

# Nezha: Deployable and High-Performance Consensus Using Synchronized Clocks

[Technical Report]

Jinkun Geng\*, Anirudh Sivaraman<sup>+</sup>, Balaji Prabhakar\*, Mendel Rosenblum\*

\*Stanford University, <sup>+</sup>New York University

## ABSTRACT

Consensus protocols are widely used to build fault-tolerant applications. At one end of the spectrum of such protocols, Multi-Paxos and Raft have seen widespread adoption, but sacrifice significant latency and throughput. At the other end, protocols like Speculative Paxos and NOPaxos provide high-performance, but require specialized hardware and in-network functionality. This makes it hard to deploy them in environments such as the public cloud, where cloud tenants cannot access the physical network. Our work aims to bridge this gap between deployability and performance.

We present Nezha, a high-performance and deployable consensus protocol that exploits accurate software clock synchronization. Nezha does not require special hardware or physical network access, making it easily deployable in virtualized environments. Instead, it uses a new primitive called *deadline-ordered multicast* (DOM) which orders client-to-replica multicast requests by deadlines specified in synchronized wall-clock time. We compare Nezha to 6 baselines in the public cloud: Multi-Paxos, FastPaxos, NOPaxos and Raft, as well as two recent protocols with clock synchronization, namely, Domino and TOQ-based EPaxos. Evaluations show that Nezha outperforms the baselines by a median of 7.1 $\times$  (range: 1.9–20.9 $\times$ ) in throughput, and by a median of 2.3 $\times$  (range: 1.3–6.5 $\times$ ) in latency. We also use Nezha to replicate two applications (Redis and a prototype financial exchange) and show that Nezha can provide fault tolerance with only a modest performance degradation: compared with the unreplicated system, Nezha sacrifices 5.9% throughput for Redis; it saturates the processing capacity of CloudEx and prolongs the order processing latency by 4.7%. Nezha is open-sourced at <https://gitlab.com/steamgjk/nezhav2>.

## 1 INTRODUCTION

Consensus protocols are a critical component of many distributed systems. They provide fault tolerance through state machine replication for a wide range of systems. These include coordination systems (e.g., Chubby [14] and ZooKeeper [9]) and storage systems (e.g., etcd [26], CockroachDB [83], TiDB [36], MongoDB [96] and Delos [11]).

Multi-Paxos [49] and Raft [74] are the most widely deployed consensus protocols. However, they only provide modest performance [41, 53, 65, 78, 90]. This is in part because they assume nothing more than best-effort service from the network they run on. Many recent proposals have improved the performance of Multi-Paxos/Raft by leveraging in-network functionality. Such proposals make use of network-level multicast [53, 62, 78], control over routing using SDN [20, 53, 78], in-network prioritization [53, 78], switch-based programmability [20, 41, 43, 53], and RDMA [2, 40,

81, 85]. However, in many environments, a system designer does not have access to such in-network functionality. An important example is the environment faced by tenants of the public cloud, where tenants have no access to the physical network.

Meanwhile, the cloud hosts a number of applications that need both high performance (i.e., low latency and high throughput) and fault tolerance. We provide both current and futuristic examples. First, modern databases (e.g., Cosmos DB, TiKV and CockroachDB) would like to provide high throughput and strong consistency (linearizability) over all their data. Yet, they often need to split their data into multiple instances because a single instance’s throughput is limited by the consensus protocol [21, 66, 77]. Second, microsecond-scale applications are pushing the limits of computing [2, 39, 42, 51, 54]. Such applications often have stateful components that must be made fault-tolerant (e.g., the matching engine within a fair-access cloud stock exchange [29], details in §8). To effectively support such applications on the public cloud, we need high-performance consensus protocols.

Driven by these current and future scenarios, we present Nezha, a high-performance consensus protocol which can be widely deployed in environments such as the public cloud by leveraging accurate software clock synchronization [28] in lieu of in-network functionality. Our use of clock synchronization is motivated by the increasing presence of “synchronized time as a service” in public clouds [8, 37], private clouds [3], and colocation providers [22, 25].

The idea of using clock synchronization for consensus is not original [83, 84, 86, 94]. However, designing a practical and correct protocol is non-trivial. We explain the comparison between Nezha and the prior clock-based consensus protocols in §7.3 and §9. In general, Nezha follows Liskov’s design principle: “depend on (clock) synchronization for performance but not for correctness” [55]. Specifically, Nezha incorporates an optimized fast path and a fallback slow path. In the fast path, client requests arrive in the same order at all replicas, allowing replicas to respond to clients without coordination. To trigger the fast path frequently, we introduce a new primitive, *deadline-ordered multicast* (DOM). DOM orders client-to-replica multicast requests by their deadlines represented in synchronized wall-clock time. Replicas respond to clients without coordination if requests are processed in order of their deadlines. When the request cannot be committed in the fast path, Nezha only needs one extra message delay to resolve the inconsistency between followers and the leader and commit it in the slow path. We summarize our contributions below.

(1) A new primitive called DOM. DOM is built on synchronized clocks, and is easily deployable. DOM releases the requests to replicas in the same order, and further triggers the requests to be committed in the fast path.

(2) A new consensus protocol called Nezha. Nezha’s fast path relies on DOM to commit requests in 2 message delays (i.e., 1 round trip time, or 1 RTT) between replicas and proxies/clients. Meanwhile, Nezha’s slow path integrates a *proactive index synchronization* strategy to rapidly resolve leader-follower inconsistency with one additional message delay.

(3) A stateless proxy design to mitigate the burden for the leader replica and/or clients. Because of their stateless nature, the proxies are inherently scalable and their failure does not affect the correctness of Nezha. Further, the proxies also make Nezha a direct drop-in replacement for Multi-Paxos/Raft from a client’s perspective because only the proxies and replicas need to be synchronized to a common clock; the client need not be synchronized to either the proxies or the replicas.

(4) Two replicated applications (the Redis key-value store [79] and the CloudEx financial exchange [29]) that employ Nezha. We find that Nezha provides better application performance than Multi-Paxos, Fast Paxos, and NOPaxos.

We compare Nezha to six baselines in public cloud: Multi-Paxos, FastPaxos, NOPaxos, Raft, Domino and TOQ-based EPaxos under closed-loop and open-loop workloads. In a closed-loop workload, commonly used in the literature [53, 65, 71, 78], a client only sends a new request after receiving the reply for the previous one. In open-loop workloads, recently suggested as a more realistic benchmark [84], clients submit new requests according to a Poisson process, without waiting for replies for previous ones. We find:

(1) In closed-loop tests, Nezha outperforms the baselines Multi-Paxos, Fast Paxos, NOPaxos and Raft by 1.9–20.9× in throughput. Without proxies, Nezha outperforms them by 1.5–6.1× in latency at close to their respective saturation throughputs. With proxies, Nezha still outperforms the baselines by 1.3–4.0× in latency.

(2) In open-loop tests, Nezha outperforms the baselines by 2.5–9.0× in throughput. Without proxies, Nezha outperforms them by 1.3–6.5× in latency at close to their respective saturation throughputs. With proxies, Nezha only sacrifices 33% of latency compared with NOPaxos.

## 2 BACKGROUND

### 2.1 Clock Synchronization

In a distributed system, each node (server or VM) may report a different time due to clock frequency variations. Clock synchronization aims to bring clocks close to each other, by periodically correcting each node’s current clock offset and/or frequency. Given two nodes  $i$  and  $j$ , their clock times are denoted as  $c_i(t)$  and  $c_j(t)$  at a certain real time  $t$ . We consider their clocks synchronized if

$$|c_i(t) - c_j(t)| \leq \epsilon$$

where  $\epsilon$  is the clock error bound, indicating how closely the clocks are synchronized at  $t$ .

Guaranteeing an  $\epsilon$  is difficult because clock synchronization may fail occasionally, and the error bound can grow arbitrarily in such cases. Therefore, Nezha does not depend on clock synchronization for correctness. However, well synchronized clocks can improve the performance of Nezha by increasing the fraction of requests that can be committed in the fast path of Nezha using the DOM primitive. While Nezha is compatible with any clock

synchronization algorithm, in our implementation, we chose to build Nezha on Huygens [28], because Huygens is a high-accuracy software-based system that can be easily deployed in the public cloud.

### 2.2 Consensus Protocols

In this section, we overview consensus protocols most closely related to Nezha, namely, Multi-Paxos [49]/Raft [74], Fast Paxos [48], Speculative Paxos [78], and NOPaxos [53]. Table 1 summarizes basic properties of them and Nezha. §9 provides a more detailed comparison to related work.

**Deployable but low-performance.** Multi-Paxos/Raft and Fast Paxos are generally deployable, but have lower performance. It takes 4 message delays (i.e., 2 RTTs) for Multi-Paxos/Raft to commit one request and the leader can become a throughput bottleneck. Fast Paxos can save 1 message delay if the request is committed in the fast path. However, when there is no in-network functionality that increases the frequency of the fast path (e.g., the MOM primitive [78] or the OUM primitive [53]), Fast Paxos performs much worse (§7.2) because most requests can only be committed in the slow path, but its slow path (2.5 RTTs) is even slower than Multi-Paxos/Raft and causes heavier bottleneck for its leader.

**High-performance but needs in-network functionality.** Speculative Paxos [78] and NOPaxos [53] can achieve high performance. However, both require considerable in-network functionality. Speculative Paxos requires that most requests arrive at replicas in the same order to commit them in 1 RTT. Speculative Paxos achieves this by (1) controlling routing to ensure the same path length for client-to-replica requests and (2) using in-network priorities to ensure that these requests encounter low queues. When reordering occurs, the request can only be committed via the slow path (3 RTTs); the slow path also requires application-specific rollback. Speculative Paxos is very sensitive to packet reordering. Its throughput drops by 10× with a 1% reordering rate [78][Figure 9]; such rates can easily occur in public cloud, where routing is out of control and packets can go different paths. We include a micro benchmark in §E, which shows the reordering rate can usually be more than 20%. NOPaxos requires a programmable switch as the sequencer to achieve its optimal latency (1 RTT). When such a switch is unavailable, NOPaxos uses a server as a software sequencer, which adds 1 additional message delay to its fast path. Besides, as we show (§7.2), NOPaxos also loses throughput when using a software sequencer in public cloud. In particular, it is not resistant to bursts in our open-loop test, which further increase packet reordering/drop and trigger its slow path, causing distinct degradation (109K reqs/sec to 80K reqs/sec).

## 3 NEZHA OVERVIEW

**Fast path/slow path division.** In order to commit requests more efficiently, a common design principle in several consensus protocols [47, 48, 53, 71, 78] is to involve two different paths to commit requests: a fast and a slow path. While the fast path commits requests more quickly, it may not always succeed and needs to fall back to the slow path when it fails. Nezha uses the same principle: the fast path opportunistically commits requests that arrive in the same order at all replicas, and the slow path is a remedy.

**Table 1: Basic Properties of Typical Consensus Protocols; F:fast path; S:slow path.**

Protocols	Message Delays	Load on Leader <sup>1</sup>	Quorum Size	Quorum Check	Inconsistency Penalty	Deployment Requirement
Multi-Paxos/Raft	4	$2(2f + 1)$	$f + 1$	Leader	Low	No special requirement
Fast Paxos	F:3 S:5	$2f + 2$	F: $f + \lceil f/2 \rceil + 1$ S: $f + 1$	F/S: Leader	Medium	No special requirement
Speculative Paxos	F:2 S:6	2	F: $f + \lceil f/2 \rceil + 1$ S: $f + 1$	F: Client S: Leader	High (rollback)	Priority scheduling, SDN control, etc
NOPaxos	F: 2 or 3 <sup>2</sup> S: 4 or 5	2	F: $f + 1$ S: $f + 1$	F: Client S: Leader	Medium	Priority scheduling, programmable switch <sup>3</sup> , SDN control, etc
Nezha (No proxy)	F: 2 S: 3	$2 + 2f/m$ <sup>4</sup>	F: $f + \lceil f/2 \rceil + 1$ S: $f + 1$	F/S: Client	Low	Clock synchronization
(With proxy)	F: 4 S: 5			F/S: Proxy		

<sup>1</sup> Load on Leader indicates how many messages the leader processes per client request.

<sup>2</sup> When NOPaxos uses a switch-based sequencer, the message flow client→sequencer→replica incurs one message delay because the switch is on path; hence, the overall latency is 2 message delays. With a software sequencer, both client→sequencer and sequencer→replica incur one message delay; hence, the overall latency is 3 message delays. The same holds for the slow path.

<sup>3</sup> Programmable switch serves as the hardware sequencer, and it is unnecessary when NOPaxos uses software sequencer. However, software sequencers can reduce throughput

<sup>4</sup> 2 of the messages are the request and the reply. The other  $2f$  messages are *sync* messages with much smaller size, and can be batched, so the load is amortized by a factor of  $m$  (details in §5.4).

**Primitive/protocol decoupling.** The underlying assumption behind the fast path/slow path division is that the fast path is the common case. To increase the fraction of client requests that take the fast path, Speculative Paxos [78] and NOPaxos [53] decouple the responsibility of consensus between 2 components: a *primitive* and a *replication protocol*. The primitive cannot guarantee consensus, but increases the likelihood that requests are received in the same order across all replicas. This allows requests to take the fast path frequently. The replication protocol is responsible for consensus and achieves consensus more quickly on the fast path. Inspired by this primitive-protocol decoupling, we build a primitive called Deadline-Ordered Multicast (DOM), for Nezha to achieve high-performance consensus. DOM uses clock synchronization in lieu of in-network functionality to order client-to-replica requests in the common case.

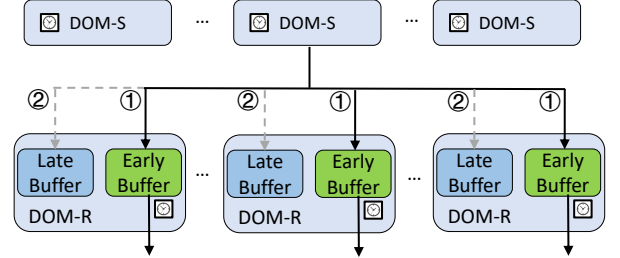
**Stateless proxies.** Nezha is still a leader-based consensus protocol, but Nezha delegates proxies to undertake the two most heavy tasks: request multicast and quorum check. Therefore, Nezha’s leader is much more lightweight than Multi-Paxos/Raft. Nezha’s proxy design is different from the offloading strategy adopted by some existing works (e.g., Speculative Paxos, NOPaxos, CURP [76]): they let clients undertake these tasks, which is intrusive to the client-side logic and constrains the performance of individual clients (§7.6). Besides, unlike the proxies employed by the other works (e.g., PigPaxos [17] and Compartmentalized Paxos [65]), Nezha’s proxies are stateless, so they enjoy much better scalability and their failure causes no harm to Nezha’s correctness, thus incurring no extra complexity to the recovery protocol.

