Domino: Using Network Measurements to Reduce State Machine Replication Latency in WANs

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

Linguan Yang
l69yang@uwaterloo.ca
University of Waterloo

Bernard Wong
bernard@uwaterloo.ca
University of Waterloo

ABSTRACT

This paper introduces Domino, a low-latency state machine replication protocol for wide-area networks. 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. Our experiments on Microsoft Azure show that Domino can achieve significantly lower commit latency than other consensus protocols, such as Mencius, Fast Paxos, and EPaxos.

CCS CONCEPTS

• **Computer systems organization → Reliability**.

ACM Reference Format:
Xinan Yan, Linguan Yang, and Bernard Wong. 2020. Domino: Using Network Measurements to Reduce State Machine Replication Latency in WANs. In *Proceedings of CoNEXT '20*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3386367.3431291

1 INTRODUCTION

Many critical infrastructure services use consensus protocols to replicate their state across geo-distributed nodes, allowing them 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 [21] extends Paxos [18, 19] 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 state machine replication 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 [21] (i.e., the slow path) to choose a request for each conflicting position. This will cause high latency, especially in wide-area networks (WANs).

Furthermore, depending on the geographic locations of the clients and replicas in a WAN, a leader-based replication protocol may have lower commit latency than Fast Paxos. This is because, in Fast Paxos, a client has to wait for the responses from at least a supermajority of replicas while a leader-based protocol requires agreement from only a majority of replicas.

In this paper, we introduce Domino, a state machine replication (SMR) protocol that uses network measurements to reduce commit latency. It supports both Fast Paxos-like consensus and 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.

Unlike in other consensus protocols, when clients use the Fast Paxos-like consensus in Domino, ordering is not determined by the request's arrival time at the servers, but instead by a client-provided timestamp that corresponds to a unique, empty and uncommitted log entry. The timestamp indicates when a request should have arrived at a supermajority of the servers. A request that arrives at a server after its timestamp will not be counted, although the protocol may still choose to accept that request if enough other servers received the request before its timestamp. Because clients can select unique timestamps corresponding to unique log entries, Domino can always complete its Fast Paxos-like consensus in a

CoNEXT '20, December 1–4, 2020, Barcelona, Spain
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7948-9/20/12…$15.00
https://doi.org/10.1145/3386367.3431291

[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 [21].

single roundtrip for a request as long as its timestamp has not expired, and in the absence of failure.

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. Although execution latency can be masked through application-level reordering, excessive execution latency should nevertheless be avoided as they can introduce user-perceptible artifacts. To address this, 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 accurate network latency predictions, 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 Azure, 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.

This paper makes three main contributions:

- By leveraging network measurements, Domino introduces a practical way of using Fast Paxos to implement SMR in WANs, where it can commit requests within one network roundtrip in the common case.
- To achieve low commit latency, Domino uses both a Fast Paxos-like protocol and a leader-based protocol. Domino clients can independently choose which to use based on network measurement data.
- Our experiments on Microsoft Azure show that Domino achieves significantly lower commit latency than Mencius and EPaxos.

2 RELATED WORK

Many systems (e.g., [1, 12, 14]) use consensus protocols to replicate their system state. Replicas in these systems are in different geographic locations to ensure the systems remain available even in the event of a region-wide failure. One of the most commonly used consensus protocols is Paxos [18, 19], which requires $2f + 1$ replicas to tolerate up to f simultaneous replica failures. A common way of using Paxos to implement state machine replication is to store received operations in a log, and have each replica execute committed operations in log order. Paxos guarantees that replicas have the same operation committed at each log entry. In the common case, it requires one network roundtrip for a client to send its request to and receive a response from a replica. It requires a second network roundtrip to select an operation for a log entry and a third roundtrip to enforce consensus on the operation.

To eliminate the roundtrip required to select an operation for a log entry, Multi-Paxos [30] and Raft [28] adopt a leader to receive and order clients' operations. The leader will replicate the operations and the ordering to other replicas. After the replication completes, the leader will notify clients that their operations are accepted. Viewstamped Replication (VR) [23, 27] is a replication protocol that can provide similar guarantees to Paxos. VR uses a primary replica to order operations, and it also requires two network roundtrips to commit an operation.

