in IA sees an average of about 100 ms commit latency.

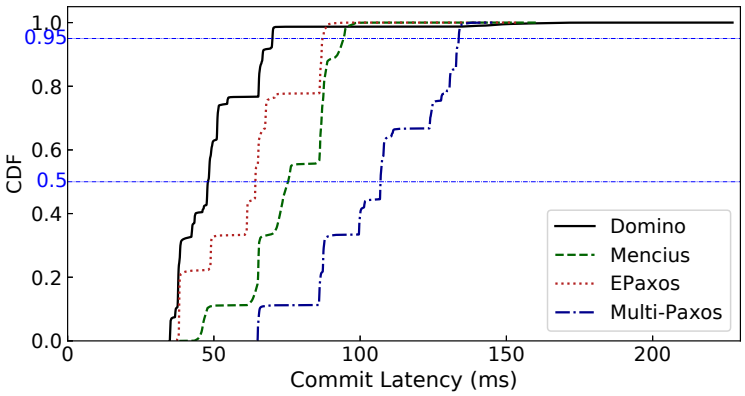
Although Fast Paxos can achieve low latency when there is a single client, our experiments show that Fast Paxos would fall back to its slow path and experience high latency compared to Multi-Paxos even if there are only a small set of concurrent clients in different datacenters. Domino addresses this problem in Fast Paxos, reducing the likelihood that Fast Paxos needs to fallback to the slow path. In the rest of our evaluation, we show that Domino can still achieve low commit latency for concurrent clients in different datacenters.

Domino Commit Latency

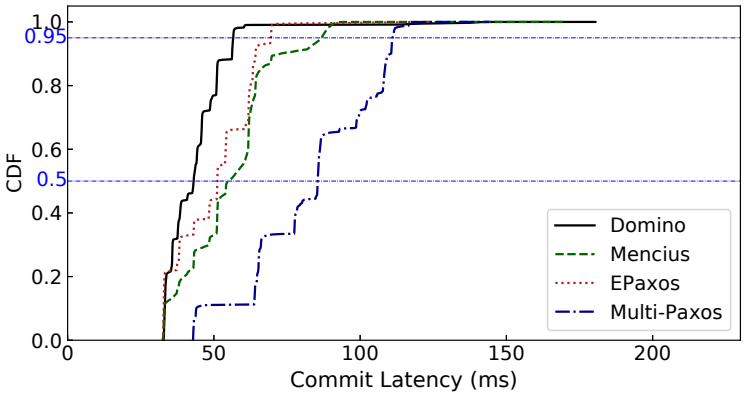
To compare the commit latency of Domino with other protocols, we first use our NA setting with 9 datacenters, and each datacenter runs one client. This setting represents applications that are deployed within a geographical region or a continent.

Figure 4.9 (a) shows the commit latency when there are 3 replicas in WA, VA, and QC, respectively, in which WA hosts the Multi-Paxos leader and the Domino's Fast Paxos coordinator (for the slow path). Domino achieves the lowest commit latency in the median (48 ms) and the 95th percentile (70 ms) compared with EPaxos (64 ms and 87 ms), Mencius (75 ms and 94 ms), and Multi-Paxos (107 ms and 134 ms). This is because 5 out of the 9 clients decide to use DFP, and Domino can commit their requests via the fast path in most cases, which only requires one network roundtrip. The 4 clients in WA, VA, QC, and TRT choose to use DM because they are either co-located with a replica in a datacenter or very close to a replica, and they will have lower commit latency by using DM than DFP in Domino. EPaxos has higher commit latency than Domino because every client has to wait for two network roundtrips to learn that its request is committed. Mencius has higher commit latency than EPaxos because a replica delays committing a request at a consensus instance (not executing yet) until it commits all previous instances. Multi-Paxos has the highest commit latency out of the four protocols since clients have to send their requests to the leader instead of a close replica.

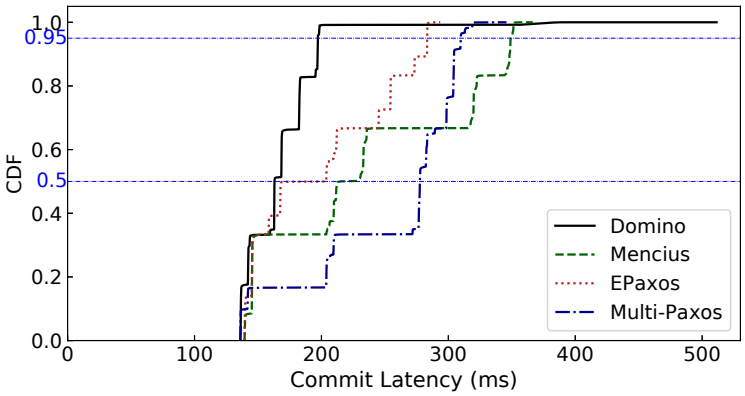
Also, we compare the commit latency of the four protocols when there are 5 replicas. We extend the replica settings by adding two replicas in CA and TX, respectively. Figure 4.9 (b) shows that Domino can still achieve the lowest commit latency at the median and the 95th percentile out of the four protocols. In this setting, 5 clients are co-located with replicas in a datacenter, and they use DM. The other 4 clients use DFP, and Domino can commit their requests via the fast path in most cases. Our experiments show that it



(a) NA with 3 replicas



(b) NA with 5 replicas



(c) Globe with 3 replicas

Figure 4.9: Commit latency on Microsoft Azure

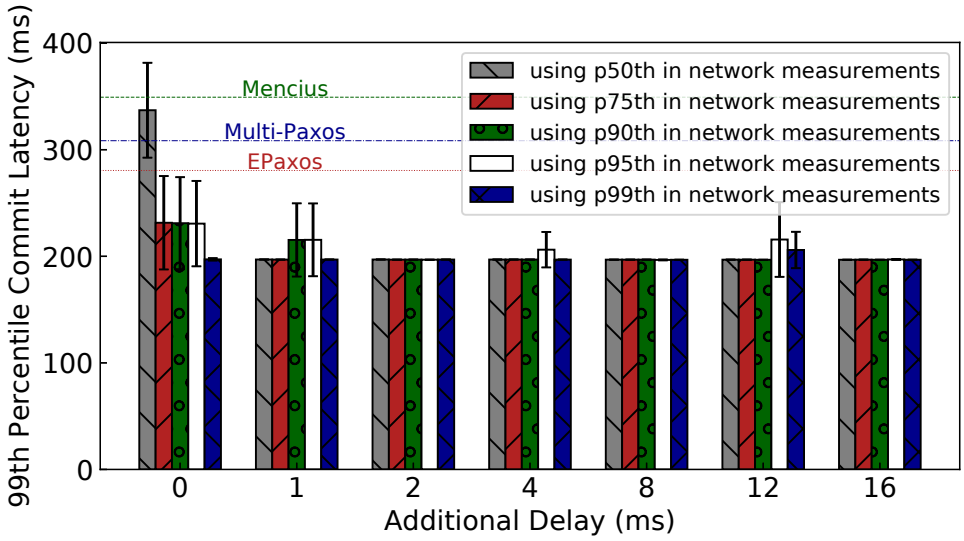


Figure 4.10: The 99th percentile commit latency

is rare that the fast path fails in Domino, where a client's request arrives at replicas later than the predicted time, and Domino has to use a slow path to commit the request.

We further evaluate the four protocols when datacenters are globally distributed. This represents applications that have global users and are deployed in datacenters in different continents. In this experiment, we use the Globe setting with 6 datacenters. Each datacenter runs one client. There are 3 replicas in WA, PR, and NSW, where WA hosts Domino's Fast Paxos coordinator and the Multi-Paxos leader.

Figure 4.9 (c) shows the four protocols' commit latency with our Globe setting. Domino has lower commit latency than the other three protocols from the median to the 95th percentile. For example, Domino achieves approximate 86 ms lower commit latency than EPaxos at the 95th percentile. This is because the 3 clients in VA, SG, and HK choose to use DFP, and Domino can commit their requests via the fast path in most cases. Mencius has higher 95th percentile commit latency than Multi-Paxos because of unbalanced loads across replicas. Domino has similar commit latency to EPaxos below the median. This is because half of the clients are co-located with replicas, and they choose to use DM, which has lower commit latency than DFP. In the rest of our experiments on Azure, we will use the client and replica settings in Figure 4.9 (c).

We also evaluate Domino's commit latency by using different percentile values from network measurements to estimate network delays, and using additional delays to increase

DFP request timestamps. As shown in Figure 4.10, when there is no additional delay, using a higher percentile value from network measurements can achieve lower 99th percentile commit latency. This is because a high percentile delay increases the probability that a request arrives at replicas before its timestamp, and Domino will commit the request via the fast path. The figure also shows that increasing DFP request timestamps by a fixed amount can also reduce the 99th percentile commit latency. Furthermore, when the additional delay is 2 ms and 16 ms, Domino has the same 99th commit latency. This demonstrates that the additional delay will not increase the commit latency when Domino commits a request via the fast path.

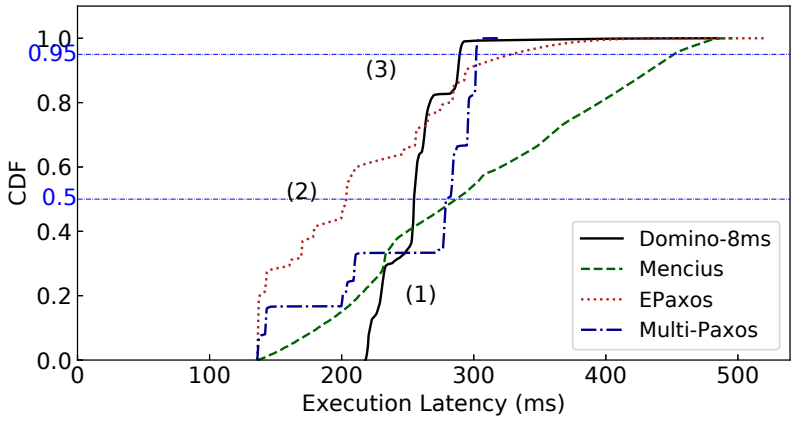
Although increasing timestamps can reduce the probability that Domino uses its slow path, and it introduces no delays to commit requests via the fast path, using an unnecessary large timestamp could introduce delays to the execution latency.

Execution Latency

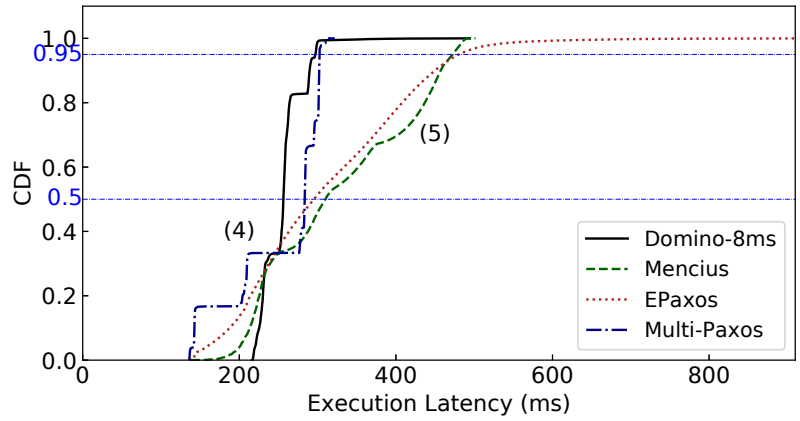
In this experiment, when a Domino client uses DFP, it increases its request timestamp by 8 ms to reduce execution latency, unless specified otherwise. This is because our analysis in Section 4.3.4 shows that the 99th percentile misprediction value for request arrival time at replicas is up to 6.24 ms in this setting.

Figure 4.11 (a) shows the execution latency of different protocols when client requests have few conflicts. As shown in the figure, at label (1), about one third of Domino's requests have higher execution latencies than the other three protocols. This is because a Domino replica executes committed requests following the timestamp order, and Domino may delay executing a DM-committed request after learning the commit of a previous request using DFP. At label (2), EPaxos has the lowest execution latency since it can execute requests out of order when request conflicts are rare. Finally, at label (3), Domino has the lowest 95th execution latency among the four protocols because Domino has a high success rate of committing requests via the fast path. Correspondingly, Mencius experiences a high 95th execution latency because of its execution delay for concurrent requests. EPaxos has higher 95th execution latency than Domino and Multi-Paxos because of its delays for executing conflicting requests. Multi-Paxos' execution latency largely depends on its commit latency, and it experiences higher 95th execution latency than Domino.