#### 4 DEADLINE-ORDERED MULTICAST (DOM)

To describe DOM, we first specify the *safety* requirements for consensus. Given two replicas  $R_1$  and  $R_2$ , we say their states are consistent if the following 2 requirements are met.

- **Consistent ordering.** If  $R_1$  processes request  $a$  before request  $b$ , then  $R_2$  cannot process request  $b$  before request  $a$ , and vice versa.

- **Set equality.** If  $R_1$  processes request  $a$ , then  $R_2$  also processes request  $a$ , and vice versa.



**Figure 1: DOM design**

A vacuous solution to both safety requirements is to drop all requests, but that would not provide *liveness*. Besides, guaranteeing both safety properties and providing *liveness* is equivalent to building an atomic broadcast primitive, which is as hard as consensus itself [16]. Similar to MOM (used by Speculative Paxos) and OUM (used by NOPaxos), DOM’s goal is to judiciously move some responsibility for both safety and *liveness* to the DOM primitive, giving the protocol a leg up in the process. Unlike MOM/OUM, DOM does this without relying on in-network functionality. As shown in Figure 1, DOM consists of two roles, DOM-S (sender) and DOM-R (receiver). DOM’s key idea is to use clock synchronization between DOM-Ses and DOM-Rs to provide each request with a deadline specified in synchronized wall-clock time. DOM-Rs release requests in deadline order. This provides consistent ordering in the common case. We describe details below.

Every DOM-S can multicast its requests to all DOM-Rs. DOM-Rs will release the received requests, and the sequence of the requests later becomes the state of the replicas. DOM aims to create favorable conditions for consensus protocol, so it wants the sequences released by each DOM-R *close to* state consistency, if *consistent ordering* and *set equality* cannot be both fulfilled. More

specifically, DOM prioritizes consistent ordering over set equality if it cannot achieve both. To make that, DOM synchronizes the clocks for DOM-Ses and DOM-Rs. DOM-Ses and DOM-Rs leverage the synchronized clocks to achieve consistent ordering, and (possibly) set equality.

**DOM-S.** Each DOM-S estimates the one-way delay (OWD) to each DOM-R (the estimation approach is explained later). When DOM-S multicasts its request to DOM-Rs, it chooses the maximum among the estimated OWDs across all DOM-Rs. The maximum provides a latency bound: an estimate for how long it takes for the request to reach all replicas. DOM-S attaches the sending time and the latency bound to the request. Adding this latency bound to the sending time provides a deadline by when the request should have reached all DOM-Rs.

**DOM-R.** Each DOM-R maintains two buffers to hold the incoming requests, namely, *early-buffer* and *late-buffer*. *early-buffer* is a priority queue sorted by request deadlines, whereas *late-buffer* is a map which can be searched by the request index<sup>1</sup>. Requests which enter the *early-buffer* will be eventually released according to their deadline order, whereas the others will be kept in the *late-buffer*.

When DOM-R receives a request, DOM-R first considers whether it is eligible to enter the *early-buffer*. Not all requests can be accepted by the *early-buffer* because consistent ordering must be guaranteed for all requests released from *early-buffer*. The simplest eligibility check to enter the *early-buffer* is to compare the request's deadline with the DOM-R's local clock time at request arrival, and only accept the request if it arrives before its deadline. However, this entrance condition is unnecessarily strict, and will damage consistent ordering if DOM-R's clock does not satisfy monotonously increasing property. Instead, DOM-R uses a more relaxed condition that ensures the consistent ordering regardless of clock skew: if the deadline of the incoming request exceeds that of the last released one, it enters the *early-buffer*. Otherwise, DOM places this request into the *late-buffer* with no further actions. These requests will be further handled by Nezha protocol, to reach state consistency finally (§5.4).

**Adaptive latency bound.** During the interaction between DOM-Ses and DOM-Rs, the latency bound becomes the key factor to the effectiveness of DOM. A good latency bound should just cover the request delays from DOM-S to all DOM-Rs, so that the request can be accepted by their *early-buffers* but suffers little additional delay. The latency bound accounts for the variance of request delays, which mainly consists of two components: the processing delay of the request at DOM-R and the transmission delay from DOM-S to DOM-R. Though the public cloud is inherently more variable than bare-metal clusters, we empirically find that Nezha's latency bound can still handle this variability and outperform the baselines (§7.2), likely because of the inter-tenant performance isolation and rate limiting enforced by cloud operators [7, 31, 67].

DOM fully utilizes the synchronized clocks on DOM-S and DOM-R to estimate the request delay (i.e. OWD) as follows.

$$\widehat{OWD} = \begin{cases} M + \beta(\sigma_S + \sigma_R), & 0 < OWD < D \\ D \end{cases}$$

<sup>1</sup>We concatenate the *client-id* to *request-id* as the index for the request.

DOM-R maintains a sliding window for each DOM-S, and records the OWD samples by subtracting the attached sending time with its receiving time. Then DOM-R picks the median value from the samples in the sliding window as  $M$ . We previously tried moving average but found that just a few outliers (i.e. the tail latency samples) can inflate the estimated value. Therefore, we choose the moving median instead for robustness. Besides  $M$ , DOM-R also obtains from Huygens the standard deviation for the sending time and receiving time, denoted as  $\sigma_S$  and  $\sigma_R$ <sup>2</sup>.  $\sigma_S$  and  $\sigma_R$  provide an *approximate* error bound for the synchronized clock time, so we add the error bound with a factor  $\beta$  to  $M$  and obtain the final estimated OWD. The involvement of  $\beta(\sigma_S + \sigma_R)$  enables an adaptive increase of the estimated value, leading to a graceful degradation of Nezha as the clock synchronization performs worse.

Even after we have considered the robustness and graceful degradation, the possibility still exists that clock synchronization goes wrong and provides invalid OWD values (i.e. very large or even negative OWDs). To tackle this, we further adopt a clamping operation: If the estimated OWD goes out of a predefined scope  $[0, D]$ , we will use  $D$  as the estimated OWD.

The estimation runs as a separate routine on DOM-Rs. Periodically, DOM-Rs piggyback the estimated OWDs to DOM-Ses, and DOM-Ses choose the maximum OWDs across DOM-Rs as their latency bound for the following request multicast.

**Comparison to other primitives.** There are three other primitives closely related to DOM, namely, MOM [78], OUM [53] and TOQ [84]. From the perspective of deployability, DOM and TOQ are both based on software clock synchronization whereas MOM and OUM rely on highly engineered network. This gives DOM and TOQ an advantage over MOM/OUM in an environment like public cloud. On the other hand, MOM and TOQ only provide best-effort ordering, i.e., requests output from MOM and TOQ can still contain out-of-order cases. By contrast, DOM and OUM guarantees consistent ordering of requests. Regarding the comparison between DOM and OUM, OUM's gap detection mechanism is more sensitive to packet reordering/drop than the deadline-based holding mechanism of DOM. Whenever the incoming sequential numbers become non-consecutive, OUM triggers gap agreement even if consistent ordering is not violated, causing performance degradation of NOPaxos (§7.2). We also include a more formal comparison in §J.

## 5 THE NEZHA PROTOCOL

**Model and assumptions.** Nezha assumes a fail-stop model and does not handle Byzantine failures. As is typical in consensus, it uses  $2f + 1$  replicas: 1 leader and  $2f$  followers. Nezha guarantees correctness and liveness under the same assumptions as Multi-Paxos/Raft. However, Nezha's performance is improved by DOM, whose effectiveness depends on the accurate clock synchronization among VMs and the variance of OWDs between DOM-Ses and DOM-Rs. Here "accurate" means the clocks among DOM-Rs and DOM-Ses are synchronized with a small error bound *in most cases*, but Nezha does not assume a deterministic worst-case error bound.

**Replica state.** Figure 2 summarizes state variables maintained by the replica. Each replica is assigned with a unique *replica-id*, ranging

<sup>2</sup> $\sigma_S$  and  $\sigma_R$  are calculated based on the method in Appendix A of [27].

- *replica-id*— replica identifier  $(0, 1, \dots, 2f)$ .
- *status*— one of NORMAL, VIEWCHANGE, or RECOVERING.
- *crash-vector*— a vector of  $2f + 1$  counters initialized as 0s.
- *view-id*— the view identifier, initialized as 0 and increased after every view change.
- *early-buffer*— the priority queue provided by DOM, which sorts and releases the requests according to their deadlines.
- *late-buffer*— the map provided by DOM, and the requests are indexed by 2-tuple  $\langle \text{client-id}, \text{request-id} \rangle$  of the request.
- *log*— a list of client requests, appended in the order of request deadlines.
- *sync-point*— the latest synchronization point, until which the replica state (*log*) is consistent with the leader.

Figure 2: Local state of Nezha replicas

from 0 to  $2f$ . Replicas leverage a view-based approach [56]: each view is indicated by a *view-id*; the *view-id* is incremented each time a new leader is elected on a leader failure. In each view, only a single replica acts as the leader; its *replica-id* equals  $\text{view-id} \% (2f + 1)$ . Replicas tag all messages to other replicas with their current *view-ids*. When a replica's *status* is NORMAL, it ignores all messages with different *view-ids*. During the view change process (i.e., when a replica's *status* is VIEWCHANGE), replicas with different *view-ids* communicate with each other and eventually all replicas set their *status* to NORMAL. Replicas start in the NORMAL *status* when first booted up or in the RECOVERING *status* when rebooted after failures. This requires stable storage attached to a replica. This stable storage is only used during the replica reboot process, but both normal operation and recovery in Nezha are diskless. The *crash-vector* preserves Nezha's correctness from *stray messages* (§6).

**Request format.** A client's request message is represented as the 5-tuple:  $\text{request} = \langle \text{client-id}, \text{request-id}, \text{command}, s, l \rangle$ . *client-id* and *request-id* represent client and request identifiers, and they combine to uniquely identify a client's request on one replica. *command* represents the content of the request, which will be executed by the leader. The client will hand its request to DOM-S, which tags sending time  $s$  and the latency bound  $l$  to the request message, so the request's deadline is  $s + l$ .

**Maintaining consistent ordering.** Nezha installs a DOM-R on each replica. When clients submit requests, they are required to hand the requests to a DOM-S. Eventually, each replica can see the requests released by their DOM-Rs in consistent ordering. To process the released requests, each replica follows an *append-only* approach to add requests into its *log* in the fast path. However, the leader can force the followers to modify their *log* entries in the slow path. Nezha maintains the following invariant for consistent ordering: within each replica's *log*, requests are ordered by their deadlines.

Ordering happens automatically through DOM in the fast path, whereas the ordering is forced by the leader in the slow path.

**Sync messages.** The leader broadcasts *sync* messages to followers every time a request is appended to its *log*. The *sync* message is a 5-tuple:  $\text{sync} = \langle \text{view-id}, \text{client-id}, \text{request-id}, \text{deadline}, \text{log-id} \rangle$ . The *sync* messages serve 2 purposes. First, they can proactively correct set inequality (§5.4), making followers' *logs* consistent with the leader's *log*. Second, they doubles up as the leader's heartbeat. After failing to receive a leader's heartbeat for some duration, followers initiate a view change to elect a new leader. The *sync* message is much smaller than the *request* because it does not include the request body (i.e., the *command* field), and can be batched under high load. Among the 5 fields, *client-id* and *request-id* identify the request on each replica. *deadline* is the request's deadline decided by DOM-Ses on the fast path or overwritten by the leader on the slow path. *log-id* is the request's position in the leader's *log*.

## 5.1 Stateless Proxy Design

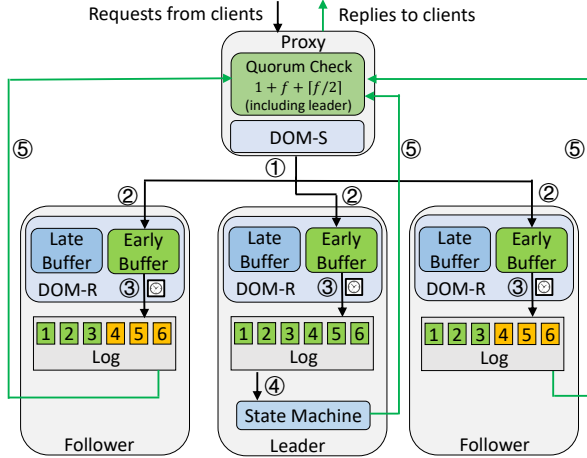
Since DOM-R has been installed on each replica, the follow-up consideration is where to install DOM-S, and furthermore, where to check the quorum of the replies corresponding to those requests multicast by DOM-Ses.

Multi-Paxos/Raft rely on the single leader to conduct request multicast and quorum check, which leads to the throughput bottleneck. An alternative design (e.g. Speculative Paxos, NOPaxos, CURP) is to offload the two tasks to the clients. It seems attractive because the offloading can mitigate the workload of the leader and also cut down the commit/execution latency. However, we note that such offloading design is not generally favorable. The reasons come from two aspects: (1) Deployment. Multi-Paxos/Raft are the *de facto* standard protocols widely applied in practice. Protocols with such offloading can hardly be deployed as a direct replacement of Multi-Paxos/Raft due to more intrusiveness to client logic. Even worse, the client VMs may be beyond control in many scenarios and clients do not allow installing extra libraries on their machines, thus making such protocols not deployable. (2) Throughput (of individual clients). Although the offloading design reduces the leader replica's burden, it can cause new bottlenecks to the client. As the number of replicas grows, the client needs to undertake more work to commit every request, and can easily become CPU-intensive, thus constraining its individual throughput (§7.6).

Considering the two main drawbacks, we introduce stateless proxies (shown in Figure 3-4) between replicas and clients which take over the 2 responsibilities instead of clients/replicas: (1) DOM-S, which includes clock synchronization and request multicast, (2) quorum check.

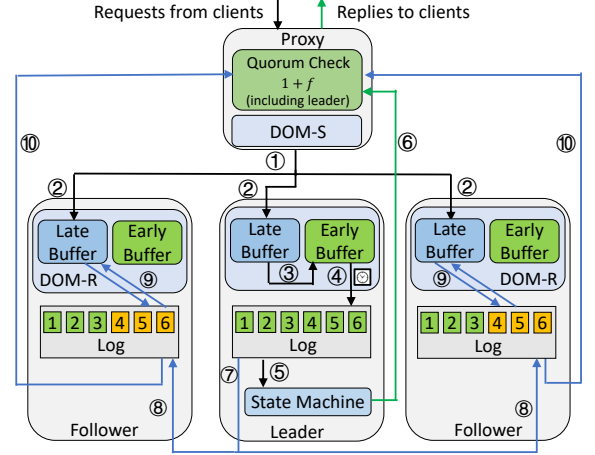
The proxy design allows the client to be unchanged from Multi-Paxos/Raft. To commit one request, the client just sends and receives one message regardless of the number of replicas. The proxy is a similar on-path device to the NOPaxos sequencer, but there are key differences. First, the stateless nature makes the proxy's failure cause no effect on the protocol correctness; a proxy failure is like a dropped request. By contrast, the NOPaxos sequencer is stateful and requires external entities (e.g., SDN controller) to guarantee its correct state (e.g., session number) after crashes. Second, since our proxy is stateless, its scalability and availability can both be





- ① The proxy broadcasts the request to all replicas via DOM-S, and the request is tagged with the sending time  $s$  and the latency bound  $l$ , summing up to be *deadline*.
- ② The request goes into Early Buffer if its *deadline* is larger than the last appended entry in Log (i.e. the last released entry from Early Buffer).
- ③ Replicas (DOM-Rs) release the request when the clock time has passed its *deadline*, and appends it to Log.
- ④ The leader replica executes the request.
- ⑤ Replicas send *fast-replies* (including a hash value of Log). The proxy considers the request as committed after receiving 1 leader's and  $f + [f/2]$  followers' replies with the same hash. Then the proxy acks a reply to the client.