Fast Paxos [21] can in some cases achieve lower end-to-end latency than Multi-Paxos by allowing a client to send an operation to every replica instead of sending it only to the leader. If at least $\lceil \frac{3}{2} f \rceil + 1$ replicas (i.e., *a supermajority*) accept the operation at the same log entry, the client can learn that the operation has been accepted in just one roundtrip, which is the *fast path*. However, if there are concurrent operations, a supermajority may not be formed for any operation at a log entry. In this case, Fast Paxos runs a recovery protocol (i.e., the *slow path*) to choose one operation for the log entry. Falling back to the slow path will introduce additional latency.

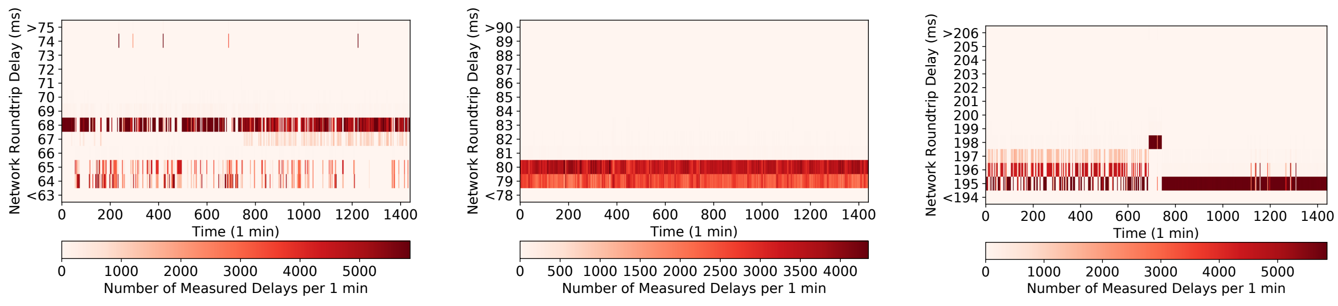
Generalized Paxos [20] can accept concurrent operations that have no conflicts in one consensus instance, but it still has to order conflicting operations. EPaxos [26] allows a client to send an operation to any replica. It can commit the operation in two network roundtrips, but it may require an additional network roundtrip to commit conflicting operations. Furthermore, to achieve its optimal performance, EPaxos requires an application to specify its definition of operation interference, which is not feasible for all applications. By pre-partitioning log entries, Mencius [24] allows each replica to serve as the leader for one partition of log entries. In both Mencius and EPaxos, a client will experience at least two WAN roundtrips to commit an operation if it is not co-located with a replica.

CAESAR [11] is a multi-leader Generalized Consensus protocol for WANs. 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. Gryff [13] introduces consensus-after-register timestamps to order read-modify-write and read/write operations, leveraging the advantages of consensus and shared register protocols to reduce tail latency. However, it still requires 3 network roundtrips to commit read-modify-write operations when there are conflicts. SDPaxos [31] separates ordering from replication. Although replication is entirely decentralized, it uses a centralized ordering server to finalize the order of operations, which may be a performance bottleneck.

NetPaxos [15, 16] 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. SpecPaxos [29] and NOPaxos [22] can achieve low latency by ordering network messages. However, all of these systems require specialized networking hardware and cannot be used in wide-area deployments.