We further evaluate the execution latency of the four protocols by increasing the amount of contention between requests. With Zipfian alpha increasing from 0.75 to 0.95, Figure 4.11 (b) shows that, at label (4), EPaxos experiences significantly higher execution latency. Request contention has no effect on Domino and Multi-Paxos because they ex-



(a) Zipfian alpha = 0.75



(b) Zipfian alpha = 0.95

Figure 4.11: Execution latency on Azure

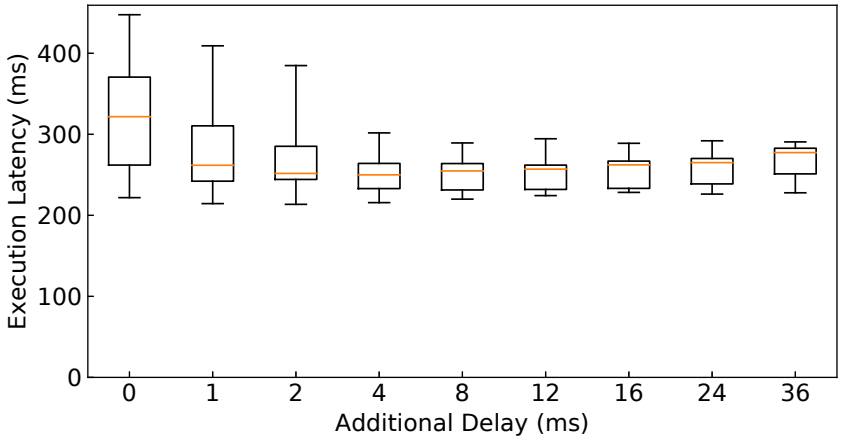


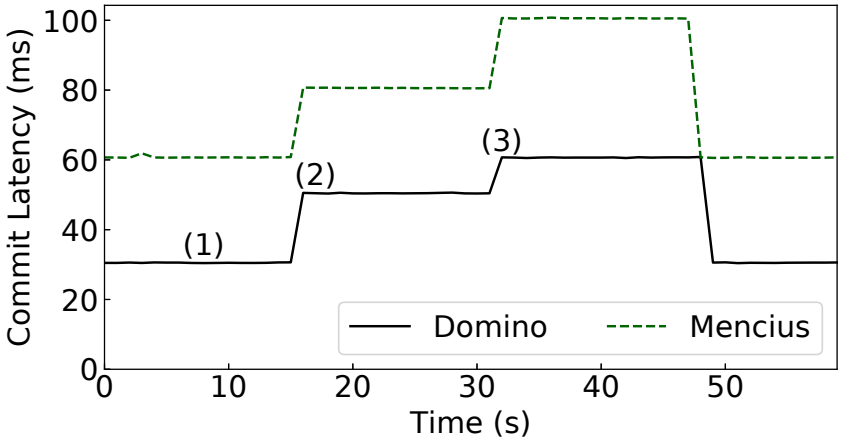
Figure 4.12: Impact of additional delays (for increasing DFP request timestamps) on execution latency

ecute committed requests in the log order. As shown at label (5), the contention has a small effect on Mencius because of its out-of-order execution.

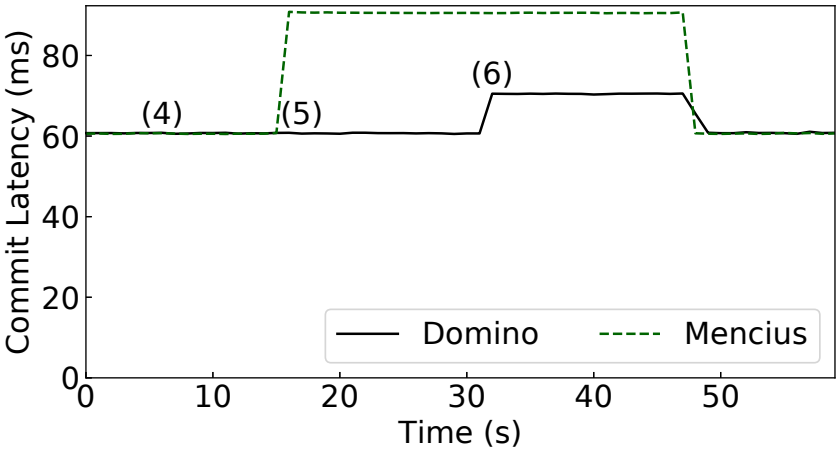
Furthermore, we evaluate the impact of introducing additional delays (for increasing DFP request timestamps) on Domino's execution latency, as shown in Figure 4.12. In the figure, the middle line in a box is the median execution latency, and the whiskers show the 5th and 95th percentile execution latency, respectively. When there is no additional delay, Domino would experience higher execution latency than using small additional delays to increase DFP request timestamps. This is because a DFP request may arrive at a replica later than the estimated time, and DFP may use the slow path to commit the request. As Domino interleaves DFP and DM log positions, a DFP slow-path committed request will delay the execution of its following requests in timestamp order. Using a small delay to increase DFP request timestamps will significantly reduce the execution latency because it decreases the likelihood that Domino uses the slow path. However, using a large delay will increase the overall execution latency. As shown in the figure, by increasing the additional delay from 8 ms to 36 ms, the median execution latency increases by about 23 ms.

4.5.3 Microbenchmark Experiments

As the network environment on Microsoft Azure is relatively stable, we use microbenchmark experiments with emulated network delays to evaluate how Domino responds to network delay variance. We run our microbenchmark experiments in our local computer cluster,



(a) Network delays change between a client and replica



(b) Network delays change between replicas

Figure 4.13: Change of network delays

where each machine has 12 CPU cores and 64 GB memory. In our experiments, we use the Linux traffic control utility to emulate artificial network delays between clients and replicas.

We evaluate how a Domino client adaptively chooses between DM and DFP based on its network measurements in order to achieve low commit latency. In this experiment, there are three replicas and one client, and we emulate a network environment where the network delay between a client and a replica (or between replicas) could significantly change, e.g., due to a routing change. To improve the clarity of the figures, the client only sends one request per second. Experiments at higher request rates show similar results. Also, we only show Mencius in our figures as other protocols have similar performance to Mencius in this specific setting.

We first set the network roundtrip delay to be 30 ms between any two nodes. There is one replica, R, which is the pre-assigned coordinator for the client in Mencius. As shown in Figure 4.13 (a), in the beginning, at label (1), Domino has lower commit latency than Mencius because the client chooses to use DFP. At about 15 s, i.e., label (2), the network roundtrip delay between the client and R changes to 50 ms. In this case, the latency of both Domino and Mencius increases. Mencius could achieve lower latency (60 ms) than the 80 ms latency in the figure if it could detect the delay change and use a different coordinator. The Domino client keeps using DFP as it has lower latency (50 ms) than using DM. At label (3), the roundtrip delay between the client and R increases to 70 ms. The Domino client begins to use DM, which has lower commit latency (60 ms) than using DFP (70 ms). It uses a DM leader other than R in this case.

Figure 4.13 (b) shows that Domino also responds to network delay changes between replicas. We change the initial settings such that the network roundtrip delay between the client and a replica (other than R) is 70 ms. In this setting, at label (4) in the figure, Domino and Mencius have the same commit latency in the beginning as DM is preferable to DFP. At label (5), the network roundtrip delay between R and both of the other two replicas, M and N, increases to 60 ms. Domino has lower latency than Mencius since the client uses M or N as the DM leader. Later at label (6), the roundtrip delay between M and N increases to 60 ms. Domino begins to use DFP which has lower commit latency than using DM.

4.5.4 Experiments within a Cluster

We evaluate the throughput of Domino by running experiments within our private computer cluster due to the expenses of using Microsoft Azure. In the cluster, each machine

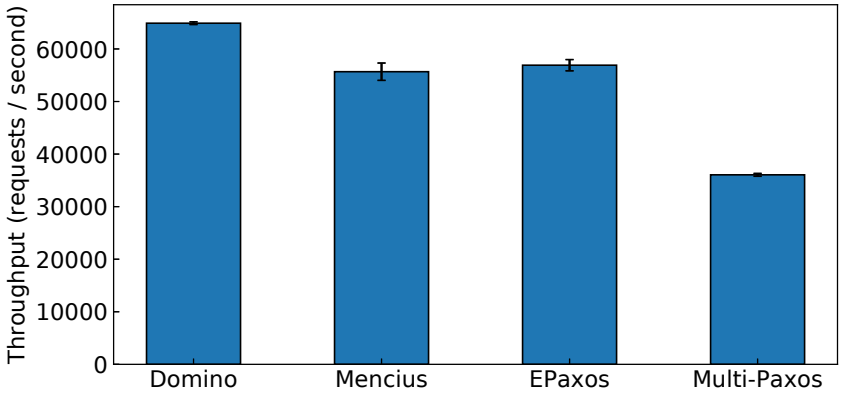


Figure 4.14: Peak throughput with 3 replicas

has 12 CPU cores and 64 GB memory, and the machines are connected through a 1 Gbps network switches. In our experiments, each replica runs on a single machine.

Figure 4.14 shows the peak throughput of Domino, Mencius, EPaxos, and Multi-Paxos, when there are three replicas. Domino can achieve a peak commit throughput of about 65K requests per second (rps), which is comparable to Mencius (56K rps) and EPaxos (57K rps). Domino has higher throughput than Mencius because our implementation has more parallelism between its I/O operations and computation. Multi-Paxos has the lowest peak throughput (36K rps) among the four protocols because all client requests have to go to the leader.

4.6 Chapter Summary

This chapter introduces Domino, a low-latency state machine replication protocol for wide-area networks. Domino uses network measurements to deterministically order client requests across replicas, and it makes Fast Paxos-like consensus protocol commit requests in one network roundtrip in the common case. This approach makes Fast Paxos practical for building state machine replication in WANs, which significantly reduces the likelihood that Fast Paxos falls back to its slow path when there are concurrent clients.

Furthermore, no single consensus protocol can always have the lowest commit latency because of network geometry. Domino executes a Fast Paxos-like consensus and a leader-based consensus in parallel under the same deployment. A Domino client uses network measurement data to independently choose which to use in order to achieve low commit

latency. Our experiments on Microsoft Azure show that Domino achieves lower commit latency than leader-based protocols, such as Mencius and EPaxos.

Chapter 5

SpecRPC

Many distributed applications sequentially execute network I/O operations to complete a task because there are data dependencies between these operations. Speculation is one way to reduce the latency of executing the dependent operations. By predicting the result of a network I/O operation, an application can speculatively execute subsequent operations. If the prediction is correct, dependent operations execute in parallel, which reduces latency compared with the sequential execution of the operations. Speculation has been used in many applications, and it is a powerful technique for reducing latency. However, it can introduce a significant amount of complexity to the application development. Adding speculative execution support to an application requires a large investment in time and resources, and the implementation of handling incorrect predictions is error-prone, especially in a complex distributed system.

In this chapter, I introduce SpecRPC, a framework that facilitates applications to perform speculative RPCs in order to reduce latencies. SpecRPC aims to reduce the barrier of implementing speculation in distributed systems, allowing for the pervasive use of speculation to reduce application latency. In the rest of this chapter, I will first present the design patterns of SpecRPC, and then I will describe SpecRPC's architecture in details. Thereafter, I will demonstrate applications that will benefit from using SpecRPC, and I will show the experimental results of using SpecRPC to reduce the latency in a transaction processing protocol. Finally, I will disscuss about how SpecRPC deals with the common issues in speculation techniques, and I will summarize the research work.

5.1 Design Patterns

SpecRPC is an asynchronous RPC framework for Java where remote calls return immediately to the caller. A dependent operation is specified as a callback function that executes after the RPC completes. A callback function implicitly accepts the RPC's return value as a function parameter. The callback can issue additional RPCs with callbacks, which allows clients to execute a sequence of dependent RPCs by specifying a chain of callback functions.

An asynchronous RPC framework offers additional opportunities to perform operations in parallel compared to a synchronous framework. For example, clients can continue execution on non-dependent operations after issuing an RPC request. Unlike many other asynchronous RPC frameworks that execute dependent RPCs and callbacks in a serial order, SpecRPC can parallelize the execution of these dependent operations via speculation. More importantly, for the purpose of speculative execution (SE), sequential operations are completely specified by a callback function chain. Therefore, by requiring that each callback function is implemented as a method in a callback object that can only modify the object's data[1], SpecRPC can encapsulate speculative results inside a collection of callback objects. This programming model allows SpecRPC to ensure that speculative results are not revealed to the rest of the application, and parallel speculations are isolated from each other.

Figure 5.1 illustrates an application that uses SpecRPC to parallelize client-side and server-side computation, where the client predicts the server's computation result. The *Math* class in Figure 5.1 (a) is provided by the RPC server and exposes the *plus* method to remote callers. The server's *main* method specifies the necessary boilerplate code to register the *plus* method, allowing remote hosts to call this method. Instead of accepting a *Math* RPC object, the *register* method accepts a factory object, which is used by the framework to create a new *Math* object for each RPC request.

Inside the *main* method in Figure 5.1 (b), the client binds the remote *plus* method to an RPC stub, and issues an RPC by executing the stub's *call* method. The *call* method takes as parameters a list of predicted RPC return values and a callback factory. Upon receiving a response from the server, the client uses the callback factory to generate an instance of *IncCB*, and executes the callback with the return value from *plus* as a parameter. Using factories enables the framework to speculate multiple times with different predicted values,

[1]This is an advisory programming model for correctness rather than a mandatory design pattern. Applications can choose to modify data outside of callback objects if they are certain that it will not affect the correctness of the application.

```
1 public class Math implements SpecRpcHost { // RPC implementation
2  public Integer plus(Integer a, Integer b) {
3   return a + b;
4  }
5  ... // Defines other RPCs
6 }
7 public class MathFactory implements SpecRpcHostFactory { // RPC factory
8  public SpecRpcHost getRpcHostObject() {
9   return new Math();
10 }
11 public String getRpcHostClassName() {
12  return Math.class.getName(); // Returns "Math"
13 }
14 }
15 public class Server { // Server implementation
16 public static void main(String args[]) {
17  SpecRpcServer rpcServer = new SpecRpcServer();
18  rpcServer.initServer("./server.config");
19  // Registers an RPC with its name, factory, return value type, and parameter types
20  rpcServer.register("plus", new MathFactory(),
21    Integer.class, Integer.class, Integer.class);
22  rpcServer.execute(); // Starts the RPC server
23 }
24 }
```

(a) Server-side code

```
1 public class IncCB implements SpecRpcCallback { // Callback implementation
2  public Object run(Object rpcResult) {
3   return (Integer)rpcResult + 1;
4  }
5 }
6 public class CBFactory implements SpecRpcCallbackFactory { // Callback factory
7  public SpecRpcCallback createCallback() {
8   return new IncCB();
9  }
10 }
11 public class Client { // Client implementation
12  public static void main(String args[]) {
13   SpecRpcClient.initClient("./client.config");
14   // Binds an RPC with its class name, method name, return value type, and parameter types
15   RpcSignature plus = new RpcSignature("Math", "plus",
16     Integer.class, Integer.class, Integer.class);
17   SpecRpcClientStub stub = SpecRpcClient.bind("localhost", plus);
18   List preds = Arrays.asList(3); // Predicts plus(1,2)
19   SpecRpcFuture future = stub.call(preds, new CBFactory(), 1, 2); // Issues RPC
20   future.getResult(); // Callback result is 4
21  }
22 }
```

(b) Client-side code

Figure 5.1: An example illustrating the code for a simple SpecRPC application

where each SE creates a different RPC or callback object. By specifying client-predicted return values, SpecRPC allows SE to begin even before the RPC has been sent to the server.