Figure 3: Fast path of Nezha



- ① Same as ① in the fast path.
- ② The request goes into Late Buffer if its *deadline* is smaller than that of the last released request from Early Buffer.
- ③ The leader sets the request's *deadline* slightly larger than the last released request, and puts it into Early Buffer.
- ④ ⑤ ⑥ are the same as ③ ④ ⑤ in the fast path.
- ⑦ (In parallel with ⑤) The leader broadcasts the indices of log entries to followers.
- ⑧ Followers refer to the indices and add/delete entries to/from Log to keep consistent with the leader.
- ⑨ When missing entries, the follower first tries to recover it from Late Buffer. If it is still missing, then the follower fetches it from others.
- ⑩ Followers send *slow-replies*. The proxy considers the request as committed after receiving 1 leader's *fast-reply* and  $f$  followers' *slow-replies*. Then the proxy replies to client.

Figure 4: Slow path of Nezha

easily improved by adding extra proxies. In contrast, because the sequencer is stateful, it is much harder to add multiple sequencers, hindering both scalability and availability.

Note that request multicast and quorum check can both be concurrently executed. Therefore, we can employ a flexible number of proxies. Clients can choose any proxy and delegate that proxy to do the multicast and the corresponding quorum check. Different proxies work concurrently without mutual interference. The proper number of proxies depends on the real workload and can be dynamically adjusted by an auto-scaling entity [6, 30, 68, 80], which is outside the design scope of Nezha.

We are aware that the introduction of proxies unavoidably adds two additional message delays to commit requests. However, we believe the benefits (i.e. easy deployment and high throughput of individual client) outweigh the latency sacrifice. Besides, Nezha is also compatible to non-proxy design (i.e. co-locating the proxy with the client): If clients have abundant CPU resource and allow code modification, the non-proxy design is favored for better latency (as shown in §7.2).

Based on the proxy design, we next describe how proxies interact with replicas to commit requests in fast/slow path.

## 5.2 Fast Path: Quorum Check

As shown in Figure 3, if the request is multicast from DOM-S (①) and enters the *early-buffers* (②), it will be released by DOM-Rs according to the request's *deadline*, so that replicas can append the request to their *logs* (③). After that, followers immediately send a reply to the proxy without executing the request (⑤), whereas the leader first executes the request (④) and sends a reply including the execution result. We name this type of reply as *fast-reply*:

*fast-reply* = < view-id, request-id, replica-id, result, hash >

Inside the reply, the replica includes its current *view-id*, *replica-id*, and the *request-id* of the corresponding request. If the replica is the leader, it also includes the execution result of the request, else, *result* is *null*. The replica also includes an additional field, *hash*, to allow the proxy to check set equality of different replicas in the fast path. Since consistent ordering has already been guaranteed by DOM, set equality is sufficient to ensure the state consistency among replicas. The *hash* field performs a hash over the set of all previously appended entries using an incremental hash function [12]:

$$H_n = \bigoplus_{1 \leq i \leq n} h(\text{request}_i)$$

Here,  $h(*)$  is a standard hash function (we use SHA1) and  $\oplus$  is the XOR operation. To calculate  $h(request_i)$ , we concatenate the values of the request's *deadline*, *client-id*, *request-id* into a bitvector, and then transform it into a hash value.

To avoid *stray message* effect [63, 64], we also XOR  $H_n$  with the hash of *crash-vector* to get the final hash value:

$$hash_n = H_n \oplus h(crash-vector)$$

Here,  $h(crash-vector)$  is calculated by concatenating every integer in the vector and transforming it into a hash value.

The replica includes  $hash_n$  while sending the *fast-reply* for the  $n$ th request in its *log*. Assuming no hash collisions,  $hash_n$  represents the replica state when replicas reply to the proxy. By comparing the *hash* in the *fast-replies* from different replicas, the proxy can check if they have the same requests. Because the hash is computed over the set (but it represents the sequence) of entries, adding/deleting requests only requires incremental computation of XOR and  $h(*)$ . Hence, followers can rapidly fix set inequality and correct hashes in the slow path (§5.4), instead of recomputing hashes from scratch.

The proxy collects *fast-replies* for its request with matching *view-ids* and *request-ids*. It considers the request as committed after receiving the *fast-replies* from the leader and  $f + \lceil f/2 \rceil$  followers, with the same *hash*. The proxy also obtains the execution result from the leader's *fast-reply*, and then replies (with the execution result) to the related client.

The fast path requires a super quorum ( $f + \lceil f/2 \rceil + 1$ ) rather than a simple quorum ( $f + 1$ ) for the same reason as Fast Paxos and Speculative Paxos: without leader-follower communication, a simple quorum cannot persist sufficient information for the new leader to always distinguish committed requests from uncommitted requests. Consider what would happen if we had used a simple majority ( $f + 1$ ) in the fast path. Suppose there are two requests *req-1* and *req-2*, and *req-1* has a larger *deadline*. *req-1* is accepted by the leader and  $f$  followers. They send *fast-replies* to the proxy, and then the proxy considers *req-1* as committed and delivers the execution result to the client application. Meanwhile, *req-2* is accepted by the other  $f$  followers. After that, the leader fails, leaving  $f$  followers with *req-1* accepted and the other  $f$  followers with *req-2* accepted. Now, the new leader cannot tell which of *req-1* or *req-2* is committed. If the new leader adds *req-2* into the recovered *log*, it will be appended and executed ahead of *req-1* due to *req-2*'s smaller *deadline*. This violates linearizability: the client sees *req-1* executed before *req-2* with the old leader and sees the reverse with the new leader.

### 5.3 Fast Path: Commutativity Optimization

§5.2 shows that, *hash* consistency is the key condition to commit requests in fast path. However, as the throughput grows, we noticed a big drop in the fast commit ratio (FCR): equipped with the adaptive latency bound, the FCR dropped from 74.3 % to 10.3 % as the throughput grows from ~10K reqs/sec to ~60K reqs/sec in open-loop tests (evaluated with No-Commutativity variant in §7.4).

The reason for this drop is because the hash consistency check of *fast-replies* is *too sensitive* towards every request. Two replicas will fail to reply with consistent *hashes* so long as their DOM-Rs take different actions towards a single request (e.g., one DOM-R puts the request in *early-buffer* whether the other puts it in *late-buffer*). The

most straightforward way to improve FCR is to use larger latency bound, which can increase the likelihood that DOM-Rs put the request into the *early-buffer* everywhere, but it also delays requests further, undermining the benefit of the fast path.

To improve FCR as well as reduce the overall latency, Nezha considers commutativity as an optimization. We say two requests are commutative if changing their execution order does not affect their execution results [76]. In our context, this happens when the 2 requests access different pieces of state, typically different keys in a key-value store. Commutativity lets us optimize 2 parts of the fast path.

First, the entrance condition of the *early-buffer* can be relaxed to make requests enter *early-buffers* more often. Without commutativity, DOM prevents the incoming request from entering the *early-buffer* if its deadline is smaller than the last request released from the *early-buffer* (§4). DOM does this to preserve consistent ordering and avoid inconsistency of execution after replica failure and recovery. This is because if consistent ordering is violated and the leader fails, the new leader's log may have a different order of requests, so it will execute committed requests in a different order. However, the execution results of commutative requests are not affected by their order [76]. Hence, consistent ordering is only required among non-commutative requests, which enables a relaxation of the *early-buffer*'s entrance check: the request can enter the *early-buffer* if its deadline is larger than the last released request, *which is not commutative with the incoming request*.

Second, the hash computation can also exploit commutativity, which makes it easier to reach hash consistency among replicas. Since read requests do not modify replica state, the *hash* field in the *fast-reply* does not need to encode read requests. Besides, when encoding previous write requests, the *hash* field only considers those that are not commutative to the current request. To do that, Nezha maintains a table of per-key hashes for the write requests. For every newly appended write request, the replica will XOR its hash to update the corresponding per-key hash in the table according to its key. While sending the *fast-reply* for a specific request, the replica only includes the hash of the same key. For compound requests, which write (and hence do not commute with) multiple keys (e.g., "move 10 from  $x$  to  $y$  and return  $x$  and  $y$ "), the replica fetches the hashes of all relevant keys (e.g.  $x$  and  $y$ ), and includes the XORed hash value (e.g.  $hash_x \oplus hash_y$ ) in the *fast-reply*.

We have also evaluated the commutativity optimization across a range of workloads (Appendix §C), with different read/write ratio and skew factors. The result shows that the commutativity optimization helps reduce the latency by 7.7 %-28.9 %.

### 5.4 Slow Path: Set Equality

DOM only ensures that each replica appends requests in consistent ordering, but set equality may be violated. For example, the clocks might be running unusually fast on some replicas, causing them to prematurely release some requests with very large deadlines. Now, let's say a few subsequent requests arrive with *smaller* deadlines. These subsequent requests are put into *late-buffers* on these replicas because their deadlines are smaller than that of the prematurely released request. On the other hand, other replicas may have clocks running at the correct frequency, so the subsequent requests are

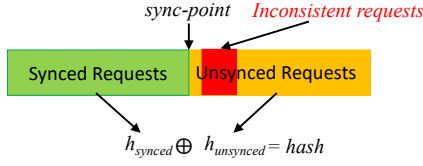


Figure 5: Illustration of *sync-status*

correctly placed in *early-buffers*. Hence, the same requests have ended up in *late-buffers* on some replicas and in *early-buffers* on others—violating set equality.

Set inequality will eventually be detected during the quorum check on the fast path due to hash inconsistency. But fixing set inequality this late inflates latency. Instead, Nezha starts fixing *potential* set inequality proactively—before it is detected by proxies. It does so by making the leader unilaterally broadcast log indices (*sync* messages): the leader takes its *log* as the reference and forces followers to modify their *logs* to match the leader. We describe the details next.

In Figure 4, when a request goes to the *late-buffer* because of its small deadline (②), followers do not handle it whereas the leader needs to pick it back to ensure progress (liveness), so the leader modifies the request’s deadline to make it larger than the last released request which is not commutative to this one. After the modification, this request becomes eligible to enter the *early-buffer* (③). After releasing and appending requests (④), the leader broadcasts its log indices to followers (⑦), to force followers to keep consistent *logs*. In parallel with the broadcast, the leader executes the request (⑤) and sends its *fast-reply* to the proxy (⑥).

Every time a follower receives a *sync* message (⑧) containing a *log-id*, it checks its log entry at the position *log-id*. (1) If the entry has the same 3-tuple  $\langle \text{client-id}, \text{request-id}, \text{deadline} \rangle$  as that included in the *sync* message, it means the follower has the same log entry as the leader at this position. (2) If only the 2-tuple  $\langle \text{client-id}, \text{request-id} \rangle$  is matched with that in the *sync* message, it means the leader has modified the deadline, so the follower also needs to replace the deadline in its entry with the deadline from the *sync* message. (3) Otherwise, the entry has different  $\langle \text{client-id}, \text{request-id} \rangle$ , which means the follower has placed a wrong entry at this position. In that case, the follower removes the wrong entry and tries to put the right one. It first searches its *late-buffer* for the right entry with matching  $\langle \text{client-id}, \text{request-id} \rangle$ . If the entry can be obtained locally, the follower modifies its deadline (if necessary) and puts it at this position (⑨). As a rare case, when the entry does not exist on this replica because the request was dropped, the follower fetches it from other replicas and puts it at the position. After processing the *sync* message, the follower updates its *sync-point*, indicating its *log* is consistent with the leader up to this point. The leader itself can directly advance its *sync-point* after appending the request to *log*.

Followers’ modifying *logs* based on *sync* messages will change the *hash* to be included in the following *fast-replies*. The incremental nature of the hash function enables a fast update of the hash. As shown in Figure 5, followers maintain two hash values during runtime, i.e.,  $h_{\text{synced}}$  and  $h_{\text{unsynced}}$ , to encode the set of synced and unsynced requests respectively. The follower’s unsynced set may include *inconsistent requests* (requests that are only on the leader

or follower, not both), causing set inequality. While sending the *fast-reply*, followers simply XOR  $h_{\text{synced}}$  and  $h_{\text{unsynced}}$  to get the final hash value to include in the *fast-reply*. The final hash value of a follower’s reply will be inconsistent with the leader’s reply if some inconsistent requests exist in the unsynced set. However, as more and more fresh *sync* messages are received from the leader replica, the followers can rapidly update  $h_{\text{synced}}$  and  $h_{\text{unsynced}}$  and move *sync-point* from left to right. This gradually eliminates inconsistent requests and generates consistent hashes for the requests appended later. The follower’s inconsistent requests, which do not exist on the leader, will be deleted.

## 5.5 Slow Path: Quorum Check

For every synced request (i.e., it has a smaller deadline than the *sync-point*), the follower also sends a reply to the client (⑩ in Figure 4). We name this type of reply as *slow-reply*:

$$\text{slow-reply} = \langle \text{view-id}, \text{request-id}, \text{replica-id} \rangle$$

Different from the *fast-reply*, which can be sent immediately after appending the request, the *slow-reply* can only be sent by followers after they ensure that all the previous requests up to this one have been synced with the leader. The 3-tuple of the *slow-reply* has the same meaning as in the *fast-reply*. *slow-reply* does not include *hash*, because it is definitely consistent with the leader. *slow-reply* does not include *result* because it is only sent by followers and followers do not execute requests.

The proxy considers the request as committed when it receives the related *fast-reply* from the leader and the *slow-replies* from  $f$  followers. The execution result is still obtained from the leader’s *fast-reply*. Unlike the quorum check of the fast path (§5.2), the slow path does not need a super quorum ( $1 + f + \lceil f/2 \rceil$ ). This is because, before sending *slow-replies*, the followers have updated their *sync-points* and ensured that all the requests (log entries) are consistent with the leader up to the *sync-points*. A simple majority ( $f + 1$ ) is sufficient for the *sync-point* to survive the crash. All requests before *sync-point* are committed requests, whose log positions have all been fixed. During the recovery (§6), they are directly copied to the new leader’s *log*.