3 INTER-DATACENTER NETWORK DELAYS

Modern cloud providers, such as Microsoft Azure [9], AWS [3], and GCP [8], 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



(a) WA (b) PR (c) NSW

Figure 1: Network roundtrip delays from VA

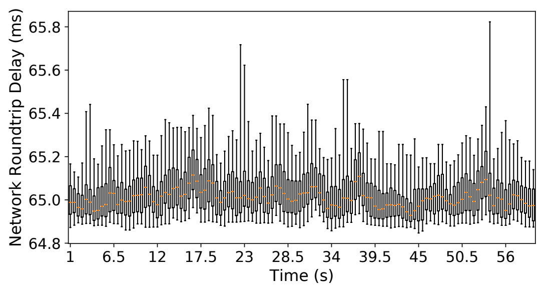


Figure 2: Network roundtrip delays between VA and WA

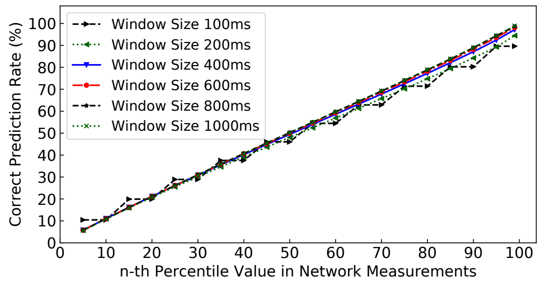


Figure 3: Correct prediction rate

variability of network delays. In this section, we 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 [17] 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 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 [4–6].

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 delay information would be a good source to estimate the current network roundtrip delay. Figure 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 5.4. 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

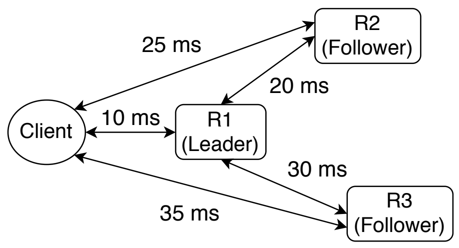


Figure 4: Multi-Paxos (30 ms) versus Fast Paxos (35 ms)

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 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 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 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-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 [10] 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 3, 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 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.

	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 1: Network roundtrip delays (ms)

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

5 DOMINO

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.

5.1 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 [2] 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 [25], can achieve loosely clock synchronization in WANs. A severe clock skew will only affect Domino's performance instead of correctness.

5.2 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 paper, 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 DFP and DM later in Section 5.6.

5.3 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

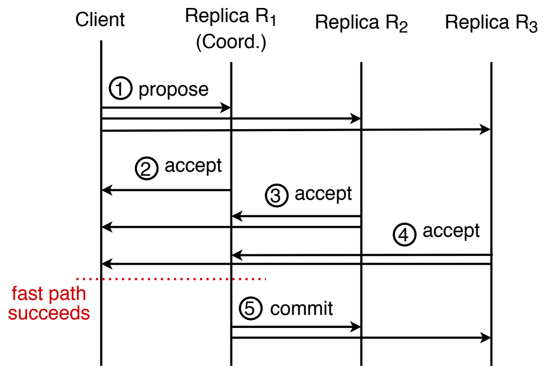


Figure 5: DFP's fast path

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.

5.3.1 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 5.4.

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 [21] to handle request collisions. We will describe the collision handling later in Section 5.3.3.

As shown in Figure 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) to refer to the corresponding numbered points in the message flow illustrated in Figure 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.

5.3.2 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.

5.3.3 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, 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 [21] to commit requests. Figure 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.

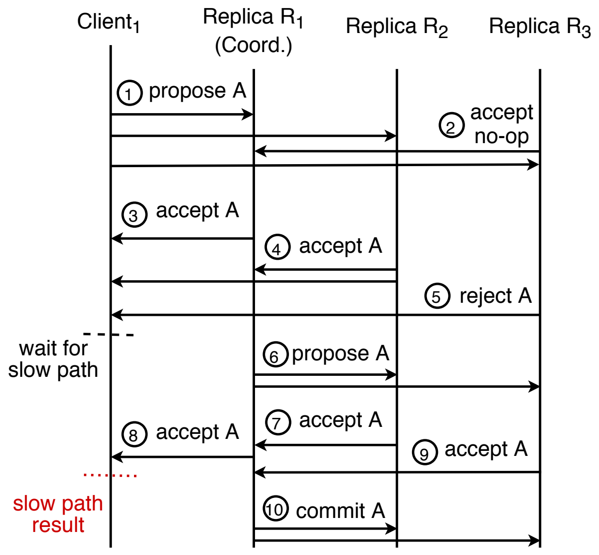


Figure 6: DFP's slow path

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.