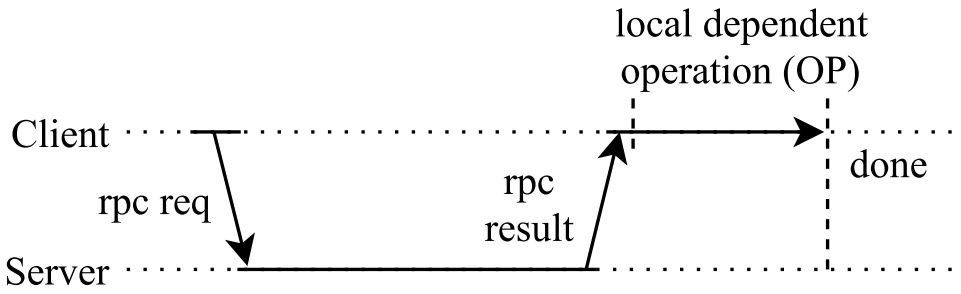
A SpecRPC call immediately returns a future object that eventually acquires the return value of the callback method from the final, non-speculative callback object. The caller can retrieve this value by calling *getResult* on the future object, which blocks until the return value is available. The framework ensures that the method returns a non-speculative result. In the example from Figure 5.1, the client will block until the future receives *4* from the callback.

5.1.1 Single-Level Speculation

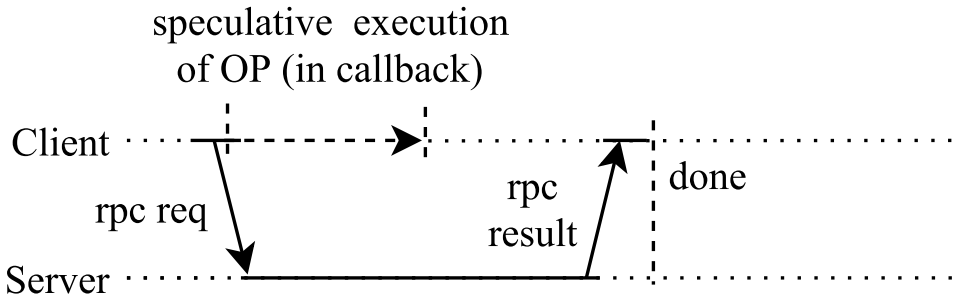
In a traditional RPC framework, operations that depend on an RPC's return value must wait until the RPC completes. This is illustrated in Figure 5.2 (a) where the local operation can only begin after receiving the RPC return value. However, in some applications, clients can often predict RPC results. For example, an application may perform an RPC repeatedly with the same parameters and, in most cases, receive the same return value. The client can use a client-side cache to predict the RPC result. Using SpecRPC, the client can use this prediction to speculatively execute the dependent operations, specified as callback objects, immediately after invoking the RPC. As shown in Figure 5.2 (b), the execution times for the RPC and its dependent operations overlap. We call this client-side speculation, where the client predicts RPC results.

In addition to client-side speculation, SpecRPC also supports server-side speculation in which the server predicts the RPC's return value before it completes its execution of the RPC function. This is useful for RPC functions that execute slowly, but their results can be accurately predicted after a small amount of preliminary computation. In Figure 5.2 (c), the client must wait until it receives the server's prediction before it can speculatively execute its dependent operations. The server can return a prediction by calling *specReturn* any time during the RPC execution. Multiple predictions can be made by both the client and the server. Each prediction creates a new callback object that executes independently.

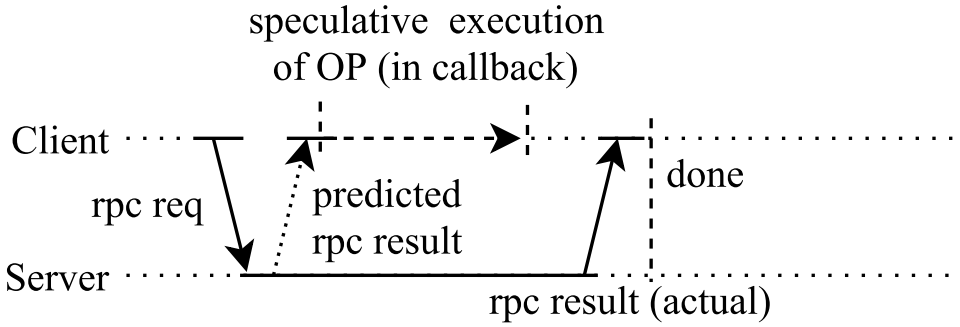
For both client-side and server-side speculation, the framework can determine if the return value prediction is correct after the client receives the RPC's actual return value. In the case where the prediction was correct, the result from the SE that is performed via callback objects can be returned to the application. Otherwise, the framework will dispose of the speculative results and re-execute the dependent operations using the actual RPC



(a) Traditional RPC



(b) SpecRPC with client-side prediction



(c) SpecRPC with server-side prediction

Figure 5.2: An example of single-level speculation

return value. We will explain how SpecRPC manages predictions and speculative results in Section 5.2.

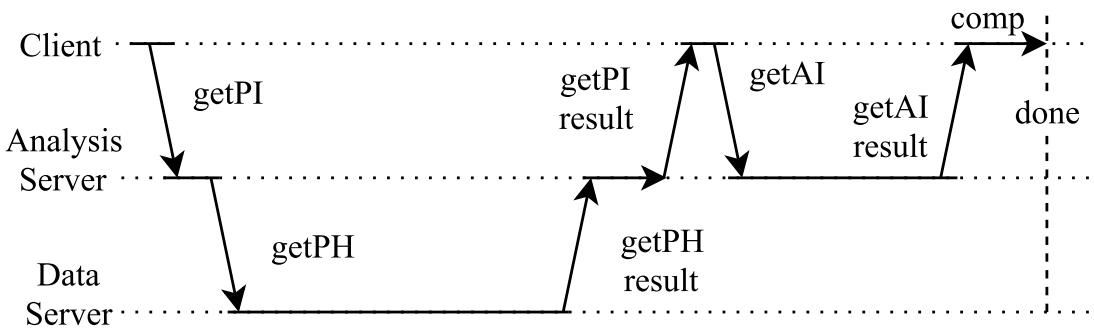
5.1.2 Multi-Level Speculation

In order to provide significant performance benefits for workloads with long chains of sequential operations, a speculation framework must support having multiple dependent operations execute concurrently. This allows an application to perform SEs that depend on the correctness of multiple predictions. We call an SE that depends on more than one prediction a *multi-level speculation* (MLS).

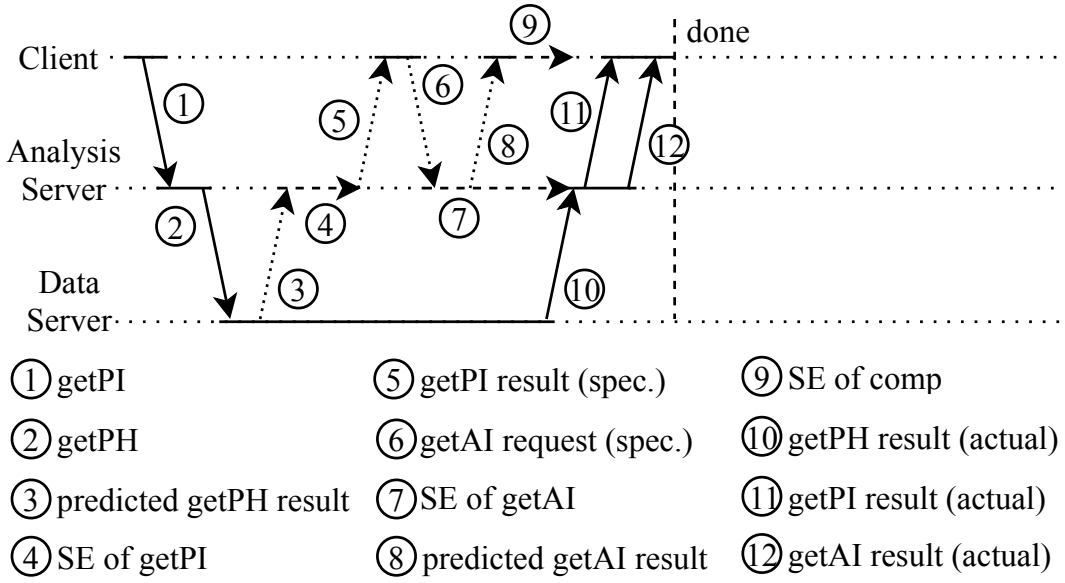
Figure 5.3 shows a sample sequence of operations that could benefit from using MLS. In this example, an *analysis server* (AS) provides a data analytics service, a *data server* (DS) manages user data, and a client is interested in making a purchase based on both individual user information and aggregate information from a specific userbase. The client first retrieves the purchasing interests (PIs) of a specific user from AS by invoking an RPC, *getPI*. To compute the user's PIs, AS issues an RPC, *getPH*, to DS for the user's purchase history (PH). Once *getPI* completes, the client invokes another RPC, *getAI*, to AS to retrieve aggregate information (AI) from the userbase that shares the same PIs as the user. This AI is generated in real-time by AS. Finally, once *getAI* completes, the client performs additional local computation, *comp*, before ending its execution.

With speculation, parts of the above three RPCs and the client's local computation can execute in parallel. To service *getPH*, DS must retrieve the PH pertaining to the user specified in the request. Although the data may be available locally, additional synchronization delays may be introduced if DS is not the primary replica for the data and linearizable consistency is required. However, DS can send a speculative response using its local data to allow the caller to continue its execution without waiting for the synchronization to complete. Similarly, when servicing *getAI*, AS may be able to send a speculative response back to the client before it finishes generating the requested AI. The speculative response may be taken from the cached response of a previous request either for the same userbase, or for a related userbase with a similar PI.

Figure 5.3 (b) illustrates that, by predicating the result of *getPH*, AS can speculatively return the result of *getPI*. This will cause the subsequent operations to be speculatively executed in parallel. Figure 5.3 (b) also shows that, by speculatively executing *getAI* and predicting its result, *comp* can execute in parallel with both *getPH* and *getAI*. This is an MLS example because the SE of *comp* depends on more than one prediction.



(a) Traditional RPC



(b) SpecRPC

Figure 5.3: An example of multi-level speculation

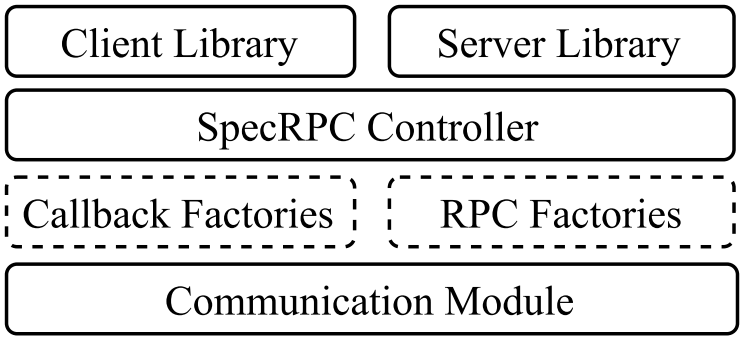


Figure 5.4: SpecRPC architecture

The previous example demonstrates the need for a speculation framework to have each speculative RPC transitively depend on all of the predicted return values that its caller relies on. It also illustrates the challenge in tracking dependency information across RPCs.

5.2 Architecture

The SpecRPC architecture consists of four layers, as shown in Figure 5.4, client and server libraries register functions for remote access, expose the functions based on their signatures, and asynchronously issue remote RPCs. The SpecRPC controller manages speculative dependencies, uses callback and RPC factories to create new callback and RPC objects, and provides isolation between concurrent callbacks and RPCs. User-provided callback and RPC factories create new callback and RPC instances to handle RPC results and requests. Each callback or RPC instance has an object, *specObj*, which encapsulates the instance's speculative state. An RPC call inside a callback or RPC inherits the caller's speculative state through the *specObj*. The communication module manages connections between clients and server.

In the following sections, we describe how the different components work together to manage speculative dependencies across multiple nodes, handle incorrect predictions, and ensure that the final result is equivalent to what a traditional RPC framework would return.