Specially, a *slow-reply* can be used in place of the same follower’s *fast-reply* in the fast path’s super quorum, because it indicates the follower’s state is consistent with the leader. By contrast, the follower’s *fast-reply* cannot replace its *slow-reply* for the simple quorum of the slow path.

## 5.6 Timeout and Retry

The client maintains a timeout while waiting for the reply from the proxy. If the timeout is triggered (due to packet drop or proxy failure), the client retries the request with the same or different proxy (if the previous proxy is suspected of failure), and the proxy resubmits the request with a different sending time and (possibly) a different latency bound. Meanwhile, as in traditional distributed systems, replicas maintain *at-most-once* semantics. When receiving a request with duplicate  $\langle \text{client-id}, \text{request-id} \rangle$ , the replica simply resends the previous reply instead of appending/executing it twice.



## 6 RECOVERY

**Assumptions.** Nezha’s recovery protocol assumes replica processes can fail because of process crashes or a reboot of its server. When a replica process fails, it will be relaunched on the same server. However, we assume that there is some stable storage (e.g., disk) that survives process crashes or server reboots. A more general case, which we do not handle, is to relaunch the replica process from a different server with a new disk where the stable storage assumption no longer holds. We also do not handle the case of changing Nezha’s  $f$  parameter by adding or removing replicas from the system. Both cases are handled by the literature on reconfigurable consensus [56, 89], which we can adapt to Nezha as well.

**“Diskless” recovery.** A crashed replica can be relaunched but must recover its state before it resumes request processing. Raft [74] achieves this by persisting the replica’s information to stable storage before it replies to RPCs, so that the replica state can always be recovered from stable storage. Though it can tolerate some extreme cases (e.g., the simultaneous powering off of all replicas), such a disk-based design adds serious delays (0.5ms~20ms per write) to normal processing [56].

By contrast, Speculative Paxos and NOPaxos inherit the diskless recovery protocol from Viewstamped Replication [56], which improves performance by avoiding disk writes. However, this diskless recovery protocol was later shown to be incorrect [44, 63, 64] due to *stray messages* (defined in [44]): messages that are sent out but not delivered before a replica crashes. None of them can handle *stray messages* correctly (discussed in §H). Nezha also adopts the diskless recovery, but avoids the *stray message* effect by using *crash-vectors* [63, 64]. We explain more details in §A.

Despite its name, the diskless recovery protocol cannot be entirely independent of stable storage (disks). This is because the replica process needs to be aware whether this is its first launch or a relaunch. Existing literature either makes this assumption (i.e., the replica process already knows this) [63, 64], or relies on an oracle that can tell the replica process whether this is its first launch or a relaunch [44]. To implement either the assumption or oracle, we still need the stable storage.

To do that, Nezha maintains a *replica-id* field on the stable storage of each replica server, which is initialized to *null*. When the replica process is launched/relaunched, the real *replica-id* is passed in as an argument. The replica process then reads the value from its stable storage. A *null* value implies that this is the first run, so the replica process persists the *replica-id* to the stable storage and joins the system directly with *NORMAL status*. Otherwise, it finds *replica-id* already persisted in its stable storage, which implies the replica process has previously been launched but failed. Therefore, the replica process needs to first recover its state before joining the system. We note that disk operations are only required when a replica process starts. Both normal request processing and the rest of recovery do not involve any disk operations.

**Recovery protocol.** Nezha’s recovery protocol consists of two main components: replica rejoin and leader change. We describe the protocol with both pseudo-code in §A and TLA+ specification in §L. Here, we only sketch the major steps for the new leader to recover its state (*log*).

After the new leader is elected via the view change protocol, it contacts the other  $f$  survived replicas, acquiring their *logs*, *sync-points* and *last-normal-views* (i.e., the last view in which the replica’s status is *NORMAL*). Then, it recovers the *log* by aggregating the *logs* of those replicas with the largest *last-normal-view*. The aggregation involves two key steps.

(1) The new leader chooses the largest *sync-point* from the qualified replicas (i.e., the replicas with the largest *last-normal-view*). Then the leader directly copies all the *log* entries up to the *sync-point* from that replica.

(2) For the remaining part, if the *log* entry has a larger *deadline* than the *sync-point*, the leader checks whether this entry exists on  $\lceil f/2 \rceil + 1$  out of the qualified replicas. If so, the entry will also be added to the leader’s *log*. All the entries are sorted by their *deadlines*.

After the leader rebuilds its *log*, it executes the entries in their *deadline* order. It then switches to *NORMAL status*. After that, the leader distributes its rebuilt *log* to followers. Followers replace their original *logs* with the new ones, and also switch to *NORMAL*.

**Acceleration of Recovery.** Since Nezha only allows the leader to execute requests during normal processing, the recovery can be inefficient when the leader fails, because the new leader has to execute all requests from scratch. To optimize this, we adopt a similar idea as NOPaxos [53] and conduct synchronization in the background. Periodically, the followers report their *sync-points* to the leader, and the leader chooses the smallest *sync-point* among the  $f + 1$  replicas as the *commit-point*, and broadcasts the *commit-point* to all replicas, so that followers can safely execute the requests up to the *commit-point*. Thus, when the leader fails, the new leader only needs to execute the requests from its *commit-point* onwards.

**Correctness.** We can prove that the recovery procedure guarantees the following correctness properties. We include the correctness proof in Appendix §B. We also include a model-checked TLA+ specification of Nezha protocol in Appendix §L.

- **Durability:** if a client considers a request as committed, the request survives replica crashes.
- **Consistency:** if a client considers a request as committed, the execution result remains unchanged after the replica’s crash and recovery.
- **Linearizability:** A request appears to be executed exactly once between start and completion.

## 7 EVALUATION

We aim to answer the following questions during the evaluation:

- (1) How does Nezha compare to the baselines (Multi-Paxos, Fast Paxos, NOPaxos in public cloud)?
- (2) How does Nezha compare to the recent protocols which also use clock synchronization (i.e., Domino and TOQ-based EPaxos)?
- (3) How effective are the proxies, especially when there is a large number of replicas?
- (4) How fast can Nezha recover from the leader failure?
- (5) How does Nezha compare to Raft when both are equipped with log persistence to stable storage?
- (6) Does Nezha provide sufficient performance for replicated applications?

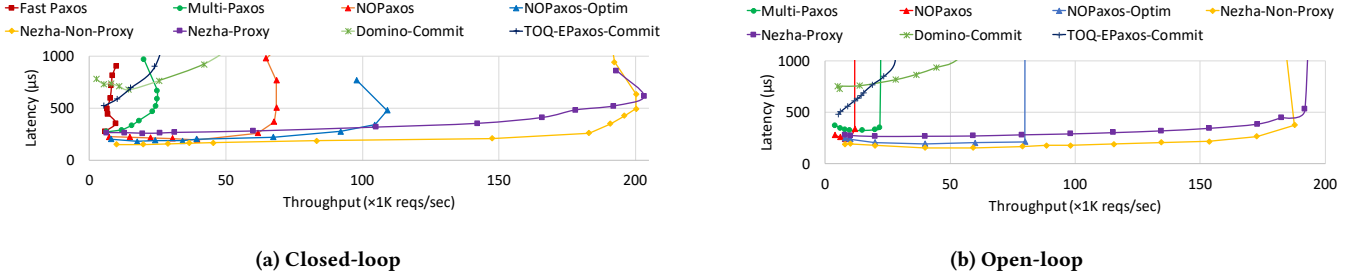


Figure 6: Latency vs. throughput

## 7.1 Settings

**Testbed.** We run experiments in Google Cloud. We employ n1-standard-4 VMs for clients, n1-standard-16 VMs for replicas and the NOPaxos sequencer, and n1-standard-32 VMs for Nezha proxies. All VMs are placed in the same cloud zone. Huygens is installed on all VMs and achieves an average 99th percentile clock offsets of 49.6 ns.

**Baselines.** We compare with Multi-Paxos, Fast Paxos and NOPaxos. For all 3 baselines, we use the implementation from the NOPaxos repository [52] with necessary modification: (1) we change multicast into multiple unicasts because network-support multicast is unavailable in cloud. (2) we use a software sequencer for NOPaxos because tenant-programmable switches are not yet available in cloud.

Besides, we also add two recently proposed protocols into comparison, i.e., Domino [94] and TOQ-based EPaxos [84]. We choose to compare them with Nezha because they also use clock synchronization to accelerate consensus. However, since both protocols target at Wide Area Network (WAN), it is unfair to expect them to perform better than Nezha in LAN, which is verified by our experiments (§7.2). We also intend to compare Nezha with Derecho [40], but its performance degrades a lot in public cloud (see details in §F). We think the comparison is unfair to Derecho and do not include it.

**Metrics.** We target at execution latency [84] instead of commit latency as one metric because clients can use results from a consensus protocol only after the execution is completed; we use throughput as the other metric. To measure latency, we use median latency because it is more robust to heavy tails. We have attempted to measure tail latency at the 99th and 99.9th percentile. But we find it hard to reliably measure these tails because tail latencies within a cloud zone can exceed a millisecond [38, 69, 91]. This is unlike the WAN setting where tails can be more reliably estimated [84]. We run each experiment 5 times and average values before plotting.

**Evaluation method.** We follow the method of NOPaxos [53] and run a *null application* with no execution logic. Traditional evaluation of consensus protocols [53, 65, 71, 72, 78, 87] use closed-loop clients, which issue a continuous stream of back-to-back requests, with exactly one outstanding request at all times. However, the recent work [84] suggests a more realistic open-loop test with a Poisson process where the client can have multiple outstanding requests (sometimes in bursts). We use both closed-loop and open-loop tests. While comparing the latency and throughput in §7.2, we use 3

replicas. For the closed-loop test, we increase load by adding more clients until saturation<sup>3</sup>. For the open-loop test, we use 10 clients and increase load by increasing the Poisson rate until saturation.

**Workloads.** Since the three baselines (Multi-Paxos, Fast Paxos and NOPaxos) are oblivious to the read/write type and commutativity of requests, and the *null application* does not involve any execution logic, we simply measure their latency and throughput under one type of workload, with a read ratio of 50 % and a skew factor [33] of 0.5. We also evaluate Nezha under various read ratios and skew factors in Appendix §C, which verifies the robustness of its performance.

## 7.2 Nezha vs. Multi-Paxos, Fast Paxos, NOPaxos

The closed-loop and open-loop evaluation results are shown in Figure 6. We plot two versions of Nezha. Nezha-Proxy uses standalone proxies whereas Nezha-Non-Proxy lets clients undertake proxies’ work. Below we discuss three main takeaways.

First, All baselines yield poorer latency and throughput in public cloud, in comparison with published numbers from highly-engineered networks [53]. Fast Paxos suffers the most<sup>4</sup> and reaches only 4.0K reqs/second at 425 μs in open-loop test (not shown in Figure 6b). When clients send at a higher rate, Fast Paxos suffers from reordering, and the reordered requests force Fast Paxos into its slow path.

Second, NOPaxos performs unexpectedly poorly in the open-loop test, because it performs *gap handling* and *normal request processing* in one thread. NOPaxos *early binds* the sequential number with the request at the sequencer. When request reordering/drop inevitably happens from the sequencer to replicas, the replicas trigger much gap handling and consume most CPU cycles. We realize this issue and develop an optimized version (NOPaxos-Optim in Figure 6) by using separate threads for the two tasks. NOPaxos-Optim outperforms all the other baselines because it offloads request serialization to the sequencer and quorum check (fast path) to clients. But it still loses significant throughput in the open-loop test compared with the closed-loop test. This is because open-loop tests create more bursts of requests, and cause packet reordering/drop more easily. Besides, the gap handling mechanism in NOPaxos causes *long-lasting effect*, which is more distinct in open-loop tests. During the closed-loop test, clients will not continue to

<sup>3</sup>Specially, when the system is overloaded, the throughput can drop instead of continuously increasing [18, 88], as shown in Figure 6.

<sup>4</sup>Speculative Paxos can hardly work in public cloud, so we do not include it in our comparative evaluation.

submit new requests when the replicas are conducting gap handling. Therefore, the latency measurement of newly submitted requests does not count the gap handling time. By contrast, open-loop clients continue to submit requests when replicas are suffering from gaps. Even if the subsequent requests all arrive in order, they cannot be processed before the gap handling is complete, so these requests will also count the gap handling cost into their latencies. Besides, the newly submitted requests can continue to add burden to the replicas cause reordering/drop (leading to more gap handling work), which worsens the performance in open-loop tests. As an example, when 10 open-loop clients submit 10K reqs/sec each, NOPaxos replicas trigger gap handling for more than 30% of requests.

Last, Nezha achieves much higher throughput than all the baselines, and Nezha-Non-Proxy also achieves the lowest latency because of co-locating proxies with clients. Even equipped with standalone proxies, Nezha-Proxy still outperforms all baselines at their saturation throughputs, except NOPaxos-Optim (open-loop). Nezha’s outperformance in throughput/latency comes from three design aspects: (1) DOM helps create consistent ordering for the replication protocol, and makes it easier for replicas to achieve consistency. (2) Nezha leverages clients/proxies to undertake quorum check, which effectively relieves leader’s burden. (3) The use of commutativity further reduces the latency by allowing more requests to be committed in fast path. To further verify the benefit of each component, we conduct an ablation study in §7.4.

### 7.3 Nezha vs. Domino, TOQ-based EPaxos

As for Domino and TOQ-based EPaxos, we intend to plot their execution latencies in Figure 6 instead of commit latencies for fair comparison, because the two protocols have both decoupled commit from execution, and their commit replies do not include execution results. However, Domino’s execution latency reaches more than 10 ms and TOQ-based EPaxos’ execution latency ranges from 1.3 to 3.3 ms, which are far larger than Nezha (and other baselines) and not suitable to be shown in the same figure, so we plot their commit latencies instead. But regarding the commit latency, Domino and TOQ-based EPaxos still perform worse than Nezha. The performance gap is partially related to the difference of implementation (Domino and TOQ-based EPaxos are implemented in Golang with grpc [82, 93] whereas the other protocols are implemented in C++ with UDP socket [52]). Apart from that, below we explain Nezha’s advantages over the two protocols from the design perspective.