5.4 Estimating Request Arrival Time for DFP

In DFP, a client assigns its request with a timestamp indicating when the request should arrive at a supermajority of the replicas. The client will estimate its request's arrival time at each replica, and it will set the request's timestamp to be the the qth smallest arrival time, where q is the supermajority quorum number.

To estimate a request's arrival time at a replica, a 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

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 2: 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 3: 99th percentile misprediction value (ms) by using Domino's OWD measurement technique

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 3, 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 2 and Table 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.

In order to account for the misprediction values shown in Table 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 5.7.

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.

5.5 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 after them. 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 Section 5.6. 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.

5.6 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.

5.7 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.

5.8 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 [21] and Mencius [24], respectively.

When there is a replica failure, by following the failure handling protocol in [24], 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 [21]. 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.

6 IMPLEMENTATION

We have implemented a prototype of Domino in the GO language, which consists of approximate 6 thousand lines of code and is publicly available online [6]. We use gRPC [17] 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

	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: Network roundtrip delays (ms) in NA

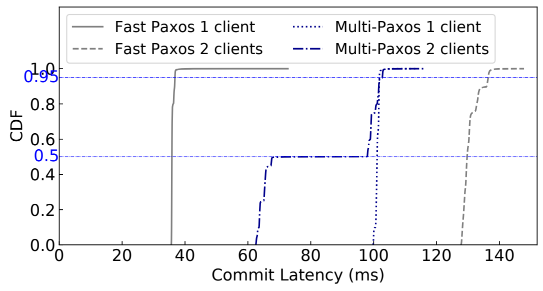


Figure 7: Fast Paxos versus Multi-Paxos

framework of Domino. We will refer this protocol as Fast Paxos in our evaluation.

7 EVALUATION

In our evaluation, we compare Domino, Fast Paxos [21], Mencius [24], EPaxos [26], and Multi-Paxos [30]. We use the open-source implementation of Mencius, EPaxos, and Multi-Paxos in [7, 26]. 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 private computer cluster compare the peak throughput of the different protocols.

7.1 Experimental Settings

Our evaluation mirrors the workload from EPaxos [26], 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 [26]. 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 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.

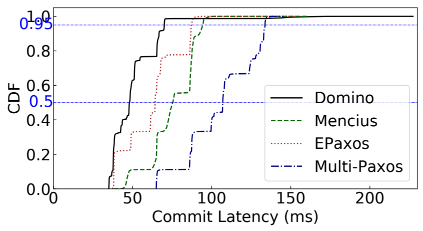
Our evaluation runs every experiment 10 times. Each experiment lasts 90 s, and we use the results in the middle 60 s. We combine the results from the 10 measurements for CDF and box-and-whisker figures. For figures that have error bars, we use the average result from the measurements, and the error bar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

7.2 Experiments on Microsoft Azure

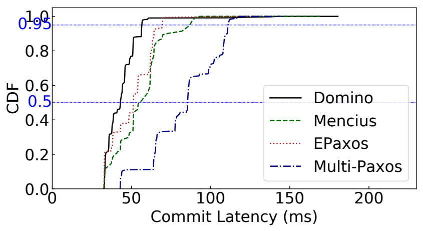
We deploy Domino, Fast Paxos, Mencius, EPaxos, and Multi-Paxos on Microsoft Azure to evaluate their commit latency and execution latency. Our deployment uses the Standard_D4_v3 VM instance that has 4 vCPUs and 16 GB memory, and an instance runs one client or one server (i.e., one replica).

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 and Table 1 (in Section 4) show the average network roundtrip latency between datacenters for NA and Globe, respectively.