5.2.1 Speculative State

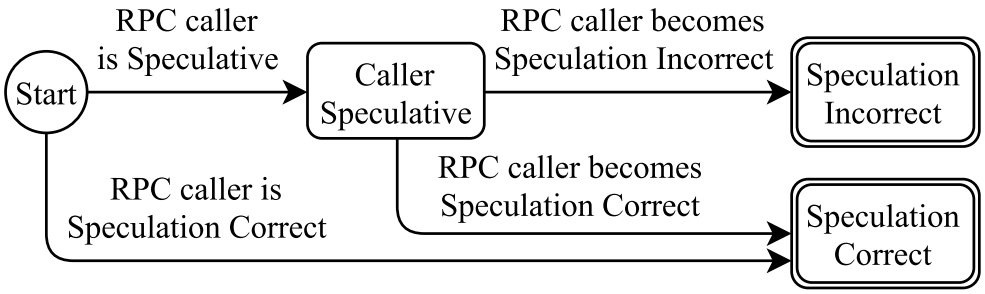
In SpecRPC, computation is performed entirely within callback and RPC objects. A callback object is created when a client receives an RPC response, and an RPC object

is created when a server receives an RPC request. A callback performs SE if it receives a predicted RPC response instead of an actual RPC response. If a callback issues an RPC request while it is performing SE, the RPC object created from the request will also speculatively execute its computation. When the actual RPC response arrives, the speculatively executed callback and its dependents will be discarded if the prediction was incorrect. Otherwise, they will be marked as actual execution.

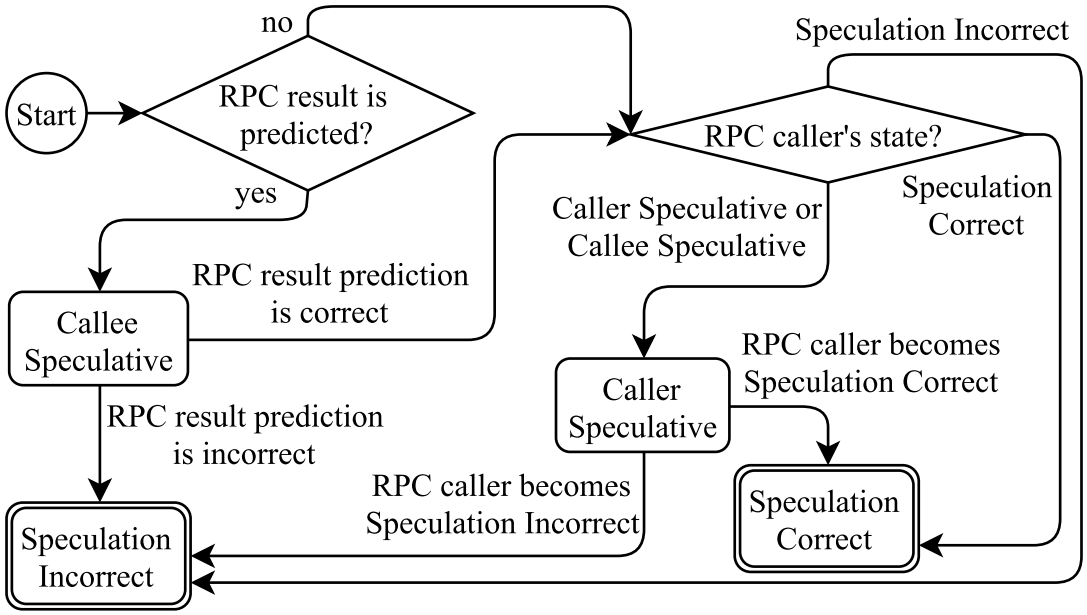
To distinguish between actual and speculative execution, each callback and RPC object contains a *speculative state* that describes its speculation status and dependency information. This state is encapsulated as a *specObj*.

An RPC's speculative state can be one of the following: *caller speculative*, *speculation correct*, and *speculation incorrect*, where the last two states are terminal states. Figure 5.5(a) illustrates the state transitions for RPC objects. An RPC is in *speculation correct* state if the caller, which can be a client, an RPC object or a callback object, is not dependent on any predicted values. This is always the case when the caller is a client because a client's RPC request cannot be dependent on a predicted value. It is also the case when the caller is an RPC or callback object that is in *speculation correct* state. An RPC is in *caller speculative* state if its caller is dependent on a predicted value. This is equivalent to the caller being in a non-terminal state. Finally, an RPC transitions from the *caller speculative* state to the *speculation incorrect* state if its caller transitions to the *speculation incorrect* state.

Each callback is associated with an RPC and executes with the RPC's return value. Multiple callbacks can be associated with the same RPC because of multiple predictions for the return value. A callback can have one of the following speculative states: *caller speculative*, *callee speculative*, *speculation correct*, and *speculation incorrect*. Figure 5.5(b) illustrates the state transitions for callback objects. A callback is in *speculation correct* state if it receives a non-predicted return value from its RPC and the RPC is in *speculation correct* state. A callback is in *callee speculative* state if it executes with a predicted return value of its RPC. Upon receiving the actual return value, the callback transitions to the *speculation correct* or *speculation incorrect* state depending on the prediction, or transitions to *caller speculative* state if its RPC's caller is in a non-terminal state, i.e., either caller speculative or callee speculative. Finally, a callback in *caller speculative* state transitions to a terminal state once its RPC's caller transitions to a terminal state.



(a) State transitions of an RPC object.



(b) State transitions of a callback object.

Figure 5.5: State transitions

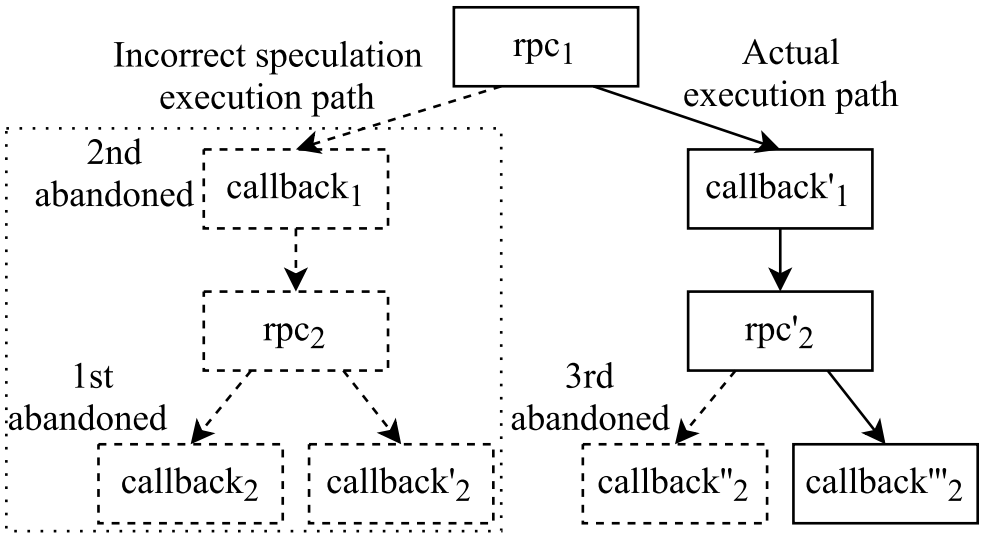


Figure 5.6: An example of a dependency tree

5.2.2 Managing Dependencies

In single-level speculation, a speculative callback depends on the correctness of a single prediction. In order to keep track of dependencies, SpecRPC would only need to maintain a mapping between predictions and their corresponding speculative callbacks. Dependency management becomes more challenging in multi-level speculation as callbacks and RPCs can be dependent on multiple predictions. These dependencies are modeled as a tree with each RPC and callback object representing a node. The root in a dependency tree is the first RPC object that is issued by the user application. The root is always in the *speculation correct* state. A callback object created on a predicted return value is a child node of the RPC in the dependency tree.

A node's speculative state depends on that of its parent node (if any). Each node only needs to track its parent node's state transition and pass the state change of itself to its child nodes. SEs that are based on the same predicted return value of an RPC form a subtree under the RPC. Also, a path from the root to a leaf in the tree links dependent non-speculative and speculative execution. There is only one path standing for the actually non-speculative execution.

Figure 5.6 shows an example of a dependency tree in a bad scenario where predicted RPC responses are always incorrect. In this example, a client performs two dependent RPCs, rpc_1 and rpc_2, followed by a local operation that is executed in $callback_2$. For each

RPC invocation, the client receives an incorrect prediction from the server. After receiving a prediction result for rpc_1, $callback_1$ is created to perform rpc_2. Therefore, rpc_2 is a child node of $callback_1$ which in turn is a child node of rpc_1. As rpc_2 executes, it also returns a predicted result which creates $callback_2$ to run the dependent local operation. In this example, rpc_2 finishes before rpc_1 even though it starts after rpc_1. When rpc_2 finishes, it returns an actual result that is different than its previous predicted result. Therefore, a new callback object $callback'_2$ is created and $callback_2$ is abandoned. Later, when rpc_1 finishes with an actual result that is different than its previous predicted result, the whole subtree of $callback_1$ is abandoned and a new $callback'_1$ is created to invoke rpc'_2. The predicted result of rpc'_2 creates $callback''_2$, which again will be abandoned when actual result of rpc'_2 is different from its predicted result. Finally, with the completion of rpc'_2, $callback'''_2$ is created to finish the remaining execution. Even with three mispredictions, the client only sees the actual execution path from rpc_1 to $callback'''_2$.

In both single and multi-level speculations, SE includes local operations and RPCs that will be executed remotely. As a result, the dependencies between predictions and SEs span multiple nodes. SpecRPC uses dedicated state-change messages to propagate state change events between remote callback and RPC objects. This is discussed in Section 5.2.4.

5.2.3 Handling Incorrect Predictions

Upon receiving the actual result for an RPC, SpecRPC evaluates the accuracy of previous predictions of that result, and retains callbacks based on correct predictions while setting callbacks based on incorrect predictions to the *speculation incorrect* (SI) state. Any callback or RPC object that depends on an object in SI state must also be set to SI state. Objects in SI state are discarded, and their computations are abandoned. This can be done safely without requiring data rollback since a callback or RPC in SpecRPC should only modify the fields in its associated object and should not have any side-effects. This advisory requirement is not enforced as it is up to application to decide the scope of side-effects caused by modifying data outside of a callback or RPC object.

Immediately terminating a discarded callback or RPC may require interrupting its computation, which can be difficult to perform cleanly in a language without non-local exception passing where a thread can raise an exception in a different running thread. To avoid this problem, SpecRPC allows callbacks and RPCs in SI state to finish execution before being abandoned. However, SpecRPC immediately terminates these callbacks and RPCs if they attempt to perform further speculative operations via SpecRPC, such as invoking a new RPC, returning a prediction to the client, or blocking on an operation that will generate visible output (see Section 5.2.5).

In the case where none of the previous predictions of an RPC result were correct, a new callback is created using the RPC's actual result. This ensures that forward progress is made even in the absence of an accurate predictor.

5.2.4 Propagation of Speculative State

The speculative state of a callback or RPC object depends on the state of the caller. Therefore, when a callback or RPC issues an RPC request, the caller's speculative state is sent alongside the RPC's parameters. Similarly, each RPC response contains the speculative state of the RPC and a field that specifies whether it is returning a predicted result or an actual result. The framework uses this information to create a callback object in the correct speculative state.

In multi-level speculation, when the caller of an RPC transitions to a *speculation correct* or *speculation incorrect* state from a non-terminal speculative state, both the state of the remote RPC instance and the corresponding local callback must be updated. To notify the RPC instance on the remote node, the caller sends a dedicated message indicating its new speculative state. Both the RPC and callback perform their own speculative state change based on the caller's new speculative state. This change is further propagated if another RPC is invoked by either the callback or RPC.

5.2.5 Implementation

The SpecRPC framework consists of approximately 3000 lines of Java code. The source code is available online at [11]. In SpecRPC, RPCs are registered by servers as signatures containing an RPC name, a return type, parameters and a server address. RPC signatures are stored in a file that is synchronized between the servers and clients using third-party tools, such as ZooKeeper [47].

Tracking Dependencies

SpecRPC tracks the dependencies between speculative and non-speculative executions by mapping a callback or RPC object to its parent node in a dependency tree. Instead of implementing the dependency tree as a centralized data structure, each node only tracks its child nodes. When a node's speculative state changes, SpecRPC propagates the changes only to its child nodes.

Applications using SpecRPC do not need to explicitly track speculation-related dependencies or inform the framework of what it depends on. When a prediction is incorrect, SpecRPC will discard all of the speculative callback and RPC objects that depend on the prediction.

Preventing Side-Effects

SE should not result in output or state changes that are irrevocable. Therefore, SpecRPC recommends that callbacks and RPCs only modify the fields in their objects, and not have any side-effects. SpecRPC's factory design pattern creates a new object when it executes a callback or RPC, and it stores speculative states inside that object. This isolates parallel SEs, which allows an application to make multiple predictions.

An application can optionally install a *rollback* function for mis-speculation in a callback or RPC. The SpecRPC framework will execute the rollback function before discarding incorrect states. This enables an application to extend its speculative states beyond the fields inside a callback or RPC object. For example, an application can store speculative states in a local database and issue a rollback for a mis-speculation.

In scenarios where it is impossible to avoid irrevocable changes or output in SE, the SpecRPC framework provides *specBlock*, a method that causes a speculative callback or RPC to block until it is in a non-speculative state. An application can call *specBlock* just before operations that will cause side-effects. Once the speculation is determined to be correct, SpecRPC will continue the application's execution. If the speculation is incorrect, the *specBlock* function will throw a mis-speculation exception.

5.3 Applications

In this section, we describe how we used SpecRPC to implement a speculation-enabled version of Replicated Commit [77], a distributed transaction commit protocol for geo-replicated database systems. Also, we perform a theoretical analysis on the expected speedup from using speculative execution (SE) on another application, a multi-objective optimizer.

5.3.1 Replicated Commit

Replicated Commit (RC) [77] is a distributed transaction commit protocol for geo-replicated database systems. In a geo-replicated system, a transaction's completion time largely depends on the number of wide-area network roundtrips that the transaction requires to complete.