**Nezha vs. Domino.** Nezha distinguishes from Domino in three main aspects. (1) Unlike Nezha, Domino decouples commit from execution and its execution, which makes it hard to become a drop-in replacement for Paxos/Raft, where applications can direct get the execution result from the commit reply. (2) Domino chooses a more conservative estimation of OWD (95th percentile), otherwise, it falls back to an under-optimized slow path. By contrast, Nezha leverages commutativity to improve fast commit ratio, which enables Nezha to choose a smaller latency bound. Even when Nezha falls back into its slow path, it can rapidly fix the inconsistency with only 1 extra message delay. (3) Domino directly uses clock time as the log positions of requests, which brings two problems. One, Domino

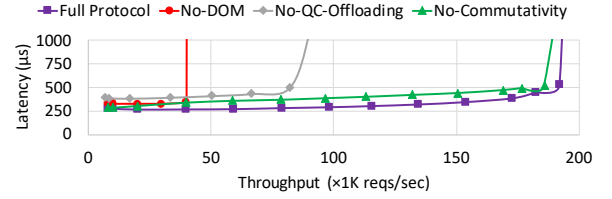


Figure 7: Ablation study

needs to fill no-ops for empty log positions, which incurs non-trivial overheads, especially when two consecutive requests have large time intervals between them. Two, when a request arrives at Domino replicas later than its timestamp, Domino is expected to reject it and trigger the slow path. But if clock skew occurs at this moment, the replicas will accept the requests and make the client consider the request as committed, though this request is later replaced by a no-op. Such permanent loss of committed requests is known as durability violation [76], which further leads to the violation of linearizability. We include detailed explanation with error traces in Appendix §G. Compared with the “timestamp-as-log-position” design, Nezha uses deadline to reduce packet reordering, and the log positions of requests are eventually decided by the replicas. Therefore, Nezha’s correctness is independent of clock skew.

**Nezha vs. TOQ-based EPaxos.** The previous work [4] has observed that EPaxos performs worse than Multi-Paxos in LAN. Our results in Figure 6 continue to show that the conclusion still holds even after the enhancement of TOQ. Nezha gains advantages over TOQ-based EPaxos from two design aspects. (1) TOQ only mitigates the conflict rate for EPaxos but does not optimize the message flow of EPaxos. When it comes to LAN, where there are no distinguishment between LAN RTTs and WAN RTTs, EPaxos *inherently* causes more RTTs than Nezha. Besides, there is no proxies to assist EPaxos replicas in request multicast and quorum check, so its replicas’ burden is heavier than Nezha. (2) Although both Nezha and TOQ-based EPaxos use commutativity to reduce conflict, Nezha starts the conflict check early in the primitive (DOM), whereas the other primitive (TOQ) is oblivious to the conflicts: TOQ adds the pending delay to all requests, and the requests’ conflict can only be detected after the requests are released and handed to the EPaxos protocol.

### 7.4 Ablation Study

During the ablation study of Nezha, we remove one component from the full protocol of Nezha each time, and yield three variants, shown as No-DOM, No-QC-Offloading, No-Commutativity in Figure 7. No-DOM variant removes the DOM primitive from Nezha. No-QC-Offloading variant relies on the leader replica to do the quorum check, and it still relies on DOM for consistent ordering (the proxies with DOM-Ses still conduct request mutlicast). No-Commutativity variant disables Nezha’s commutativity optimization. We run all protocols under the same setting as Figure 6b.

Figure 7 shows that, removing any of the three components can degrade the performance (i.e., throughput and/or latency).

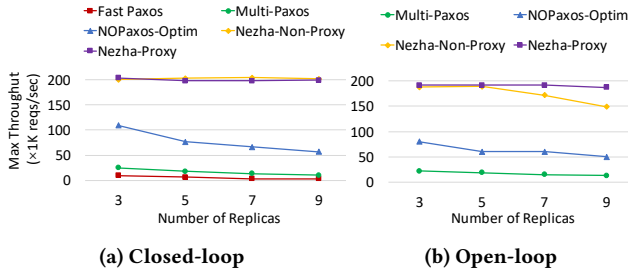


Figure 8: Max throughput vs. number of replicas

(1) The No-DOM variant makes the fast path meaningless, because consistent ordering is no longer guaranteed and set equality (i.e. reply messages with consistent hash) no longer indicates the state consistency among replicas. In this case, the No-DOM variant actually becomes the Multi-Paxos protocol with quorum check offloading, and the leader replica still takes the responsibility of ordering and request multicast, which makes No-DOM variant only yields a much lower throughput and higher latency.

(2) The No-QC-Offloading variant still uses DOM for ordering and request multicast, but it relies on the leader to do quorum check for every request. Therefore, the leader’s burden becomes much heavier than the full protocol, and the heavy bottleneck at the leader replica degrades the throughput and latency performance.

(3) The No-Commutativity variant degrades the fast commit ratio and causes more requests to commit via the slow path. It does not cause distinct impact on the throughput. However, compared with the full protocol, the lack of commutativity optimization degrades the latency performance by up to 32 %.

## 7.5 Scalability

Figure 8 shows that, Nezha achieves much higher throughput than all baselines with different number of replicas. However, in open-loop tests with only 10 clients (Figure 8b), the throughput of Nezha-Non-Proxy distinctly degrades from 187.8K reqs/sec to 148.7K reqs/sec, as the number of replicas grows. This indicates that the clients become the new bottleneck when submitting at high rates. By contrast, when equipped with proxies, Nezha-Proxy maintains a high throughput regardless of the number of replicas. We continue to evaluate the proxy design in §7.6.

## 7.6 Proxy Evaluation

Figure 9a and Figure 9b compare the two versions of Nezha with 10 open-loop clients and 9 replicas. Nezha-Proxy also employs 5 proxies. As the client increases its submission rate, we measure the latency and the average CPU utilization per client. Compared with Nezha-Non-Proxy, which sends 9 messages and receives 17 messages (i.e., 9 *fast-replies* and 8 *slow-replies*) for each request, Nezha-Proxy incurs 2 extra message delays, but reduces significant CPU cost at the client side. It achieves even lower latency as the throughput grows, because Nezha-Non-Proxy makes the clients CPU-intensive.

Figure 9c compares the maximum throughput achieved by individual client with/without proxies. Given the same CPU

resource<sup>5</sup>, the throughput of the single client without proxies declines distinctly as the number of replicas increases. Such bottlenecks can also occur in the other works with similar offloading design (e.g., Speculative Paxos, NOPaxos, Domino, CURP). By contrast, when equipped with proxies, the individual client remains a high throughput regardless of the number of replicas.

## 7.7 Failure Recovery

We evaluate the failure recovery as shown in Figure 10 and Figure 11. Since follower’s crash and recovery do not affect the availability of Nezha, we mainly focus on the evaluation of the leader’s crash and recovery. We study two aspects: (1) How long does it take for the remaining replicas to complete a view change with the new leader elected? (2) How long does it take to recover the throughput to the same level as before crash?

We maintain 3 replicas and 10 open-loop clients, and vary per-client submission rate from 1K reqs/sec to 20K reqs/sec, so the total submission rate varies from 10K reqs/sec to 200K reqs/sec. Under different submission rates, we kill the leader and measure the time cost of view change. As shown in Figure 10, the time cost grows as the submission rate increases, because there is an increasing amount of state (log) transfer to complete the view change. But the time cost of view change is generally low (150 ms–300 ms), because of the acceleration idea (§6) integrated in Nezha.

The time cost to recover the same throughput level (Figure 11) is larger than the time cost of view change, because there are other necessary works to do after the replicas enter the new view. For example, replicas need to relaunch the working threads and reinitialize the contexts; replicas need to handle clients’ retried requests, which fail to be responded before crash; followers may need additional state transfer due to lagging too far behind, etc.

Based on the measured trace, we calculate the throughput every 10 ms, and plot the data points in Figure 11. Figure 11 implies that the recovery time is related to the throughput level to recover. A lower throughput level takes a shorter time to recover, and vice versa. It takes approximately 0.7 s, 1.9 s, 4.0 s, to recover to the same throughput level under the submission rate of 20K reqs/sec, 100K reqs/sec, 200K reqs/sec, respectively. As a reference to compare, Figure 3.20 in [86] evaluates the recovery time for an industrial Raft implementation [34], which takes about 6 seconds to recover to 18K reqs/sec.

## 7.8 Nezha vs. Raft

Raft establishes its correctness on log persistence and relies on the stable storage for stronger fault tolerance (e.g. power failure). For a fair comparison to Raft, we convert Nezha from its diskless operation to a disk-based version, making it achieve the same targets as Raft. Before Nezha replicas send replies, they first persist the corresponding log entry (including *view-id* and *crash-vector*) to stable storage. Then, if a replica is relaunched, it can recover its state and replay the *fast-replies/slow-replies*. We want to study whether Nezha is fundamentally more I/O intensive than Raft.

We initially use the original Raft implementation [74] (shown as Raft-1 in Figure 12), which is written in C++, but uses a slower

<sup>5</sup>Every client uses one thread for request submission and the other thread for reply handling.



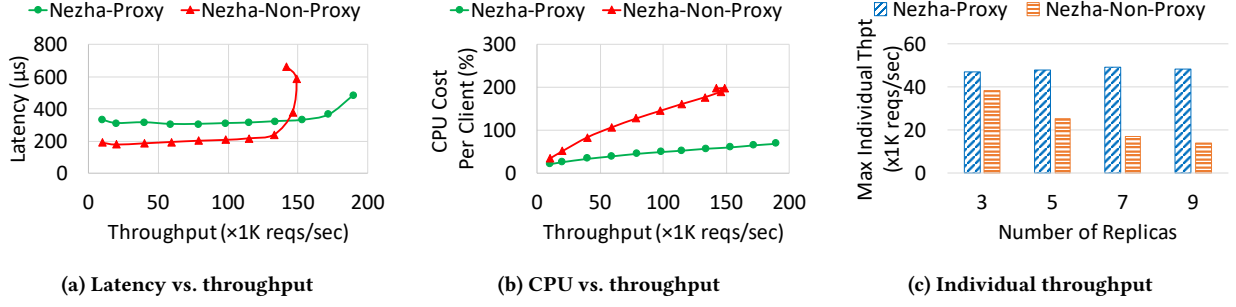


Figure 9: Proxy Evaluation

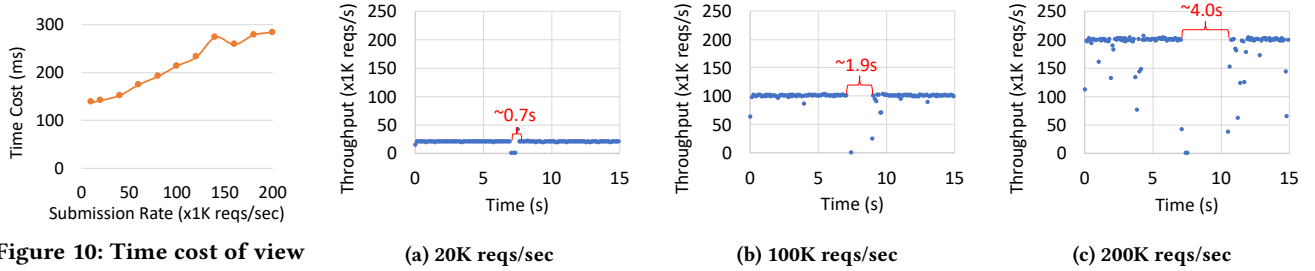


Figure 10: Time cost of view change

Figure 11: Time cost to recover to the same throughput level

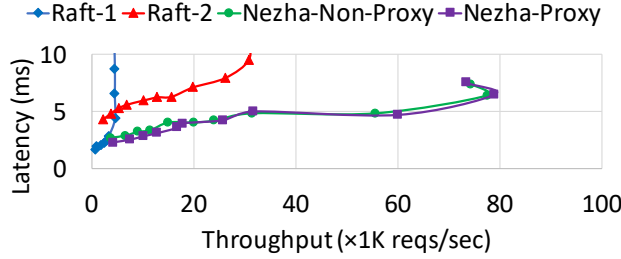


Figure 12: Nezha vs. Raft (closed-loop)

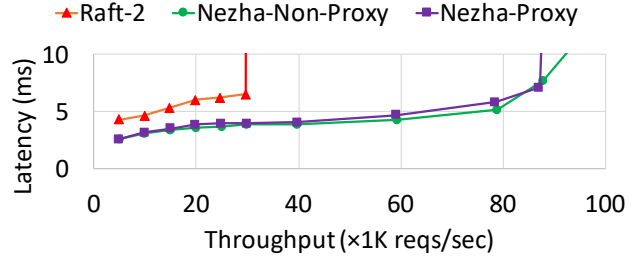


Figure 13: Nezha vs. Raft (open-loop)

communication library based on TCP, and involves additional mechanisms (e.g. snapshotting). Raft-1 can only work in closed-loop tests because of its blocking API. For Raft-1, we use its default batching and pipeline mechanism, and noticed that Raft-1 achieves very low throughput of 4.5K reqs/sec on Google Cloud VMs equipped with zonal standard persistent disk [32]. Hence, we implement and optimize Raft (shown as Raft-2 in Figure 12 and Figure 13), by using the Multi-Paxos code from [52] as a starting point. For both Raft-2 and Nezha, we tune their batch sizes to reach the best throughput. Our evaluation shows that Nezha outperforms Raft in both closed-loop and open-loop tests.

## 8 APPLICATION PERFORMANCE

**Redis.** Redis [79] is a typical in-memory key-value store. We choose YCSB-A [92] as the workload, which operate on 1000 keys with HMSET and HGETALL. We use 20 closed-loop clients to submit requests, which can saturate the processing capacity of the unreplicated Redis. Figure 14 illustrates the maximum throughput of each protocol under 10 ms SLO. Nezha outperforms all the baselines

on this metric: it outperforms Fast Paxos by 2.9 $\times$ , Multi-Paxos by 1.9 $\times$ , and NOPaxos by 1.3 $\times$ . Its throughput is within 5.9% that of the unreplicated system.

**CloudEx.** CloudEx [29] is a research fair-access financial exchange system for public cloud. There are three roles involved in CloudEx: matching engine, gateways and market participants. To provide fault tolerance, we replicate the matching engine and co-locate one gateway with one proxy. Market participants are unmodified. Before porting it to Nezha, we have improved the performance of CloudEx, compared with the version in [29], by multithreading and replacing ZMQ [95] with raw UDP transmission. We first run the unreplicated CloudEx with dynamic delay bounds disabled [29]. We configure a fixed sequencer delay parameter ( $d_s$ ) of 200 $\mu s$ . Similar to [29], we launch a cluster including 48 market participants and 16 gateways, with 3 participants attached to one gateway. The matching engine is configured with 1 shard and 100 symbols. We vary the order submission rate of market participants, and find the matching engine is saturated at 43.10K orders/sec, achieving an inbound unfairness ratio of 1.49%.