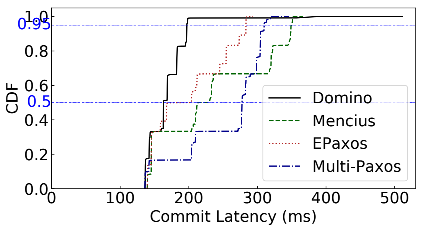
We first show that Fast Paxos [21] could experience high latency when there are only a small number of concurrent clients. Because of the high latency of Fast Paxos, we will focus on comparing Domino with Mencius [24], EPaxos [26], and Multi-Paxos [30] in the rest of our evaluation.



(a) NA with 3 replicas



(b) NA with 5 replicas



(c) Globe with 3 replicas

Figure 8: Commit latency on Azure

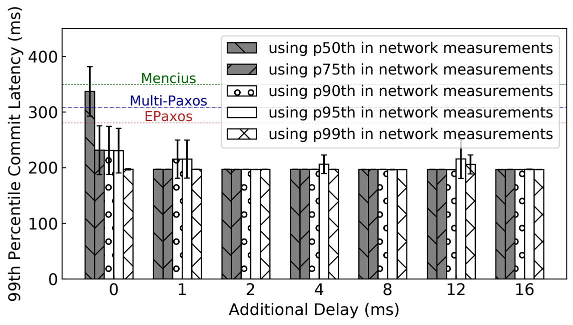


Figure 9: 99th percentile commit latency

7.2.1 *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 7 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 7, 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.

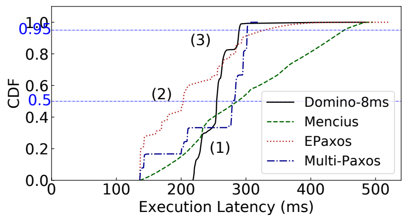
Although Fast Paxos can achieve low latency when there is a single client, our experiments show that Fast Paxos would fall back to its slow path and experience high latency compared to Multi-Paxos even if there are only a small set of concurrent clients in different datacenters. In the rest of our evaluation,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

7.2.2 *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.

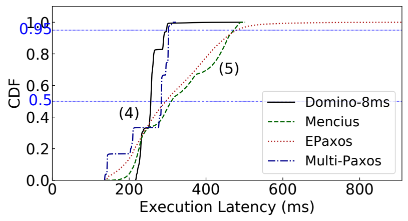
Figure 8 (a) shows the commit latency when there are 3 replicas in WA, VA, and QC, respectively, in which WA hosts the Multi-Paxos leader and the Domino's Fast Paxos coordinator (for the slow path). Domino achieves the lowest commit latency in the median (48 ms) and the 95th percentile (70 ms) compared 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 8 (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 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.



(a) Zipfian alpha = 0.75



(b) Zipfian alpha = 0.95

Figure 10: Execution latency on Azure

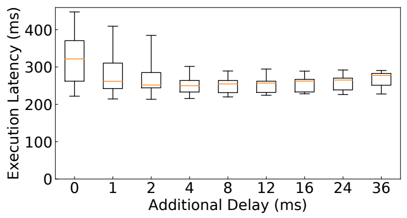


Figure 11: Impact of additional delays (for increasing DFP request timestamps) on execution latency

Figure 8 (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 8 (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 9, 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.

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.

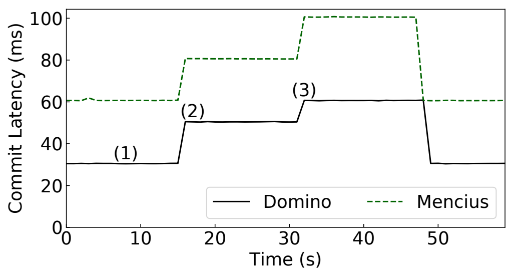
7.2.3 Execution Latency. In this experiment, when a Domino client uses DFP, it increases its request timestamp by 8 ms to reduce execution latency, unless specified otherwise. This is because our analysis in Section 5.4 shows that the 99th percentile misprediction value for request arrival time at replicas is up to 6.24 ms in this setting. Figure 10 (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 10 (b) shows that, at label (4), EPaxos experiences significantly higher execution latency. Request contention has no effect on Domino and Multi-Paxos because they execute 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.