RC introduces a commit protocol that only requires one wide-area network roundtrip to complete both 2PC and consensus among replicas across datacenters. However, to achieve this, local read operations have to be replaced with quorum reads across multiple datacenters. Each quorum read introduces one wide-area network roundtrip. Writes are not affected as they are buffered until the transaction commits. RC's evaluation shows that, as the number of dependent reads increases, the transaction completion time is quickly dominated by read latency.

SE can parallelize the execution of dependent reads in RC in order to reduce the overall completion time of a transaction. This is possible because the read results from the first responding quorum member are often the same as the final quorum results. Therefore, we can use the first response to speculatively execute the next read operation. In RC, because data is fully replicated in every deployed datacenter, the first responding member will always be from the local datacenter, and its response will return almost immediately.

In the original RC protocol, two-phase locking is used to isolate concurrent transactions that have conflicts, and writes always preempt read locks that are held by conflicting transactions in order to prevent deadlocks. In our implementation of RC with speculation, speculative reads will acquire read locks in the same way as non-speculative reads. When speculative reads are discarded due to incorrect speculations, there is no need to release the acquired read locks since the read locks will not block any writes in RC. Note that we only perform SE for quorum reads before starting the commit protocol.Before calling commit on a transaction, an RC client will issue a specBlock (see Section 5.2.5) to wait until all quorum reads become non-speculative. We have implemented a fully working prototype of the SpecRPC version of RC, and we evaluate its performance in Section 5.4.

5.3.2 Multi-Objective Optimizer

Many scientific computing problems require solving optimization problems (OPs) with multiple objectives. A common approach for solving multi-objective problems is to construct them as a series of dependent OPs in which the output of one OP serves as an input to the next OP. Each of these intermediate OPs can be solved using an optimizer such

as CPLEX [48] and Gurobi [45]. Therefore, the completion time is largely dependent on the performance of the optimizer, and the number of OPs in the series. Although most optimizers benefit from additional CPUs, their scalability is limited. Most do not achieve additional speedup beyond 16 processors [57].

SE can leverage additional CPUs to reduce computation time of dependent OPs by overlapping their computation. With SpecRPC, each OP can be registered as an RPC function, and the OPs can be deployed on a group of server nodes. To execute an OP, a client node issues an RPC to a server. Before the optimization is complete, the server can return its current best solution to the client. The client can use this result to issue an RPC to another server in order to speculatively execute the next OP. The current best solution serves as a prediction for the final result from the optimizer. If the prediction is correct, where correctness is based on the user's equivalence definition, there will be a reduction in the total completion time. The correctness probability depends on the amount of time the function was allowed to run before the current best solution was retrieved. This is because the more time the optimizer is given to run, the more likely that it has found the optimal result.

Assuming that there are n dependent OPs (stages), we define S_{lat} as the speedup of using SE to complete the n stages compared to sequentially executing them with the same total number of CPUs. We also define T_{new} and T_{old} as the expected completion time with and without SE, respectively. Therefore, we have $S_{lat} = \frac{T_{old}}{T_{new}}$.

In this analysis, we assume there are $n * N$ total CPUs. We define the amount of time it takes for stage i's optimizer to complete as: $T_i = g_i(m)$, where m is the number of CPUs, and g_i is a monotonically increasing function of m. When m is above a threshold, the increase of T_i is negligible. We also define the prediction correctness percentage at stage i as: $P_i = f_i(t_i)$, where t_i is the amount of time that stage i's optimizer executes before the best current solution was retrieved. We denote $E_{i,j}$ as the expected completion time of executing all stages from i to j. $E_{i,n}$ can be recursively calculated as follows:

$$\begin{cases} E_{n,n} = T_n \\ E_{i,n} = P_i * (t_i + E_{i+1,n}) + (1 - P_i) * (T_i + E_{i+1,n}) \end{cases} \tag{5.1}$$

where $1 \leq i < n$. The last stage's completion time is always T_n, and no prediction occurs at this stage.

By solving the recursion in Equation (5.1), we have T_{new}:

$$T_{new} = E_{1,n} = \sum_{i=1}^{n-1} [P_i * (t_i - T_i) + T_i] + T_n \tag{5.2}$$

where $0 \leq t_i \leq T_i$, and $T_i = g_i(N)$ since, with SE, it is possible that all n stages run in parallel, so each stage can only use N CPUs.

Without SE, each stage can use the total $n * N$ CPUs, so we have T_{old}:

$$T_{old} = \sum_{j=1}^{n} T_j = \sum_{j=1}^{n} g_j(n * N) \tag{5.3}$$

With a fixed N, we can determine the set of t_is that maximize the total speedup, S_{lat}.

We illustrate our model with a two-stage example where the stages have the same completion time T. In this example, there are enough CPUs such that using N and $2N$ CPUs at each stage will achieve the same completion time, i.e., $|g(N) - g(2N)| < \epsilon$, where ϵ is negligible. We also assume that the prediction correctness percentage at the first stage can be described as a cumulative distribution, $P = 1 - \exp^{(-\lambda t)}$, where λ is a constant. This is because the convergence rates of many multi-objective optimizations have been shown to follow an exponential function over computation time [98, 9, 17]. From Equations (5.2) and (5.3), we have S_{lat}:

$$S_{lat} = \frac{2T}{(1 - \exp^{(-\lambda t)}) * (t - T) + 2T} \tag{5.4}$$

where $0 \leq t \leq T$. The goal is to find t_0 to maximize the speedup, S_{lat}. This is equivalent to solving:

$$1 + \exp^{(-\lambda t_0)} * (\lambda(t_0 - T) - 1) = 0, \quad 0 \leq t_0 \leq T \tag{5.5}$$

We have further generalized the previous example to support more than two stages. Figure 5.7 illustrates the relationship between the maximum S_{lat} and λ for different number of stages. It shows that the maximum S_{lat} increases with an increase in the prediction rate. This is not surprising since a higher prediction rate results in fewer re-executions of the stages. The figure also shows that for a given prediction rate (i.e., a fixed value of λ), the maximum speedup increases with more stages.

5.4 Evaluation

In this section, we first use a microbenchmark to evaluate the performance of SpecRPC, and then we examine the performance improvements in Replicated Commit (RC) [77] when using SpecRPC. For comparison we use Google's open-source RPC framework, gRPC [42],

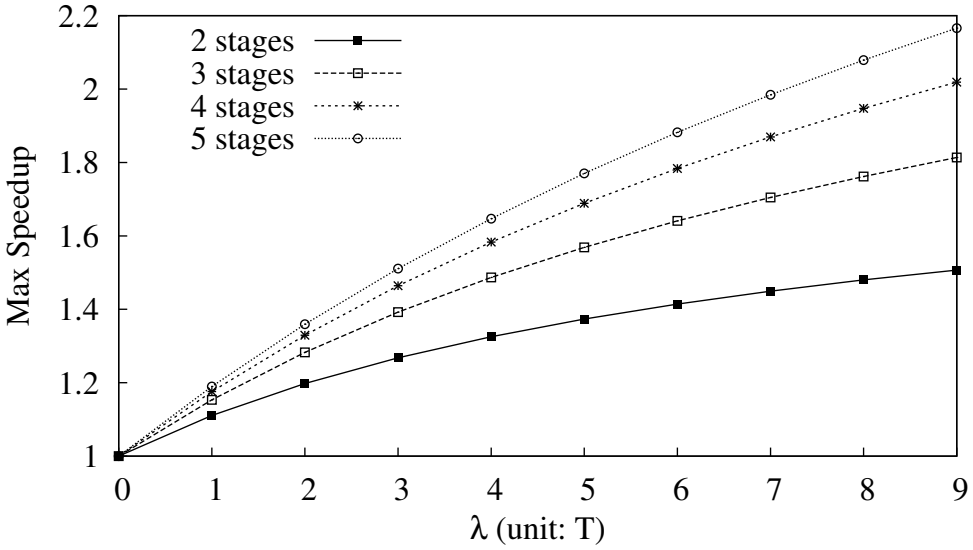


Figure 5.7: Maximum speedup versus λ

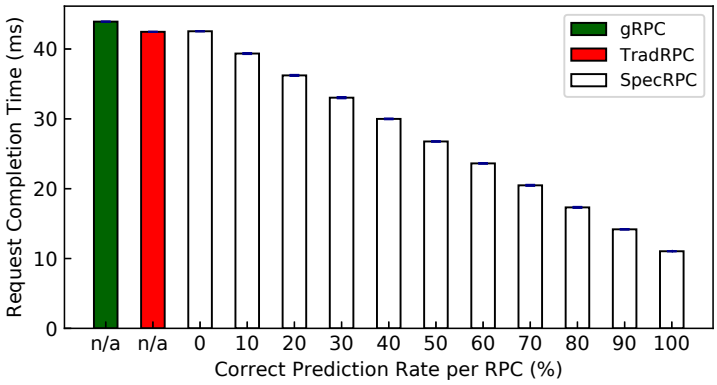
as a baseline. Since gRPC has more features than SpecRPC, which may increase its latency, we also compare our system with *TradRPC*, an RPC framework sharing much of SpecRPC's code base without speculation.

Our experimental testbed uses standard server-class machines, each of which has two 6-core 2.10 GHz Intel Xeon E5-2620 v2 CPUs and 64 GB RAM. The machines are connected to a 1 Gbps Ethernet network. Each experiment consists of 5 runs, and each run lasts 60 seconds during which we measure the performance of the system throughout the middle 30 seconds.

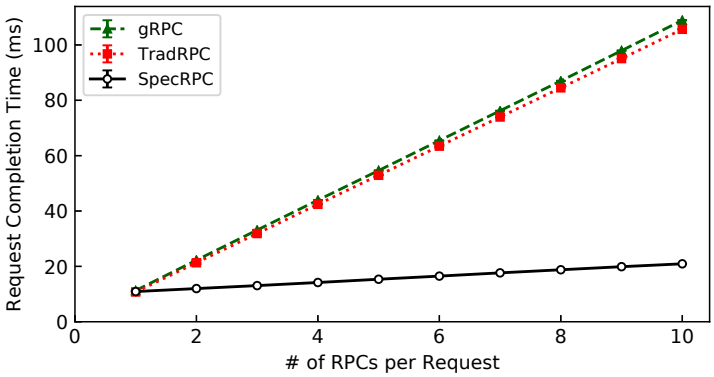
5.4.1 Microbenchmark

Our microbenchmark consists of 16 clients with each performing a sequence of dependent RPCs, defined as a *request*, to multiple servers. Each RPC sends and receives 64 bytes of data. Unless specified, a request consists of 4 RPCs, each of which requires 10 ms to complete. The client issues one request at a time, and each client issues 10 requests per second. This system load allows us to examine the performance of SpecRPC when sufficient system resources are available.

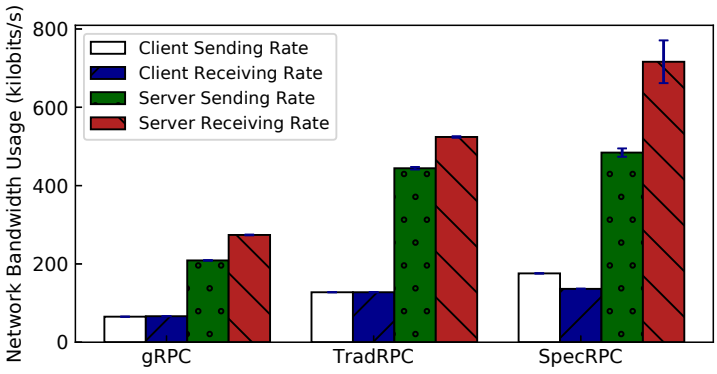
While executing a request by using SpecRPC, the client makes a prediction for each RPC result. We define the probability of the prediction being correct as the *correct pre-*



(a) Latency versus correct prediction rate



(b) Latency versus # of RPCs per request



(c) Network bandwidth usage

Figure 5.8: Microbenchmark results

diction rate per RPC. Figure 5.8 (a) shows the mean completion time of requests under different correct prediction rates per RPC. Compared with the sequential execution of RPCs via using gRPC and TradRPC, SpecRPC achieves up to 75% reduction in request completion time. When prediction is always incorrect, SpecRPC introduces approximately 0.1 ms of overhead compared with TradRPC, which is negligible in a request that requires more than 40 ms to complete. Our experiments show that gRPC has slightly higher overhead than both TradRPC and SpecRPC. This may be because gRPC provides additional features that are not supported by TradRPC and SpecRPC. The results also show that even with only a 50% correct prediction rate per RPC, SpecRPC still provides about a 40% reduction in request completion time compared to gRPC.

We further examine the performance of SpecRPC by varying the number of dependent RPCs in a request. In the following experiments, the correct prediction rate per RPC is set to be 90%. Figure 5.8 (b) shows that the mean request completion time for SpecRPC increases more slowly than for gRPC and TradRPC. As expected, the request completion times for both gRPC and TradRPC increase linearly with the number of dependent RPCs per request. SpecRPC experiences a small increase in its request completion time with additional dependent RPCs because only incorrect predictions lead to sequential execution of RPCs.

Lastly, we examine the network bandwidth usage of the three different RPC frameworks. Figure 5.8 (c) shows that TradRPC has higher network bandwidth usage than gRPC. This is because gRPC has a more optimized implementation of message serialization than TradRPC. The results also show that SpecRPC has higher network bandwidth usage than TradRPC. This is because SpecRPC must re-execute some of its RPCs due to incorrect predictions.