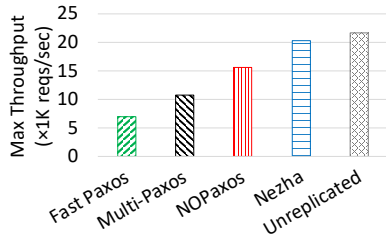


Figure 14: Redis throughput

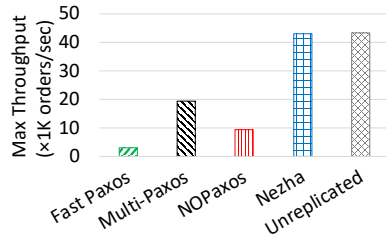


Figure 15: CloudEx throughput

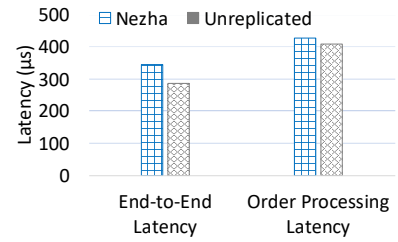


Figure 16: CloudEx latency

We then run CloudEx atop the four protocols with the same setting. In Figure 15, only Nezha reaches the throughput (42.93K orders/sec) to nearly saturate the matching engine, and also yields a close inbound unfairness ratio of 1.97%. We further compare the end-to-end latency (i.e., from order submission to the order confirmation from the matching engine) and order processing latency (i.e., from order submission to receiving the execution result from the matching engine.) between Nezha and the unreplicated CloudEx. In Figure 16, Nezha prolongs the end-to-end latency by 19.7% (344  $\mu$ s vs. 288  $\mu$ s), but achieves very close order processing latency to the unreplicated version (426  $\mu$ s vs. 407  $\mu$ s).

## 9 RELATED WORK

**Classic protocols.** Paxos [46, 49] has nearly become synonymous with consensus since its proposal. Most practical deployments use Multi-Paxos [49], which maintains a stronger leader role than basic Paxos. Viewstamped Replication (VR) [56, 73] provides equivalent functionality to Multi-Paxos [5, 53, 63, 64, 78], and Raft [74] builds on VR to improve understandability relative to Multi-Paxos. Multi-Paxos, VR and Raft all share the similar workflow, which requires 2 RTTs to commit all requests; further, the use of a single leader, while simplifying the design, makes the leader a bottleneck.

**Recent protocols.** Mencius [61] exploits a multi-leader design to mitigate the single leader bottleneck. However, it introduces extra coordination cost among multiple leaders and suffers from straggler effect. The crash of any of the leaders will stop the progress. Unlike that, Nezha reduces the leader’s bottleneck with proxies and followers’ crash does not affect progress. CURP [76] can complete commutative requests in 1 RTT, but it does not use the ordering information, and costs up to 3 RTTs even if all witnesses process the non-commutative requests in the same order. SPaxos [13], BPaxos [90] and Compartmentalized Paxos [65] address the throughput scaling of consensus protocols with modularity, trading more latency for throughput improvement. The proxy design in Nezha is similar to compartmentalization [65], but Nezha’s proxies are stateless. By contrast, [65] also decomposes stateful entities, which complicates the recovery protocol and affects scalability.

**Clock synchronization applied to consensus protocols.** CRaft [86] and CockroachDB [83] use clock synchronization to improve the throughput of Raft. However, they base their correctness on the assumption of a known worst-case clock error bound, which is not practical for clock synchronization [55, 57, 58]. Domino [94] and TOQ [84] try using clock synchronization to accelerate Fast Paxos and EPaxos respectively. We evaluate and compare them

with Nezha in §7.3. Besides, Domino’s correctness can be damaged due to clock skew, and we include detailed explanation in §G.

## 10 CONCLUSION

Recent development of accurate software clock synchronization techniques brings us new opportunities to develop novel consensus protocols to achieve high performance in public cloud. Leveraging this, we present Nezha in the paper, which can be easily deployed in the public cloud, and achieves both higher throughput and lower latency than baselines.

## REFERENCES

- [1] [n.d.]. Derecho Discussion Issue 237. <https://github.com/Derecho-Project/derecho/discussions/237>.
- [2] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xyglis, and Igor Zablotchi. 2020. Microsecond Consensus for Microsecond Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 599–616. <https://www.usenix.org/conference/osdi20/presentation/aguilera>
- [3] Byagowi Ahmad and Obleukhov Oleg. 2021. Open-sourcing a more precise time appliance. <https://engineering.fb.com/2021/08/11/open-source/time-appliance/>.
- [4] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. 2019. Dissecting the Performance of Strongly-Consistent Replication Protocols. In *Proceedings of the 2019 International Conference on Management of Data (Amsterdam, Netherlands) (SIGMOD ’19)*. Association for Computing Machinery, New York, NY, USA, 1696–1710. <https://doi.org/10.1145/3299869.3319893>
- [5] Ganesan Aishwarya, Alagappan Ramnathan, C. Arpacı-Dusseau Andrea, and H. Arpacı-Dusseau Remzi. 2021. Exploiting Nil-Externality for Fast Replicated Storage. In *Proceedings of the Twenty-Eighth ACM Symposium on Operating Systems Principles (SOSP ’21)*. Association for Computing Machinery, New York, NY, USA, 17.
- [6] Amazon. [n.d.]. Cluster Autoscaler - Amazon EKS. <https://docs.aws.amazon.com/eks/latest/userguide/cluster-autoscaler.html>. (Accessed on 06/01/2020).
- [7] Amazon. [n.d.]. Infrastructure security in Amazon EC2. <https://docs.aws.amazon.com/AWSEC2/latest/WindowsGuide/infrastructure-security.html>.
- [8] Amazon. [n.d.]. Introducing the Amazon Time Sync Service. [https://aws.amazon.com/about-aws/whats-new/2017/11/introducing-the-amazon-time-sync-service/?nc1=h\\_ls](https://aws.amazon.com/about-aws/whats-new/2017/11/introducing-the-amazon-time-sync-service/?nc1=h_ls).
- [9] Apache Software Foundation. [n.d.]. ZooKeeper. <https://zookeeper.apache.org>.
- [10] Hagit Attiya and Jennifer L. Welch. 1994. Sequential Consistency versus Linearizability. *ACM Trans. Comput. Syst.* 12, 2 (may 1994), 91–122. <https://doi.org/10.1145/176575.176576>
- [11] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jun Song. 2020. Virtual Consensus in Delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 617–632. <https://www.usenix.org/conference/osdi20/presentation/balakrishnan>
- [12] Mihir Bellare and Daniele Micciancio. 1997. A New Paradigm for Collision-Free Hashing: Incrementality at Reduced Cost (*EUROCRYPT’97*). Springer-Verlag, Berlin, Heidelberg, 163–192.
- [13] Martin Biely, Zarko Milosevic, Nuno Santos, and André Schiper. 2012. S-Paxos: Offloading the Leader for High Throughput State Machine Replication. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*. 111–120. <https://doi.org/10.1109/SRDS.2012.66>
- [14] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *7th USENIX Symposium on Operating Systems Design and*

- Implementation (OSDI).
- [15] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing* (Portland, Oregon, USA) (PODC '07). Association for Computing Machinery, New York, NY, USA, 398–407. <https://doi.org/10.1145/1281100.1281103>
  - [16] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. 1996. The Weakest Failure Detector for Solving Consensus. *J. ACM* 43, 4 (jul 1996), 685–722. <https://doi.org/10.1145/234533.234549>
  - [17] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2021. PigPaxos: Devouring the Communication Bottlenecks in Distributed Consensus. In *Proceedings of the 2021 International Conference on Management of Data*. Association for Computing Machinery, New York, NY, USA, 235–247. <https://doi.org/10.1145/3448016.3452834>
  - [18] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. 2020. Overload Control for  $\mu$ -scale RPCs with Breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 299–314. <https://www.usenix.org/conference/osdi20/presentation/cho>
  - [19] Debian community. [n.d.]. ramfs. <https://wiki.debian.org/ramfs>.
  - [20] 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* (Santa Clara, California) (SOSR '15). Association for Computing Machinery, New York, NY, USA, Article 5, 7 pages. <https://doi.org/10.1145/2774993.2774999>
  - [21] Ben Darnell. [n.d.]. Scaling Raft. <https://www.cockroachlabs.com/blog/scaling-raft>.
  - [22] Perrine Dean. [n.d.]. NYI Launches High Precision Time Service, Powered by Perseus, at Its Data Centers. <https://www.jsa.net/blog/nyi-launches-high-precision-time-service-powered-by-perseus-at-its-data-centers/16213/>
  - [23] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Comput. Surv.* 36, 4 (dec 2004), 372–421. <https://doi.org/10.1145/1041680.1041682>
  - [24] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1–14. <https://www.usenix.org/conference/atc19/presentation/duplyakin>
  - [25] Inc. Equinix. [n.d.]. Equinix Precision Time. <https://docs.equinix.com/en-us/Content/Edge-Services/EPT/EPT.htm>.
  - [26] etcd. [n.d.]. A distributed, reliable key-value store for the most critical data of a distributed system. <https://etcd.io>.
  - [27] Yilong Geng. 2018. *Self-Programming Networks: Architecture and Algorithms*. Ph.D. Dissertation. Stanford University. <https://www.proquest.com/dissertations-theses/self-programming-networks-architecture-algorithms/docview/2438700930/se-2?accountid=14026>
  - [28] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. 2018. Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, USA) (NSDI'18). USENIX Association, Berkeley, CA, USA, 81–94.
  - [29] Ahmad Ghalayini, Jinkun Geng, Vignesh Sachidananda, Vinay Sriram, Yilong Geng, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. 2021. CloudEx: A Fair-Access Financial Exchange in the Cloud. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. 8. <https://doi.org/10.1145/3458336.3465278>
  - [30] Google. [n.d.]. Cluster autoscaler | Kubernetes Engine Documentation | Google Cloud. <https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler>. (Accessed on 06/01/2020).
  - [31] Google. [n.d.]. Network Bandwidth. <https://cloud.google.com/compute/docs/network-bandwidth>.
  - [32] Google. [n.d.]. Storage options. <https://cloud.google.com/compute/docs/disks>.
  - [33] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. *SIGMOD Rec.* 23, 2 (may 1994), 243–252. <https://doi.org/10.1145/191843.191886>
  - [34] HashiCorp. [n.d.]. HashiCorp Raft. <https://github.com/hashicorp/raft>. Accessed: 2022-03-31.
  - [35] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. <https://doi.org/10.1145/78969.78972>
  - [36] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-Based HTAP Database. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3072–3084. <https://doi.org/10.14778/3415478.3415535>
  - [37] Clockwork Inc. [n.d.]. Clockwork Accurate Common Time Service. <https://www.clockwork.io/>.
  - [38] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable Message Latency in the Cloud. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 435–448. <https://doi.org/10.1145/2829988.2787479>
  - [39] Theo Jepsen, Stephen Ibanez, Gregory Valiant, and Nick McKeown. 2022. From Sand to Flour: The Next Leap in Granular Computing with NanoSort. *arXiv preprint arXiv:2204.12615* (2022).
  - [40] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. 2019. Derecho: Fast State Machine Replication for Cloud Services. *ACM Trans. Comput. Syst.* 36, 2, Article 4 (apr 2019), 49 pages. <https://doi.org/10.1145/3302258>
  - [41] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*.
  - [42] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for microsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. <https://www.usenix.org/conference/nsdi19/presentation/kaffes>
  - [43] Marios Kogias and Edouard Bugnion. 2020. Hovercraft: Achieving Scalability and Fault-Tolerance for Microsecond-Scale Datacenter Services. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) (EuroSys '20). Association for Computing Machinery, New York, NY, USA, Article 25, 17 pages. <https://doi.org/10.1145/3342195.3387545>
  - [44] Jan Kończak, Paweł T. Wojciechowski, Nuno Santos, Tomasz Żurkowski, and André Schiper. 2021. Recovery Algorithms for Paxos-Based State Machine Replication. *IEEE Transactions on Dependable and Secure Computing* 18, 2 (2021), 623–640. <https://doi.org/10.1109/TDSC.2019.2926723>
  - [45] L. Lamport. 1997. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Comput.* 46, 7 (1997), 779–782. <https://doi.org/10.1109/12.59898>
  - [46] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. <https://doi.org/10.1145/279227.279229>
  - [47] Leslie Lamport. 2005. *Generalized Consensus and Paxos*. Technical Report MSR-TR-2005-33. 60 pages. <https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/>
  - [48] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19 (October 2006), 79–103. <https://www.microsoft.com/en-us/research/publication/fast-paxos/>
  - [49] Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.
  - [50] Leslie Lamport and P. M. Melliar-Smith. 1985. Synchronizing Clocks in the Presence of Faults. *J. ACM* 32, 1 (jan 1985), 52–78. <https://doi.org/10.1145/2455.2457>
  - [51] Collin Lee and John Ousterhout. 2019. Granular Computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) (HotOS '19). Association for Computing Machinery, New York, NY, USA, 149–154. <https://doi.org/10.1145/3317550.3321447>
  - [52] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. [n.d.]. NOPaxos Code Repository. <https://github.com/UWSysLab/NOPaxos>.
  - [53] 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 *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA.
  - [54] Yilong Li, Seo Jin Park, and John Ousterhout. 2021. MilliSort and MilliQuery: Large-Scale Data-Intensive Computing in Milliseconds. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 593–611. <https://www.usenix.org/conference/nsdi21/presentation/li-yilong>
  - [55] Barbara Liskov. 1991. Practical Uses of Synchronized Clocks in Distributed Systems. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing* (Montreal, Quebec, Canada) (PODC '91). Association for Computing Machinery, New York, NY, USA, 1–9. <https://doi.org/10.1145/112600.112601>
  - [56] Barbara Liskov and James Cowling. 2012. Viewstamped replication revisited. (2012).
  - [57] Jennifer Lundelius and Nancy Lynch. 1984. A New Fault-Tolerant Algorithm for Clock Synchronization. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, British Columbia, Canada) (PODC '84). Association for Computing Machinery, New York, NY, USA. <https://doi.org/10.1145/800222.806738>
  - [58] Jennifer Lundelius and Nancy Lynch. 1984. An upper and lower bound for clock synchronization. *Information and Control* 62, 2 (1984), 190–204. [https://doi.org/10.1016/S0019-9958\(84\)80033-9](https://doi.org/10.1016/S0019-9958(84)80033-9)
  - [59] Jennifer Lundelius and Nancy Lynch. 1984. An upper and lower bound for clock synchronization. *Information and Control* 62, 2 (1984), 190–204. [https://doi.org/10.1016/S0019-9958\(84\)80033-9](https://doi.org/10.1016/S0019-9958(84)80033-9)