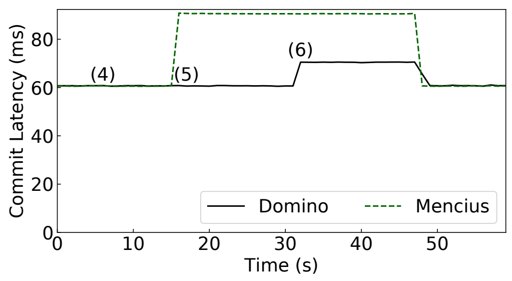
Furthermore, we evaluate the impact of introducing additional delays (for increasing DFP request timestamps) on Domino's execution latency, as shown in Figure 11. In the figure, the middle line in a box is the median execution latency, and the whiskers show the 5th and 95th percentile execution latency, 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.

7.3 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 private computer cluster, 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.



(a) Between a client and replica



(b) Between replicas

Figure 12: Change of network delays

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. 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 12 (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.

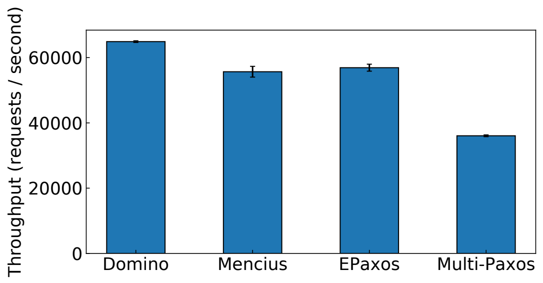


Figure 13: Peak throughput with 3 replicas

7.4 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 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 13 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.

8 CONCLUSION

In this paper, we have presented Domino, a low-latency state machine replication protocol in WANs. Domino uses network measurements to make its Fast Paxos-like consensus protocol commit client requests in one network roundtrip in the common case. It also runs a leader-based consensus protocol in parallel in the same deployment, and allows a client to use network measurement data to decide which consensus protocol to use in order to achieve low commit latency. Our experiments on Microsoft Azure show that Domino can achieve lower commit latency than other protocols, such as Mencius and EPaxos.

ACKNOWLEDGMENTS

We would like to thank our shepherd and the anonymous reviewers for their valuable feedback. We would also like to thank Srinivasan Keshav and Kenneth Salem for their comments on Domino's protocol design. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Canada Foundation for Innovation, Ontario Research Fund, and a grant from the Waterloo-Huawei Joint Innovation Lab.

REFERENCES

[1] 2019. CockroachDB. https://github.com/cockroachdb/cockroach.

[2] 2019. Virtual Network Peering in Microsoft Azure. https://docs.microsoft.com/en-us/azure/virtual-network/virtual-network-peering-overview.

[3] 2020. Amazon AWS. https://aws.amazon.com/.

[4] 2020. 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.

[5] 2020. 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.

[6] 2020. Domino. https://github.com/xnyan/domino.

[7] 2020. EPaxos. https://github.com/efficient/epaxos.

[8] 2020. Google Cloud Platform. https://cloud.google.com/.

[9] 2020. Microsoft Azure. https://azure.microsoft.com.

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

[11] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. 2017. Speeding up Consensus by Chasing Fast Decisions. In *DSN*.

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

[13] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. 2020. Gryff: Unifying Consensus and Shared Registers. In *NSDI*.

[14] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-distributed Database. In *OSDI*.

[15] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. 2016. Paxos Made Switch-y. *SIGCOMM Comput. Commun. Rev.* 46, 2 (2016).

[16] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: Consensus at Network Speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*.

[17] Google. 2019. gRPC-go. https://github.com/grpc/grpc-go.

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

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

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

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

[22] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *OSDI*.

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

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

[25] David L. Mills. 1991. Internet Time Synchronization: the Network Time Protocol. *IEEE Transactions on Communications* 39, 10 (1991).

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

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

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

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

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

[31] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. 2018. SDPaxos: Building Efficient Semi-Decentralized Geo-replicated State Machines. In *SoCC*.