5.4.2 Replicated Commit

In this section, we examine the performance improvements in Replicated Commit (RC) when using SpecRPC. We implement an RC prototype in an in-memory key-value store, and our implementation asynchronously persists transaction logs to SSDs. We compare three versions of RC, one using gRPC [42] as a baseline, one using SpecRPC to enable speculative execution, and one using TradRPC which shares much of SpecRPC's code base without speculation. Our RC prototype using gRPC consists of approximate 4000 lines of Java code, while SpecRPC introduces an additional 100 lines of changes on the server side and about 300 lines of changes on the client side. These changes are to modify RPC registrations and invocations in order to introduce speculation on read results. Our SpecRPC changes do not modify the commit protocol.

	Ireland	Seoul
Oregon	140	122
Ireland	-	243

Table 5.1: Network roundtrip delays (ms) between datacenters from [86]

Experimental Setup

In our experiments, we use the Linux traffic control utility to set the network latency between machines in order to emulate the geo-distributed environment specified in [86], which consists of three datacenters. The network roundtrip delays between datacenters are shown in Table 5.1.

Our transactional key-value store contains 10 million key-value pairs. The data is sharded into three partitions, with each partition having a replica at every datacenter. One RC server manages one replica. Our evaluation uses close-loop experiments, in which a client sends transactions back-to-back, and there are 16 clients in each datacenter.

Our evaluation uses two workloads, YCSB+T [32] (an extension of the YCSB workload [25] with transactional support) and Retwis [68] (a Twitter-like workload). In our experiments, by default, the data access pattern in both workloads follows a Zipfian distribution with $alpha = 0.75$.

YCSB+T Workload

We first use YCSB+T to repeat the RC experiments in [77]. Without using speculation, read latency dominates the transaction completion time as the number of reads increases. In this experiment, the number of operations (reads and writes) varies from 5 to 50, and the ratio of reads and writes is 1:1 (mirroring the values used in the original experiments in [77]). Figure 5.9 shows that the average transaction completion time of our RC prototype with gRPC and TradRPC increases linearly with the number of reads, which matches the results in [77]. In contrast, the average transaction completion time of the SpecRPC version of RC is nearly independent of the number of reads in a transaction. Going from 5 to 50 operations per transaction, the transaction completion time only increases by 23% for SpecRPC, compared to more than 600% for the non-speculative systems. This low increase in completion time is a result of correct speculation, which allows SpecRPC to parallelize dependent read operations. This experiment also shows that the read result from the first responding replica is a good predictor of the final result of a quorum read. This approach correctly predicted the final quorum read result with more than 95% accuracy.

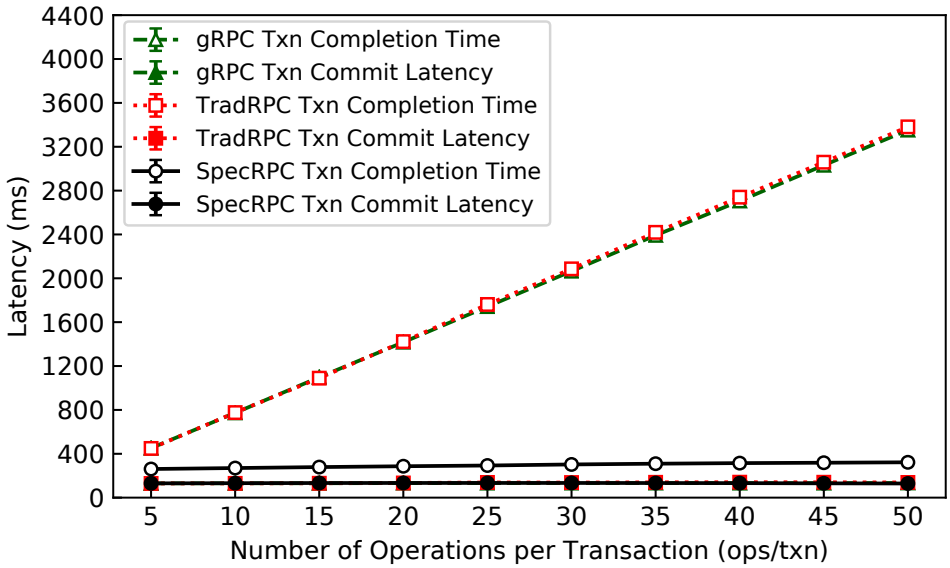
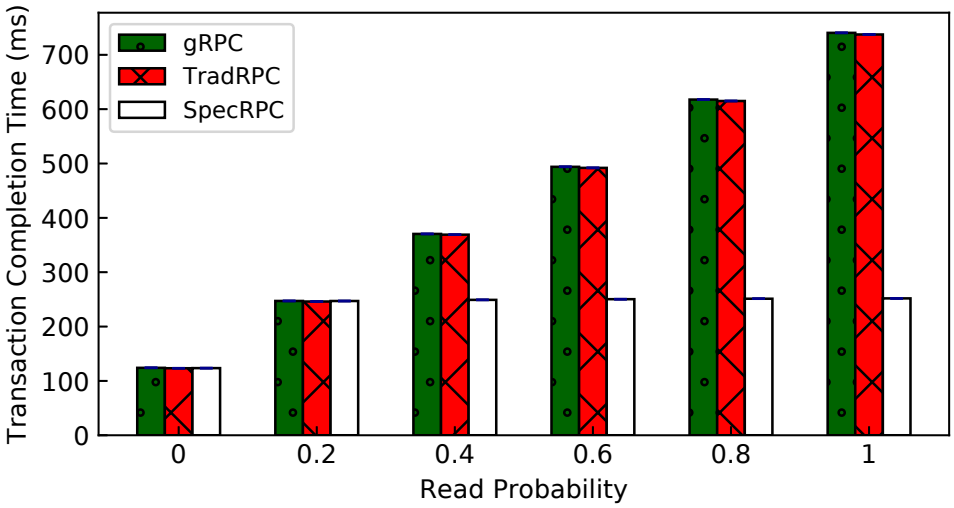


Figure 5.9: Mean latency versus the number of operations per transaction with YCSB+T

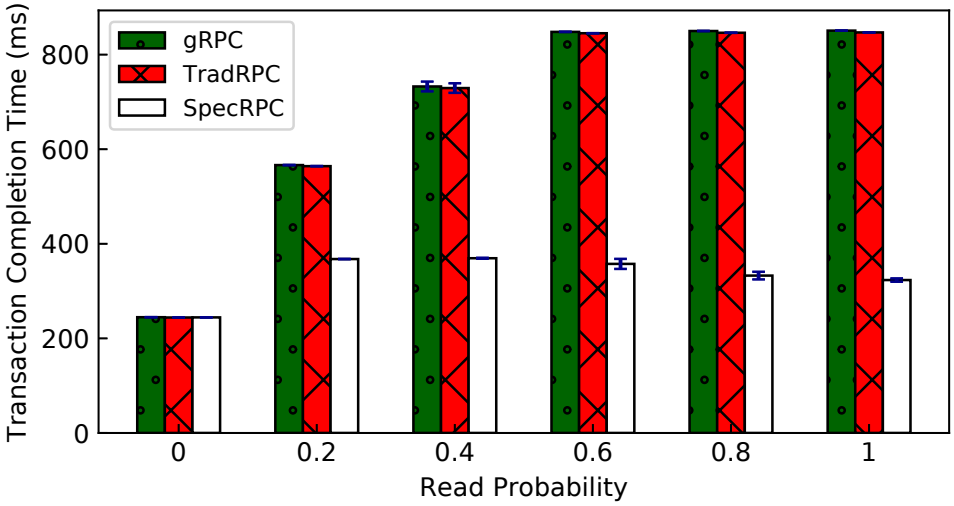
We further examine the impact of the probability that a request in a transaction is a read (instead of a write) on transaction completion time. In this experiment we use 5 operations per transaction. Figure 5.10 shows both the median and 99th percentile of the transaction completion time. As expected, with gRPC and TradRPC, the median transaction completion time grows linearly with the read probability. The tail transaction completion time grows even more quickly, as even with a 0.2 probability, the transactions in the tail consist mostly of read operations. At 0.6 probability and higher, nearly all of the transactions in the tail consist of 5 read operations. With SpecRPC, the median and tail completion times are largely unaffected by read probability. This is because the correct prediction rate for this workload is above 99%, with the rate growing with increasing read probability.

Retwis Workload

In this section, we use a Twitter-like workload, Retwis, to evaluate the performance of RC using SpecRPC. We use the same transaction profile as the Retwis workload in [118], which is shown in Table 5.2. Figure 5.11 shows the CDF of RC's transaction completion time when using gRPC, TradRPC, and SpecRPC. Compared with gRPC and TradRPC, SpecRPC reduces the average transaction completion time by 58%.



(a) Median



(b) 99th percentile

Figure 5.10: Latency versus read probability with YCSB+T

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 5.2: Retwis transaction profile from [118]

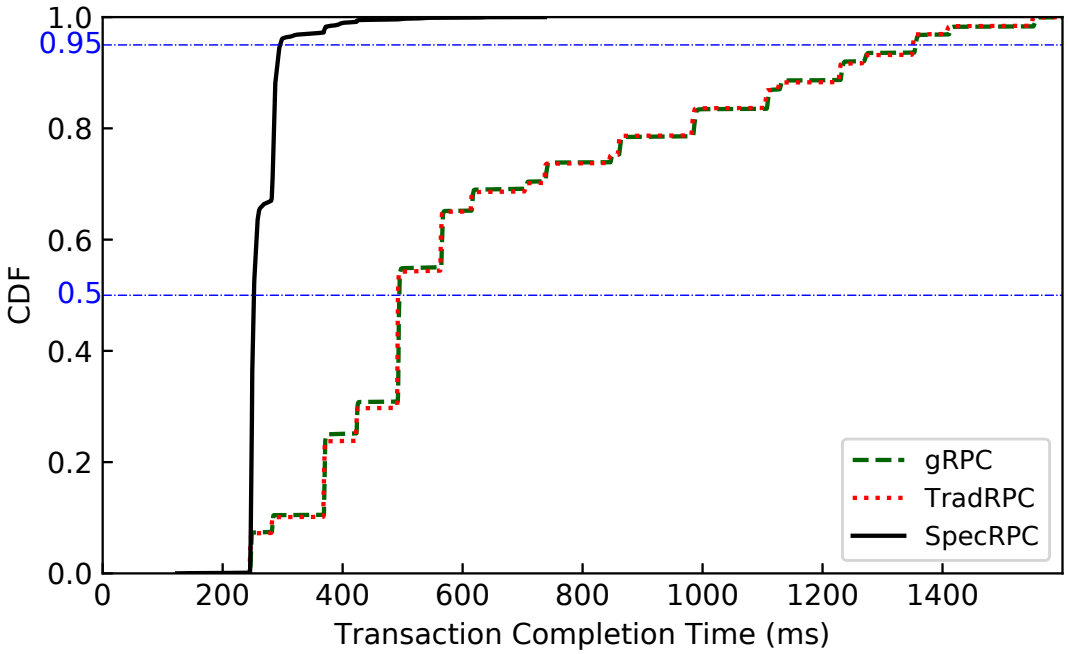
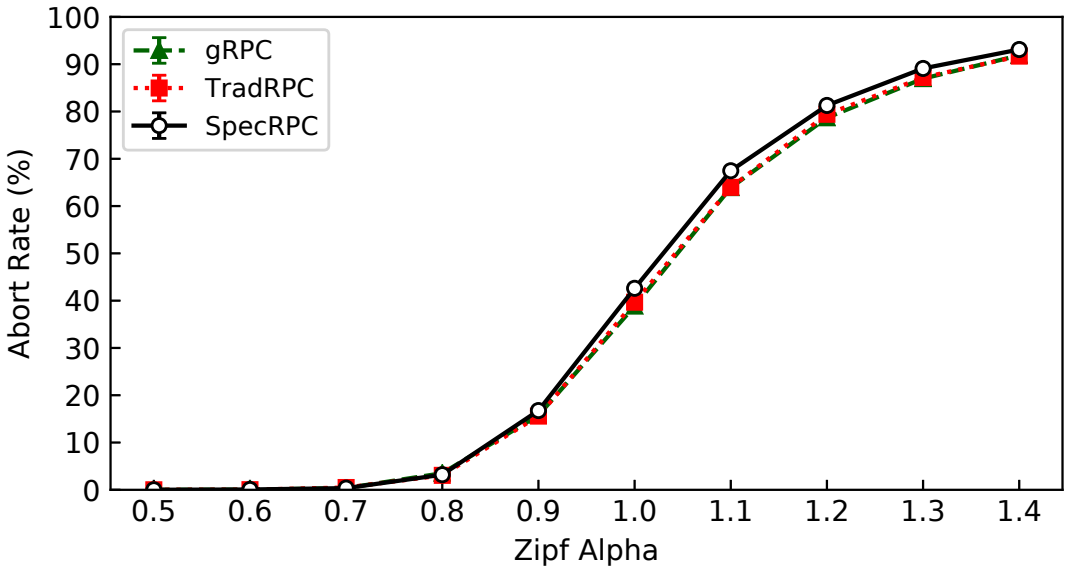
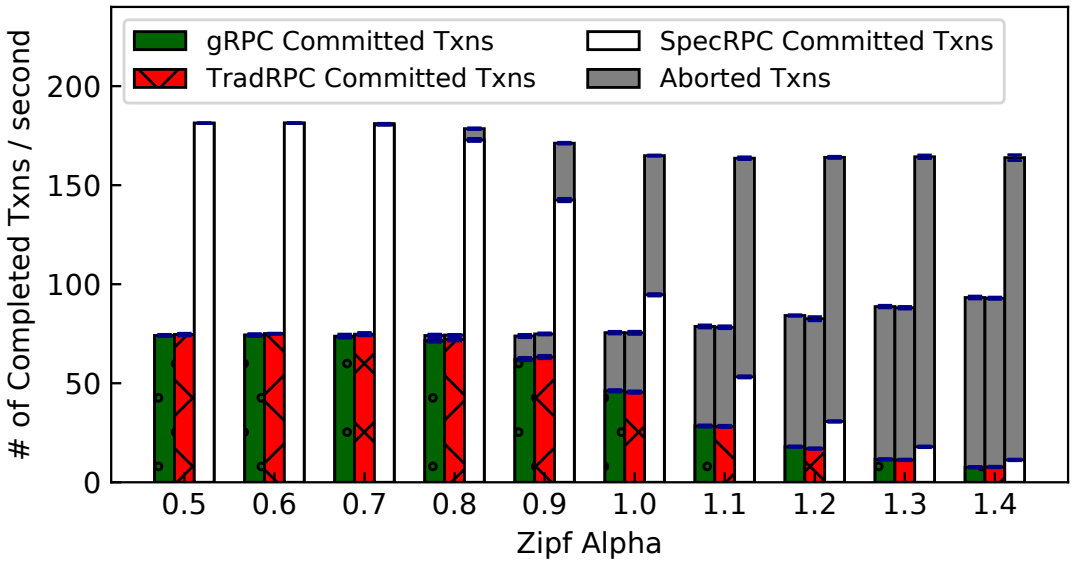


Figure 5.11: Transaction completion time with Retwis

We then adjust the Zipfian alpha value to examine the impact of transaction contention on the performance of these systems. Figure 5.12 (a) and (b) show that SpecRPC's abort rate is only marginally higher than gRPC and TradRPC despite processing twice the number of transactions per second in this closed loop experiment with a fixed number of clients. For example, when the Zipfian alpha value is 0.9, using SpecRPC introduces about a 1% higher abort rate than using gRPC and TradRPC. However, SpecRPC is able to commit 142 transactions per second, while gRPC and TradRPC can only commit 62 and 63 transactions per second, respectively. The higher transaction processing rate of SpecRPC is due to its transaction completion time being half of that of the other systems.