- [60] Yanhua Mao, Flavio Junqueira, and Keith Marzullo. 2008. *Mencius: Building Efficient Replicated State Machines for WANs*. Technical Report. University of California San Diego. <https://doi.org/CS2008-0930>
- [61] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) (OSDI'08). USENIX Association, USA, 369–384.
- [62] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. 2011. High performance state-machine replication. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. 454–465. <https://doi.org/10.1109/DSN.2011.5958258>
- [63] Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres. 2017. Recovering Shared Objects Without Stable Storage. In *31st International Symposium on Distributed Computing (DISC 2017)*, Vol. 91. 36:1–36:16. <https://doi.org/10.4230/LIPIcs.DISC.2017.36>
- [64] Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres. 2017. *Recovering Shared Objects Without Stable Storage [Extended Version]*. Technical Report. University of Washington. <https://doi.org/UW-CSE-17-08-01>
- [65] Whittaker Michael, Ailijiang Ailidani, Charapko Aleksey, Demirbas Murat, Giridharan Neil, Hellerstein Joseph, Howard Heidi, Stoica Ion, and Szekeres. Adriana. 2021. Scaling Replicated State Machines with Compartmentalization. *Proc. VLDB Endow.* (2021), 12.
- [66] Microsoft. [n.d.]. Global data distribution with Azure Cosmos DB-under the hood. <https://docs.microsoft.com/en-us/azure/cosmos-db/global-dist-under-the-hood>.
- [67] Microsoft. [n.d.]. Network Isolation Options for Machines in Windows Azure Virtual Networks. <https://azure.microsoft.com/en-us/blog/network-isolation-options-for-machines-in-windows-azure-virtual-networks/>.
- [68] Microsoft. [n.d.]. Use the cluster autoscaler in Azure Kubernetes Service (AKS) - Azure Kubernetes Service | Microsoft Docs. <https://docs.microsoft.com/en-us/azure/aks/cluster-autoscaler>. (Accessed on 06/01/2020).
- [69] Jeffrey C. Mogul and Ramana Rao Kompella. 2015. Inferring the Network Latency Requirements of Cloud Tenants. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland. <https://www.usenix.org/conference/hotos15/workshop-program/presentation/mogul>
- [70] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2013. *A proof of correctness for Egalitarian Paxos*. Technical Report. <https://www.cs.cmu.edu/~imoraru/epaxos/tr.pdf>
- [71] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farmington, Pennsylvania) (SOSP '13). Association for Computing Machinery, New York, NY, USA, 358–372. <https://doi.org/10.1145/2517349.2517350>
- [72] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 517–532. <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/mu>
- [73] Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ontario, Canada) (PODC '88). Association for Computing Machinery, New York, NY, USA, 8–17. <https://doi.org/10.1145/62546.62549>
- [74] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319. <https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro>
- [75] OpenFabrics Alliance. [n.d.]. libfabric Programmer Manual. [https://ofiwg.github.io/libfabric/v1.11.1/man/fi\\_tcp.7.html](https://ofiwg.github.io/libfabric/v1.11.1/man/fi_tcp.7.html)
- [76] Seo Jin Park and John Ousterhout. 2019. Exploiting Commutativity for Practical Fast Replication (NSDI'19). USENIX Association, USA, 47–64.
- [77] PingCap. [n.d.]. TiKV-Data Sharding. <https://tikv.org/deep-dive/scalability/data-sharding>
- [78] 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 *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) (NSDI'15). USENIX Association, USA, 43–57.
- [79] Redis Enterprise. [n.d.]. Redis. <https://redis.io>.
- [80] Krzysztof Rzadca, Paweł Findeisen, Jacek Swiderski, Przemysław Zych, Przemysław Broniek, Jarek Kusmerek, Paweł Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. 2020. Autopilot: workload autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.
- [81] Jha Sagar, Rosa Lorenzo, and Ken Birman. 2021. Spindle: Techniques for Optimizing Atomic Multicast on RDMA. *arXiv:2110.00886v1* (2021).
- [82] Tollman Sarah. [n.d.]. TOQ-based EPaxos Repository. <https://github.com/PlatformLab/epaxos>.
- [83] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieder, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) (SIGMOD '20). Association for Computing Machinery, New York, NY, USA, 1493–1509. <https://doi.org/10.1145/3318464.3386134>
- [84] Sarah Tollman, Seo Jin Park, and John Ousterhout. 2021. EPaxos Revisited. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 613–632. <https://www.usenix.org/conference/nsdi21/presentation/tollman>
- [85] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. 2017. APUS: Fast and Scalable Paxos on RDMA. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) (SoCC '17). Association for Computing Machinery, New York, NY, USA, 94–107. <https://doi.org/10.1145/3127479.3128609>
- [86] Feiran Wang. 2019. *Building High-performance Distributed Systems with Synchronized Clocks*. Ph.D. Dissertation. Stanford University. <https://www.proquest.com/dissertations-theses/building-high-performance-distributed-systems/docview/2467863602/se-2?accountid=14026>
- [87] Zhaoguo Wang, Changgeng Zhao, Shuai Mu, Haibo Chen, and Jinyang Li. 2019. On the Parallels between Paxos and Raft, and How to Port Optimizations. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. Association for Computing Machinery, New York, NY, USA, 445–454. <https://doi.org/10.1145/3293611.3331595>
- [88] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *SIGOPS Oper. Syst. Rev.* 35, 5 (oct 2001), 230–243. <https://doi.org/10.1145/502059.502057>
- [89] Michael Whittaker, Neil Giridharan, Adriana Szekeres, Joseph M Hellerstein, Heidi Howard, Faisal Nawab, and Ion Stoica. 2020. Matchmaker paxos: A reconfigurable consensus protocol [technical report]. *arXiv preprint arXiv:2007.09468* (2020).
- [90] Michael Whittaker, Neil Giridharan, Adriana Szekeres, Joseph M Hellerstein, and Ion Stoica. 2020. Bipartisan paxos: A modular state machine replication protocol. *arXiv preprint arXiv:2003.00331* (2020).
- [91] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. 2013. Bobtail: Avoiding Long Tails in the Cloud. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 329–341. [https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/xu\\_yunjing](https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/xu_yunjing)
- [92] Yahoo! [n.d.]. YCSB Workload. <https://github.com/brianfrankcooper/YCSB/tree/master/workloads>.
- [93] Xinan Yan. [n.d.]. Domino Repository. <https://github.com/xnyan/domino>.
- [94] Xinan Yan, Linguan Yang, and Bernard Wong. 2020. Domino: Using Network Measurements to Reduce State Machine Replication Latency in WANs. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies* (Barcelona, Spain) (CoNEXT '20). Association for Computing Machinery, New York, NY, USA, 351–363. <https://doi.org/10.1145/3386367.3431291>
- [95] ZeroMQ community. [n.d.]. ZeroMQ. <https://zeromq.org/>. Accessed: 2021-02-02.
- [96] Siyuan Zhou and Shuai Mu. 2021. Fault-Tolerant Replication with Pull-Based Consensus in MongoDB. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 687–703. <https://www.usenix.org/conference/nsdi21/presentation/zhou>

## APPENDICES

In this appendix, we include the following:

- The explanation of Nezha’s recovery (§A).
- The correctness proof of Nezha (§B).
- The evaluation of Nezha under different workloads (§C).
- The detailed discussion and evaluation about the effect of clock variance on Nezha’s performance (§D).
- The micro benchmark to evaluate packet reordering in public cloud (§E).
- The deployment experience and evaluation of Derecho in bare-metal servers and public cloud (§F).
- The analysis on the incorrectness of Domino due to clock skew/failure (§G).
- The analysis on the incorrectness of NOPaxos recovery due to stray messages (§H).
- The analysis on the incorrectness of Raft without stable storage (§I).
- Formal comparison among DOM, MOM, and OUM (§J).
- The discussion about deploying Nezha in WAN and its expected advantages over EPaxos in WAN (§K).
- The TLA+ specification of Nezha (§L).

### A RECOVERY PROTOCOL AND ALGORITHMS

We explain how Nezha leverages the diskless crash recovery algorithm [56] from Viewstamped Replication in 3 steps. First, we explain how we adopt the recent concept of crash-vectors [63, 64] to fix the incorrectness in the crash recovery algorithm. Second, we explain how a replica rejoins Nezha following a crash. Third, we describe how the leader election works if the leader crashes.

#### A.1 Crash Vector

Like Viewstamped Replication, Speculative Paxos and NOPaxos, Nezha also adopts diskless recovery to improve performance. However, in contrast to them, Nezha avoids the effect of *stray messages* [44] (i.e., messages that are sent out but not delivered before replica crash, so that the relaunched replicas forget them) using the *crash-vector* [63, 64]. *crash-vector* is a vector containing  $2f + 1$  integer counters corresponding to the  $2f + 1$  replicas. Each replica maintains such a vector, with all counters initialized as 0s.

*crash-vectors* can be aggregated by taking the max operation element-wise to produce a new *crash-vector*. During the replica rejoin (§A.2) and leader change (§A.3) process, replicas send their *crash-vectors* to each other. Receivers can make their *crash-vectors* more up-to-date by aggregating their *crash-vector* with the *crash-vector* from the sender. Meanwhile, by comparing its local *crash-vector* and the sender’s *crash-vector*, the receiver can recognize whether or not the sender’s message is a potential *stray message* (refer to [64] for detailed description of *crash-vector*).

Nezha uses *crash-vectors* to avoid two types of *stray messages*, i.e. the *stray messages* during recovery and the *stray messages* (*fast-replies*) during quorum check.

(1) There can be *stray messages* during the recovery process. The stray messages cause replicas to elect a leader, whose state falls behind the others, and finally causes permanent loss of committed requests. Such error cases have been analyzed in [64] and we also

illustrate it in §H.1 by taking NOPaxos as the example. The *crash-vector* prevents the *stray messages* effect because it enables the replicas to recognize potential *stray messages* by comparing a *crash-vector* received from a replica with the local *crash-vector*. During recovery, the RECOVERING replica first recovers its *crash-vector* by collecting and aggregating the *crash-vectors* from a majority of NORMAL replicas. Then, the replica increments its own counter (i.e. replica  $i$  increments the  $i$ th counter in the vector) and tags the new *crash-vector* to the messages sent afterwards. Once the update of *crash-vector* is exposed to the other replicas, they can recognize the *stray messages* sent by the replica before crash (i.e., those messages have a smaller value at the  $i$ th counter), and avoid processing those messages. Thus, the recovery will not be affected by *stray messages*.

(2) Stray messages can also occur during the quorum check in the fast path: some replicas send *fast-replies* and crash after that. These *fast-replies* may become *stray messages* and participate into the quorum check, which makes the proxies/clients prematurely believe the request has been persisted to a super-majority of replicas, but actually not yet (i.e. the recovered replicas may not hold the requests after their recovery). Such error cases are illustrated in §H.2. The *crash-vector* prevents the effect of such *stray fast-replies*, because we include the information of *crash-vectors* in the *fast-replies* (§5.2). When a failed replica rejoins the system (Algorithm 1), it leads to the update of *crash-vectors* for the leader and other remaining followers, so these replicas will send *fast-replies* with different *hashes* from the *stray fast-replies* sent by the rejoined replica. Therefore, the stray *fast-replies* from the rejoined replica and the normal *fast-replies* from the other replicas cannot form the super-quorum together (refer to §H.2 for more details).

#### A.2 Replica Rejoin

Crashed replicas can rejoin the system as followers. After the replica crashes and is relaunched, it sets its *status* as RECOVERING. Before it can resume request processing, the replica needs to recover its replica state, including *crash-vector*, *view-id*, *log* and *sync-point*. With reference to Algorithm 1, we explain how the replica rejoin process works.

**Step 1:** The replica sets its *status* as RECOVERING (line 2), and broadcasts the same CRASH-VECTOR-REQ to all replicas to request their *crash-vectors*. A *nonce* (line 4) is included in the message, which is a random string locally unique on this replica, i.e., this replica has never used this *nonce* <sup>6</sup>.

**Step 2:** After receiving the CRASH-VECTOR-REQ, replicas with NORMAL status reply to the recovering replica with <CRASH-VECTOR-REP, *nonce*, *crash-vector*> (line 40-47).

**Step 3:** The recovering replica waits until it receives the corresponding replies (containing the same *nonce*) from a majority ( $f + 1$ ) of replicas (line 23). Then it aggregates the  $f + 1$  *crash-vectors* by taking the maximum in each dimension (line 7, line 99-104). After obtaining the aggregated crash vector  $cv$ , the replica increment its own dimension, i.e.  $cv[replica-id] = cv[replica-id] + 1$  (line 8).

**Step 4:** The recovering replica broadcasts a recovery request to all replicas, which includes its *crash-vector*, i.e. <RECOVERY-REQ,  $cv$ > (line 11, line 26-30).

<sup>6</sup>There are many options available to generate the locally unique *nonce* string [56, 64]. Nezha uses the universally unique identifier (UUID) (GENERATE-UUID in line 4), which have been widely supported by modern software systems.

---

**Algorithm 1** Replica rejoin
 

---

```

Local State:                                     ▶
nonce,                                             ▶ A locally unique string on this replica
C,                                                 ▶ Reply set of CRASH-VECTOR-REP
R,                                                 ▶ Reply set of RECOVERY-REP
r,                                                 ▶ short for replica-id (the message sender)
cv,                                               ▶ short for crash-vector
status, view-id, last-normal-view, log, sync-point ▶ Other attributes

1: upon RECOVER do
2:   status = RECOVERING
3:   C = ∅
4:   nonce = GENERATE-UUID
5:   READ-CRASH-VECTOR
6:   cv-set = {m.cv | m ∈ C}
7:   cv = AGGREGATE(cv-set ∪ {cv})
8:   cv[r] = cv[r] + 1                                ▶ Increment its own counter
9:   do
10:    R = ∅
11:    READ-RECOVERY-INFO
12:    highest-view = max{m.v | m ∈ R}
13:    leader-id = highest-view % (2f + 1)
14:    while (leader-id = r)
15:      Pick m ∈ R: m.v = highest-view
16:      STATE-TRANSFER(leader-id)
17: function READ-CRASH-VECTOR
18:   m.type = CRASH-VECTOR-REQ
19:   m.r = r
20:   m.nonce = nonce
   ▶ Broadcast CRASH-VECTOR-REQ to all replicas
21:   for i ← 0 to 2f do
22:     SEND-MESSAGE(m, i)                ▶ Send message m to the replica i
23:   Wait until |C| ≥ f + 1              ▶ C is initialized as ∅ by the caller
24:   return
25: function READ-RECOVERY-INFO
26:   m.type = RECOVERY-REQ
27:   m.r = r
28:   m.cv = cv
   ▶ Broadcast RECOVERY-REQ to all replicas
29:   for i ← 0 to 2f do
30:     SEND-MESSAGE(m, i)
31:   Wait until |R| ≥ f + 1              ▶ R is initialized as ∅ by the caller
32:   return
33: function STATE-TRANSFER(i)
34:   m.type = STATE-TRANSFER-REQ
35:   m.r = r
36:   m.cv = cv
37:   SEND-MESSAGE(m, i)
38:   Wait until status = NORMAL
39:   return
40: upon RECEIVING CRASH-VECTOR-REQ, m do
41:   if status ≠ NORMAL then
42:     return
43:   m'.type = CRASH-VECTOR-REP
44:   m'.r = r
45:   m'.nonce = m.nonce
46:   m'.cv = cv
47:   SEND-MESSAGE(m', m.r)
48: upon RECEIVING CRASH-VECTOR-REP, m do
49:   if status ≠ RECOVERING then
50:     return
51:   if nonce ≠ m.nonce then
52:     return
53:   C = C ∪ {m}
54: upon RECEIVING RECOVERY-REQ, m do
55:   if status ≠ NORMAL then
56:     return
57:   cv = AGGREGATE(cv, m.cv)
58:   m'.type = RECOVERY-REP
59:   m'.r = r
60:   m'.v = view-id
61:   m'.cv = cv
62:   SEND-MESSAGE(m', m.r)
63: upon RECEIVING RECOVERY-REP, m do
64:   if status ≠ RECOVERING then
65:     return
66:   if CHECK-CRASH-VECTOR(m, cv) = false then
67:     Resend RECOVERY-REQ to m.r
68:   else                                ▶ Remove stray messages and add the fresh one
69:     R' = {m' ∈ R | m'.cv[m'.r] < cv[m'.r]}
70:     R = R ∪ {m} - R'
71:     ∀m' ∈ R', resend RECOVERY-REQ to m'.r
72: upon RECEIVING STATE-TRANSFER-REQ, m do
73:   if status ≠ NORMAL then
74:     return
75:   if CHECK-CRASH-VECTOR(m, cv) = false then
76:     return
77:   m'.type = STATE-TRANSFER-REP
78:   m'.log = log
79:   m'.v = view-id
80:   m'.sp = sync-point
81:   m'.cv = cv
82:   SEND-MESSAGE(m', m.r)
83: upon RECEIVING STATE-TRANSFER-REP, m do
84:   if status ≠ RECOVERING then
85:     return
86:   if CHECK-CRASH-VECTOR(m, cv) = false then
87:     return
88:   log = m.log
89:   last-normal-view = view-id = m.v
90:   log = m.log
91:   sync-point = m.sp
92:   status = NORMAL                                ▶ Rejoin as a NORMAL follower
93: function CHECK-CRASH-VECTOR(m, cv)
94:   if m.cv[m.r] < cv[m.r] then            ▶ A potential stray message
95:     return false
96:   else
97:     cv = AGGREGATE({cv, m.cv})            ▶ Update local cv
98:     return true
99: function AGGREGATE(cv-set)
100:  ret = [0 ... 0]
        2f+1
101:  for c ∈ cv-set do
102:    for i ← 0 to 2f do
103:      ret[i] = max(ret[i], c[i])
104:  return ret