(a) Abort rate



(b) Number of completed transactions

Figure 5.12: Retwis workload with varying alpha values

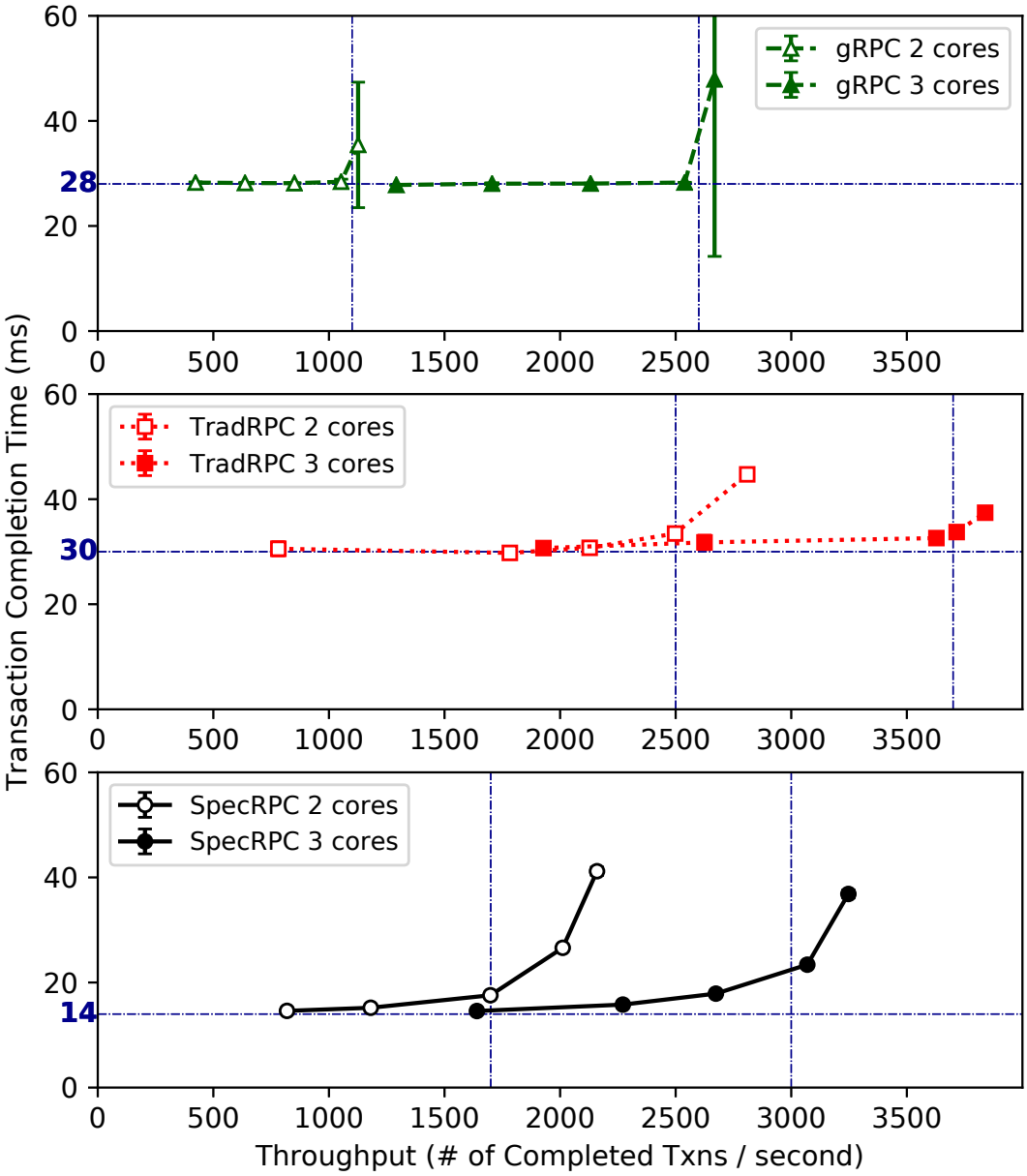


Figure 5.13: Average transaction completion time versus throughput with Retwis

In order to measure the maximum throughput of these systems, we must saturate them with client requests. To accomplish this in our cluster, we have to reduce the computing resources of the RC servers. For these experiments, we set the roundtrip network latency between datacenters to be 5 ms and limit the number of CPU cores per RC server. Since the number of CPU cores is artificially limited, these experimental results do not represent the maximum throughput of the systems in practice. Instead, these experiments aim to compare the throughput of the three systems under the same resource limit and to examine the impact of increasing computing resources on the performance of the three different systems.

As shown in Figure 5.13, all three systems have near perfect speedup in throughput when increasing the number of cores from 2 to 3, where the throughput is indicated by the vertical lines in the graphs. As expected, SpecRPC's throughput is lower than TradRPC's due to speculation overhead. Surprisingly, gRPC has a lower throughput than both other systems, which may be due to additional features that it provides which are not supported by SpecRPC and TradRPC. Although SpecRPC introduces processing overhead, we believe that this is a reasonable tradeoff since throughput can be increased by increasing the number of cores or data shards, whereas transaction completion cannot be reduced with more resources without speculation. In this experiment, it is not possible for gRPC or TradRPC to achieve SpecRPC's 14 ms transaction completion time.

5.5 Discussion

A common pitfall to SE is that stateful execution is not correctly undone after an incorrect speculation. We address this pitfall by recommending or requiring SpecRPC users to follow design patterns that avoid this problem. For example, instead of passing a speculative object directly to a speculative callback, SpecRPC requires users to follow a factory design pattern in which a new speculative object is created for each callback that encapsulates all intermediate results. This allows programmers to not worry about cross-contamination between results from different speculative and non-speculative executions.

Another potential issue with speculation is in supporting operations with side-effects that is outside of the control of the speculation framework. We provide a *specBlock* function that prevents the computation from proceeding until it is in a non-speculative state. SpecRPC also allows an application to register a rollback function for speculative executions so that incorrect speculations can be rolled back. However, because SpecRPC is a user library, a decision that we made to simplify and promote adoption, it cannot enforce correctness in much the same way that a threading library cannot enforce thread safety.

When application developers follow SpecRPC's recommended design patterns, SpecRPC can guarantee that return results are equivalent to the sequential execution of dependent operations (without speculation). SpecRPC also allows developers to log the speculative states of RPCs and callbacks. The developers can use this information to debug their applications, such as tracing side-effects from incorrect speculations. It is possible to implement language primitives to make SpecRPC provide additional guarantees on preventing side-effects, but this requires language-level support.

To provide an abstraction that saves developers' efforts on implementing speculative execution in a distributed environment, the current design of SpecRPC uses an asynchronous RPC interface with callbacks and factory patterns. This should be familiar to many application developers who have experience in asynchronous RPCs and callbacks. Although there may be some learning curve for new programmers to get familiar with SpecRPC's design patterns, the required learning efforts are much less than building their own data structures and state management systems for tracking speculative executions across nodes.

SpecRPC provides the necessary tools to enable a developer to implement a speculative application without manually managing dependencies, discarding incorrect speculations, and hiding speculative state from non-speculative execution, which allows developers to focus on leveraging their domain knowledge to improve prediction rather than spending their time implementing a speculative execution infrastructure. However, the developer must still make reasonable decisions with respect to predictions, and must follow provided guidelines to avoid potential problems.

In addition to the applications described in Section 5.3, many other latency-sensitive applications can benefit from SpecRPC. For example, web applications often execute a chain of services to generate a response for a client request. These applications can use caches to predict service results, enabling services in the chain to execute in parallel. Social network applications can also benefit from speculation as they often perform multiple dependent graph computations. Many of these graph computations, such as triangle counting, are expensive but their results can be estimated quickly using approximation algorithms [46]. These estimates can be used as predictions. A social network application can perform many of its dependent graph computations in parallel if the predictions are correct or within some error bound.

5.6 Chapter Summary

In this chapter, I present SpecRPC, an RPC framework for performing speculative execution. By managing dependencies between callbacks and RPCs, SpecRPC simplifies

the process of using speculation and reduces application latency. I evaluate SpecRPC by implementing a distributed transaction protocol using the framework. The experimental results show that SpecRPC can significantly reduce transaction completion time by using speculation to perform dependent reads in parallel.

We designed SpecRPC with the hope that it would be used by many applications. As a result, some of the core designs of the framework were chosen to simplify adoption for developers and allow deployment in various environments. For example, instead of requiring operating system support for state rollback after an incorrect speculation, we perform state rollback completely within the framework. These design choices necessitate an advisory programming model where correctness relies on applications following our suggested design pattern. Although we believe this is acceptable for most applications, we are exploring other designs, such as introducing language-level changes, that can provide stronger guarantees. We plan to work closely with developers interested in our recently open-sourced implementation, and use their feedback to improve future versions of our framework.

Chapter 6

Future Work and Conclusion

Many distributed systems run a chain of dependent networked services to complete a task. It is straightforward to execute these services in a serial manner. However, this results in high latency, especially when the services require inter-datacenter network I/O operations. This thesis explores approaches to address the latency problem of executing a sequence of dependent network I/O operations in different domains, including geo-distributed database systems and state machine replication. It also studies solutions for a general class of distributed systems that can benefit from using speculation. The result of this work is the design and implementation of three systems, Carousel, Domino, and SpecRPC. Looking back at the three systems, the techniques in one system can be applied to address the limitations in one of the other systems. In the rest of this chapter, I will first discuss future research directions that build on the three systems, and then I will conclude this work.

6.1 Future Research Directions

Network-Aware Transaction Processing

Carousel borrows ideas from Fast Paxos [66] to introduce a fast path to commit transactions. Unlike state machine replication that has replicas agree on a request's position in a log before executing the request, Carousel replica servers independently prepare a transaction and then agree on the prepare result. In Carousel, concurrent transactions that have no conflicts will not compete for a log position and thus will not cause the fast path to fail. However, when two concurrent transactions that have conflicts arrive at replicas in

different orders, the replicas may have different prepare results for the two transactions, and neither of the transactions can commit via the fast path.

To increase the likelihood that Carousel commits a transaction via the fast path when there are conflicting transactions, we can apply a Domino-like ordering approach that leverages network measurement. Although directly using Domino in Carousel can achieve this goal, it would introduce additional delays to commit a transaction. This is because Domino delivers replicated transactions to Carousel by following its log order regardless of whether the transactions have conflicts or not. Instead, we can leverage network measurement to order transactions for Carousel servers to process. A Carousel client can assign a transaction with a timestamp, indicating when the transaction should have arrived at enough participant servers. Once a Carousel server receives a transaction, it will not process the transaction until its clock passes the transaction's timestamp. Unlike Domino's deterministic ordering, this ordering approach is best-effort, and a server does not need to reject a transaction that arrives late. In addition, this ordering approach can be used for other transaction processing protocols, like TAPIR [118] and Janus [86], to increase their fast-path success rate.

In this ordering approach, the latency of committing a transaction is sensitive to the transaction's timestamp because each replica follows wall-clock time to process transactions. An unnecessarily large timestamp could significantly introduce delays to the commit latency. One research direction is to design a feedback control system to estimate the arrival time based on historical data. In addition to recent network measurement data, this control system can also consider the success rate of the fast path and the popularity of the keys in a transaction. This is because non-conflicting transactions can be committed via the fast path even if they arrive at replicas in different orders.

Supporting General Read-Write Transactions in Carousel

Carousel requires pre-defined read and write keys in a transaction. Some applications may not be able to adopt this transaction model, and they have to perform general read-write transactions. To extend Carousel to support general read-write transactions, one research direction is to use speculation to overlap the execution of a transaction's reads, writes, and commit. For example, a client can use a cache to predict its read result. It can use this predicted result to speculatively perform subsequent read/write operations that depend on the read result. The client can also piggyback any speculatively known read and write keys on its read requests to data servers, and the servers can speculatively start 2PC and replication on the partial read and write sets. SpecRPC can be used to extend Carousel to support such speculative execution. Another research problem in this direction is how

to efficiently make correct speculations. One approach is to leverage workload statistics to increase the cache hit rate for reads.

Resource Provision for Speculative Execution

SpecRPC allows an application to make multiple predictions for an operation's result in order to increase the chances of having correct speculations. An application can further predict the results of speculative operations in order to execute a chain of dependent operations in parallel. If there are multiple predicted results for each operation in a chain, the number of (speculatively) executed function instances increases exponentially with the chain length, ending with at most one correct-speculation path. Incorrect speculations will waste a significant amount of computing resources. This may saturate a system when computing resources are limited. Application developers need to make a trade-off between speculations and the resources available for speculative execution.

To address this problem, we can design a speculation manager to control speculative execution based on available computing resources and statistics on correct speculations. For example, the speculation manager can automatically count the rate of correct speculations for each operation, and monitor currently available computing resources in the system. The manager will only speculatively execute an operation if there are sufficient resources or the speculation is likely to be correct. It can also stop performing speculative executions if predictions are incorrect most of the time. Application developers can also define workload-tailored policies to reduce the resource cost for incorrect speculations.

Another research direction is to extend SpecRPC to support serverless computing that can provide computing resources on demand. With elastic resources in serverless computing, application developers only need to set up a budget for using speculation to reduce latency. This extended SpecRPC should minimize the latency for a given budget.

6.2 Concluding Remarks

This thesis presents techniques to reduce latency for executing dependent network I/O operations, especially for those across datacenters. Most of these techniques require additional computing resources to increase parallelism in a system and reduce the number of sequentially executed inter-datacenter operations in a task. I believe that this trade-off is reasonable for latency-critical applications, especially when the prevalence of cloud computing makes computing resources more and more affordable. However, the cost might

be reduced for some applications if most of their network I/O operations can be limited within a datacenter. For example, instead of deploying data across datacenters in a static manner, a geo-distributed database system can leverage workload statistics to move data between datacenters to benefit from data locality.

Limiting a chain of network I/O operations within a datacenter can have lower latency than a single inter-datacenter operation. Although this is a promising approach to design latency-critical systems, practical constraints, such as data privacy or fault-tolerance requirements, may result in unavoidable inter-datacenter network I/O operations. It is critical to limit the number of sequential inter-datacenter network I/O operations in completing a task. I believe that the designs in this thesis will continue to play a part in shaping future geo-distributed systems.

For applications running within a datacenter, there is a trend that many applications shift to split their computation on one server into a chain of tiny networked services to serve a client request, such as in a microservice architecture. Although these network services can benefit from high-speed datacenter networks, the data transfer between these services introduces additional delays compared to running all computation on a single server. The designs of the three systems in this thesis demonstrate different approaches to tackle the sequential execution of dependent network I/O operations. I hope that these techniques will also contribute to the design of future networked systems within a datacenter as reducing latency can improve user experience and increase revenue for many applications.

References

[1] Data Trace for Inter-Region Latency on Azure for the Globe Setting. https://rgw.cs.uwaterloo.ca/BERNARD-domino/trace-azure-globe-6dc-24h-202005170045-202005180045.tar.gz, 2020.

[2] Data Trace for Inter-Region Latency on Azure for the NA Setting. https://rgw.cs.uwaterloo.ca/BERNARD-domino/trace-azure-na-9dc-24h-202005071450-202005081450.tar.gz, 2020.

[3] Domino. https://github.com/xnyan/domino, 2020.

[4] Epaxos. https://github.com/efficient/epaxos, 2020.

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

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

[7] Amazon. Amazon AWS. https://aws.amazon.com/, 2020.

[8] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. Sharding the shards: Managing datastore locality at scale with akkio. In *OSDI*, 2018.

[9] Aristeidis Antonakis, Theoklis Nikolaidis, and Pericles Pilidis. Multi-Objective Climb Path Optimization for Aircraft/Engine Integration Using Particle Swarm Optimization. *Applied Sciences*, 7(5), 2017.

[10] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up Consensus by Chasing Fast Decisions. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN'17, 2017.

[11] SpecRPC Authors. SpecRPC Implementation. https://github.com/xnyan/specrpc, 2018.

[12] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM'07, 2007.

[13] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.*, 8(3), 2014.

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

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

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

[17] Kalyan Shankar Bhattacharjee, Hemant Kumar Singh, and Tapabrata Ray. Multi-Objective Optimization With Multiple Spatially Distributed Surrogates. *ASME Journal of Mechanical Design*, 138(9), 2016.

[18] Anasua Bhowmik and Manoj Franklin. A General Compiler Framework for Speculative Multithreading. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, SPAA'02, 2002.

[19] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov,

Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the USENIX Conference on Annual Technical Conference*, USENIX ATC'13, 2013.

[20] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI'06, 2006.

[21] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI'99, 1999.

[22] Fay Chang and Garth A. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI'99, 1999.

[23] Cockroach Labs. CockroachDB. https://github.com/cockroachdb/cockroach, 2017.

[24] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1(2), 2008.

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

[26] 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. Spanner: Google's Globally-Distributed Database. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

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

[28] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine

Replication. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, NSDI'08, 2008.

[29] Francis Dang, Hao Yu, and Lawrence Rauchwerger. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *Proceedings of the International Parallel and Distributed Processing Symposium*, IPDPS'02, 2002.

[30] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. *ACM SIGCOMM Computer Communication Review*, 46(2), 2016.

[31] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR'15, 2015.

[32] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *Proc. VLDB Endow.*, 4(8), 2011.

[33] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP'07, 2007.

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

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

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

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

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

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

[40] Keir Fraser and Fay Chang. Operating System I/O Speculation: How Two Invocations Are Faster Than One. In *Proceedings of the USENIX Conference on Annual Technical Conference*, USENIX ATC'03, 2003.

[41] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosunblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *NSDI*, 2018.

[42] Google. gRPC-go. https://github.com/grpc/grpc-go, 2017.

[43] Google. Google Cloud Platform. https://cloud.google.com/, 2020.

[44] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental Consistency Guarantees for Replicated Objects. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, 2016.

[45] Gurobi. Gurobi Optimization. http://www.gurobi.com/, 2018.

[46] Mohammad Al Hasan and Vachik S. Dave. Triangle Counting in Large Networks: A Review. *WIREs Data Mining and Knowledge Discovery*, 8(2), 2018.

[47] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the USENIX Conference on Annual Technical Conference*, USENIX ATC'10, 2010.

[48] IBM. CPLEX Optimizer. https://www.ibm.com/analytics/cplex-optimizer, 2018.

[49] David R. Jefferson, Brian Beckman, Frederick P. Wieland, Leo. Blume, and Mike Diloreto. Time Warp Operating System. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP'87, 1987.

[50] Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. Speculative Separation for Privatization and Reductions. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI'12, 2012.

[51] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-Performance Broadcast for Primary-Backup Systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN'11, 2011.

[52] 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. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.*, 1(2), 2008.

[53] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, STOC'97, 1997.

[54] Kirk Kelsey, Tongxin Bai, Chen Ding, and Chengliang Zhang. Fast Track: A Software System for Speculative Program Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO'09, 2009.

[55] Farhan Khan. The Cost of Latency. https://www.digitalrealty.com/blog/the-cost-of-latency/, 2015.

[56] Sangman Kim, Michael Z. Lee, Alan M. Dunn, Owen S. Hofmann, Xuan Wang, Emmett Witchel, and Donald E. Porter. Improving Server Applications with System Transactions. In *Proceedings of the European Conference on Computer Systems*, EuroSys'12, 2012.

[57] Thorsten Koch, Ted Ralphs, and Yuji Shinano. Could We Use a Million Cores to Solve an Integer Program? *Mathematical Methods of Operations Research*, 76(1), 2012.

[58] Ron Kohavi, Alex Deng, Brian Frasca, Toby Walker, Ya Xu, and Nils Pohlmann. Online controlled experiments at large scale. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD'13, 2013.

[59] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP'07, 2007.

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

[61] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2), 2010.

[62] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), 1978.

[63] Leslie Lamport. The Part-time Parliament. *ACM Transactions on Computer Systems*, 16(2), 1998.

[64] Leslie Lamport. Paxos Made Simple. *Technical Report, Microsoft*, 2001.

[65] Leslie Lamport. Generalized Consensus and Paxos. *Technical Report, Microsoft*, 2005.

[66] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2), 2006.

[67] John R. Lange, Peter A. Dinda, and Samuel Rossoff. Experiences with Client-Based Speculative Remote Display. In *Proceedings of the USENIX Conference on Annual Technical Conference*, USENIX ATC'08, 2008.

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

[69] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: Efficient Online Multiprocessor Replayvia Speculation and External Determinism. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'10, 2010.

[70] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, 2016.

[71] Chia-Chi Lin, Virajith Jalaparti, Matthew Caesar, and Jacobus Van Der Merwe. DEFINED: Deterministic Execution for Interactive Control-Plane Debugging. In *Proceedings of the USENIX Conference on Annual Technical Conference*, USENIX ATC'13, 2013.

[72] Greg Linden. Technical Blog. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html, 2006.

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

[74] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. *Technical Report, MIT*, MIT-CSAIL-TR-2012-021, 2012.

[75] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP'11, 2011.

[76] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, 2013.

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

[78] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, 2008.

[79] Marissa Mayer. What Google Knows. The 2006 Web 2.0 Submit http://conferences.oreillynet.com/presentations/web2con06/mayer.ppt, 2006.

[80] James Mickens, Jeremy Elson, Jon Howell, and Jay Lorch. Crom: Faster Web Browsing Using Speculative Execution. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, 2010.

[81] Microsoft. Microsoft Azure. https://azure.microsoft.com, 2020.

[82] Microsoft. Virtual Network Peering in Microsoft Azure. https://docs.microsoft.com/en-us/azure/virtual-network/virtual-network-peering-overview, 2020.

[83] David L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10), 1991.

[84] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP'13, 2013.

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

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

[87] Arun Natarajan, Peng Ning, Yao Liu, Sushil Jajodia, and Steve E. Hutchinson. NSDMiner: Automated Discovery of Network Service Dependencies. In *Proceedings of the International Conference on Computer Communications*, INFOCOM'12, 2012.

[88] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative Execution in a Distributed File System. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP'05, 2005.

[89] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the Sync. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI'06, 2006.

[90] Brian M. Oki and Barbara H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, PODC'88, 1988.

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

[92] Andrew Pavlo. 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, 2017.

[93] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating System Transactions. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP'09, 2009.

[94] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, 2015.

[95] Manohar K. Prabhu and Kunle Olukotun. Exposing Speculative Thread Parallelism in SPEC2000. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP'05, 2005.

[96] Lawrence Rauchwerger and David A. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2), 1999.

[97] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4), 1990.

[98] Bryan Van Scoy, Randy A. Freeman, and Kevin M. Lynch. The Fastest Known Globally Convergent First-Order Method for Minimizing Strongly Convex Functions. *IEEE Control Systems Letters*, 2(1), 2018.

[99] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction Chopping: Algorithms and Performance Studies. *ACM Transactions on Database Systems*, 20(3), 1995.

[100] Yee Jiun Song, Marcos K. Aguilera, Ramakrishna Kotla, and Dahlia Malkhi. RPC Chains: Efficient Client-Server Communication in Geodistributed Systems. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, NSDI'09, 2009.

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

[102] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. AutoBash: Improving Configuration Management with Operating System Causality Analysis. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP'07, 2007.

[103] Martin Susskraut, Thomas Knauth, Stefan Weigert, Ute Schiffel, Martin Meinhold, and Christof Fetzer. Prospect: A Compiler Framework for Speculative Parallelization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO'10, 2010.

[104] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP'95, 1995.

[105] Alexander Thomson and Daniel J. Abadi. The Case for Determinism in Database Systems. *Proc. VLDB Endow.*, 3(1-2), 2010.

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

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

[108] Robbert Van Renesse and Deniz Altinbuken. Paxos Made Moderately Complex. *ACM Computing Surveys*, 47(3), 2015.

[109] Robbert Van Renesse, Nicolas Schiper, and Fred B. Schneider. Vive La Différence: Paxos vs. Viewstamped Replication vs. Zab. *IEEE Transactions on Dependable and Secure Computing*, 12(4), 2015.

[110] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and Surviving Data Races Using Complementary Schedules. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP'11, 2011.

[111] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'11, 2011.

[112] Michael Wei, Matias Bjørling, Philippe Bonnet, and Steven Swanson. I/O Speculation for the Microsecond Era. In *Proceedings of the USENIX Conference on Annual Technical Conference*, USENIX ATC'14, 2014.

[113] Benjamin Wester, Peter M. Chen, and Jason Flinn. Operating System Support for Application-Specific Speculation. In *Proceedings of the European Conference on Computer Systems*, EuroSys'11, 2011.

[114] Benjamin Wester, James Cowling, Edmund B. Nightingale, Peter M. Chen, Jason Flinn, and Barbara Liskov. Tolerating Latency in Replicated State Machines through Client Speculation. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, NSDI'09, 2009.

[115] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems.

In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, NSDI'09, 2009.

[116] Paraskevas Yiapanis, Gavin Brown, and Mikel Luján. Compiler-Driven Software Speculation for Thread-Level Parallelism. *ACM Transactions on Programming Languages and Systems*, 38(2), 2015.

[117] Hao Yu and Lawrence Rauchwerger. Adaptive Reduction Parallelization Techniques. In *Proceedings of the ACM International Conference on Supercomputing 25th Anniversary Volume*, 2014.

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

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