```

---



**Step 5:** After receiving the RECOVERY-REQ, replicas with NORMAL status update their own *crash-vectors* by aggregating with *cv*, obtained from the request in step 4. Then, these replicas send back a reply including their own *view-id* and *crash-vector*, i.e. <RECOVERY-REP, *view-id*, *crash-vector*> (line 54-63).

**Step 6:** The recovering replica waits until it receives the recovery replies from  $f + 1$  replicas (line 31). If the RECOVERY-REP is not a *stray message*, it updates its own *crash-vector* by aggregating it with the *crash-vectors* included in these replies (line 66); otherwise, it resends RECOVERY-REQ to that replica, asking for a fresh message (line 67). Because the *crash-vectors* may have been updated (line 66), those RECOVERY-REP which have been received can also become *stray messages* because their *crash-vectors* are no longer fresh enough. Therefore, we also remove them ( $R'$  in line 69) from the reply set  $R$  (line 70), and resend requests to the related replicas for fresher replies (line 71).

**Step 7:** The RECOVERING replica picks the highest *view-id* among the  $f + 1$  replies (line 12). From the highest *view-id*, it knows the corresponding leader of this view (line 13). If the RECOVERING replica happens to be the leader of this view, it keeps broadcasting the recovery request (line 9-14), until the majority elects a new leader among themselves. Otherwise, the RECOVERING replica fetches the *log*, *sync-point*, *view-id* from the leader via a state transfer (line 16, line 33-39). After that, the replica set its *status* to NORMAL and can continue to process the incoming requests.

Specially, the RECOVERING replica(s) do not participate in the view change process (§A.3). When the majority of replicas are conducting a view change (possibly due to leader failure), the RECOVERING replica(s) just wait until the majority completes the view change and elects the new leader.

### A.3 Leader Change

When the follower(s) suspect that the leader has failed, they stop processing new client requests. Instead, they perform the view change protocol to elect a new leader and resume request processing. With reference to Algorithm 2, we explain the details of the view change process.

**Step 1:** When a replica fails to receive the heartbeat (i.e., *sync* message) from the leader for a threshold of time, it suspects the leader has failed. Then, it sets its *status* as VIEWCHANGE, increments its *view-id*, and broadcasts a view change request to all replicas including its *crash-vector*, i.e. <VIEW-CHANGE-REQ, *view-id*, *replica-id*, *cv*> (line 6-10)<sup>7</sup>. The replica switches its *status* from NORMAL to VIEWCHANGE, and enters the view change process.

**Step 2:** After receiving a VIEW-CHANGE-REQ message, the recipient checks the *cv* and *replica-id* with its own *crash-vector* (line 32). If this message is a potential *stray message*, then the recipient ignores it. Otherwise, the recipient updates its *crash-vector* by aggregation. After that, the recipient also participates in the view change (line 35) if its *view-id* is lower than that included in the VIEW-CHANGE-REQ message.

**Step 3:** All replicas under view change send a message <VIEW-CHANGE, *view-id*, *log*, *sync-point*, *last-normal-view*> to the leader

<sup>7</sup>The view change request will be rebroadcast if the replica times out but is still waiting for the view change process to complete. The same is also true for the view change message described in the next step.

of the new view ( $replica-id = view-id \% (2f + 1)$ ) (line 11). Here *last-normal-view* indicates the last view in which the replica's *status* was NORMAL.

**Step 4:** After the new leader receives the VIEW-CHANGE messages from  $f$  followers with matching *view-ids*, it can recover the system state by merging the *logs* from the  $f+1$  replicas including itself (line 67). The new leader only merges the *logs* with the highest *last-normal-view*, because a smaller *last-normal-view* indicates the replica has lagged behind for several view changes, thus its *sync-point* cannot be larger than the other replicas with higher *last-normal-view* values. Therefore, it makes no contribution to the recovery and does not need to join.

**Step 5:** The new leader initializes an empty log list (denoted as *new-log*) (line 74). Among the VIEW-CHANGE messages with the highest *last-normal-view*, it picks the one with the largest *sync-point* (line 75-77). Then it directly copies the log entries from that message up to the *sync-point* (line 78-82).

**Step 6:** Afterwards, the new leader checks the remaining entries with larger *deadlines* than *sync-point* (line 83-88). If the same entry (2 entries are the same iff they have the same <*deadline*, *client-id*, *request-id*>) exists in at least  $\lceil f/2 \rceil + 1$  out of the  $f + 1$  logs, then leader appends the entry to *new-log*.

**Step 7:** After *new-log* is built, the new leader broadcasts <START-VIEW, *cv*, *view-id*, *new-log*> to all replicas (line 68-70).

**Step 8:** After receiving the START-VIEW message with a *view-id* greater than or equal to its *view-id*, the replica updates its *view-id* and *last-normal-view* (line 97), and replaces its *log* with *new-log* (line 98). Besides, it updates *sync-point* as the last entry in the new *log* (line 98), because all the entries are consistent with the leader. Finally, replicas set their *statuses* to NORMAL (line 100), and the system state is fully recovered.

**Step 9:** After the system is fully recovered, the replicas can continue to process the incoming requests. Recall in §5.3 that the incoming request is allowed to enter the *early-buffer* if its deadline is larger than *the last released request* which is not commutative. To ensure uniform ordering, the eligibility check is still required for the incoming request even if it is the first one arriving at the replica after recovery. The replica considers the entries (requests) in the recovered *log*, which are not commutative to the incoming request, and chooses the one as *the last released request* with the largest deadline among them. The incoming request can enter the *early-buffer* if its deadline than *the last released request*, otherwise, it is put into the *late-buffer*.

Note that the view change protocol chooses the leader in a round-robin way ( $view-id \% (2f + 1)$ ). Specially, a view change process may not succeed because the new leader also fails (as mentioned in [56]). In this case (i.e. after followers have spent a threshold of time without completing the view change), followers will continue to increment their *view-ids* to initiate a further view change, with yet another leader.

After the replica rejoin or leader change process, replicas' *crash-vectors* will be updated. Due to packet drop, some replicas may fail to receive the update of *crash-vectors* during the recovery, thus they cannot contribute to the quorum check of the fast path in the following request processing, because their *crash-vectors* are still old and cannot generate the consistent hash with the leader's hash. To enable every replica to obtain the fresh information of *crash-vectors* rapidly, the leader can piggyback the fresh *crash-vectors*

---

**Algorithm 2** Leader change
 

---

```

Local State:
V,                                ▶ Reply set of VIEW-CHANGE
r,                                ▶ short for replica-id (the message sender)
cv,                                ▶ short for crash-vector
last-normal-view,                 ▶ The most recent view in which
                                  ▶ the replica's status is NORMAL
status, view-id, log, sync-point  ▶ Other attributes

1: upon SUSPECT LEADER FAILURE do
2:   do
3:     INITIATE-VIEW-CHANGE(view-id + 1)
4:   while (status ≠ NORMAL)
5:   function INITIATE-VIEW-CHANGE(v)
6:     status = VIEWCHANGE
7:     view-id = v
8:     V = ∅
9:     ▶ Broadcast VIEW-CHANGE-REQ to all replicas
10:    for i ← 0 to 2f do
11:      SEND-VIEW-CHANGE-REQ(i)
12:      ▶ Send VIEW-CHANGE to the new leader
13:    SEND-VIEW-CHANGE(v%(2f + 1))
14:    Wait until status = NORMAL or TIMEOUT
15:    return
16:  function SEND-VIEW-CHANGE-REQ(i)
17:    m.type = VIEW-CHANGE-REQ
18:    m.v = view-id
19:    m.cv = cv
20:    SEND-MESSAGE(m, i)
21:    return
22:  function SEND-VIEW-CHANGE(i)
23:    m.type = VIEW-CHANGE
24:    m.v = view-id
25:    m.cv = cv
26:    m.log = log
27:    m.sp = sync-point
28:    m.lnv = last-normal-view
29:    SEND-MESSAGE(m, i)
30:    return
31: upon RECEIVING VIEW-CHANGE-REQ, m do
32:   if status = RECOVERING then
33:     return
34:   if CHECK-CRASH-VECTOR(m, cv)=false then
35:     return
36:   if m.v > view-id then
37:     INITIATE-VIEW-CHANGE(m.v)
38:   else
39:     if status = NORMAL then
40:       SEND-START-VIEW(m.r)
41:     else ▶ The leader is asking for fresher VIEW-CHANGE
42:       SEND-VIEW-CHANGE(m.r)
43:   function SEND-START-VIEW(i)
44:     m.type = START-VIEW
45:     m.v = view-id
46:     m.cv = cv
47:     m.log = log
48:     SEND-MESSAGE(m, i)
49:     return
50:   upon RECEIVING VIEW-CHANGE, m do
51:     if status = RECOVERING then
52:       return
53:     if CHECK-CRASH-VECTOR(m, cv)=false then
54:       return
55:     if status = NORMAL then
56:       if m.v > view-id then
57:         INITIATE-VIEW-CHANGE(m.v)
58:       else ▶ The sender lags behind
59:         SEND-START-VIEW(m.r)
60:     else if status = VIEWCHANGE then
61:       if m.v > view-id then
62:         INITIATE-VIEW-CHANGE(m.v)
63:       else if m.v < view-id then ▶ The sender lags behind
64:         SEND-VIEW-CHANGE-REQ(m.r)
65:       else ▶ Remove stray messages and add the fresh one
66:         V' = {m' ∈ V | m'.cv[m'.r] < cv[m'.r]}
67:         V = V ∪ {m} - V'
68:         ∀m' ∈ V', resend VIEW-CHANGE-REQ to m'.r
69:         if |V| ≥ f + 1 then
70:           log = MERGE-LOG(V)
71:           for i ← 0 to 2f do
72:             SEND-START-VIEW(i)
73:           last-normal-view = view-id
74:           status = NORMAL ▶ Leader becomes NORMAL
75:         function MERGE-LOG(V)
76:           new-log = ∅
77:           largest-normal-view = max{m.lnv | m ∈ V}
78:           largest-sync-point = max{m.sp | m ∈ V
79:             and m.lnv = largest-normal-view}
80:           Pick m ∈ V:
81:             m.lnv = largest-normal-view and
82:             m.sp = largest-sync-point
83:           ▶ Directly copy entries up to sync-point
84:           for e ∈ m.log do ▶ m.log is already sorted by deadlines
85:             if e.deadline ≤ largest-sync-point.deadline then
86:               new-log.append(e)
87:             else
88:               break
89:           ▶ Add other committed entries beyond sync-point
90:           entries = {e | e ∈ m.log
91:             and e.deadline > largest-sync-point.deadline
92:             and m.lnv = largest-normal-view}
93:           for e ∈ entries do
94:             ▶ Check how many replicas contain e
95:             S = {m | m ∈ V and e ∈ m.log}
96:             if |S| ≥ ⌈f/2⌉ + 1 then
97:               log.append(e)
98:           Sort new-log by entries' deadlines
99:           return new-log
100:   upon RECEIVING START-VIEW, m do
101:     if status = RECOVERING then
102:       return
103:     if CHECK-CRASH-VECTOR(m, cv)=false then
104:       return
105:     if m.v < view-id then
106:       return
107:     last-normal-view = view-id = m.v
108:     log = m.log
109:     sync-point = log.last()
110:     status = NORMAL ▶ Followers become NORMAL

```

in the *sync* messages, so that replicas can check and update their *crash-vectors* as soon as possible.

#### A.4 Reconfiguration

Just like Speculative Paxos and NOPaxos, Nezha can also use the standard reconfiguration protocol from Viewstamped Replication [56] (with its incorrectness fixed by *crash-vector* [63, 64]) to change the membership of the replica group, such as replacing the failed replicas with the new ones that have a new disk, increasing/decreasing the number of replicas in the system, etc. However, Nezha is free from reconfiguring the network, whereas Speculative Paxos and NOPaxos require to modify their network (e.g., updating the forwarding rules of the Openflow controller, initializing a new session number at the sequencer, etc.) for every reconfiguration, which adds non-trivial complexity in a real deployment.

### B CORRECTNESS PROOF OF NEZHA

With the normal behavior described in §5.2~§5.5, we can prove that the recovery protocol of Nezha guarantees the following correctness properties.

- **Durability:** if a client considers a request as committed, the request survives replica crashes.
- **Consistency:** if a client considers a request as committed, the execution result of this request remains unchanged after the replica's crash and recovery.
- **Linearizability:** A request appears to be executed exactly once between start and completion. The definition of linearizability can also be reworded as: if the execution of a request is observed by the issuing client or other clients, no contrary observation can occur afterwards (i.e., it should not appear to revert or be reordered).

#### B.1 Proof of Durability

The client/proxy considers *req* as committed after receiving the corresponding quorum or superquorum of replies. Since the quorum checks on both the fast path and slow path require the leader's reply, a committed request indicates that the request must have been accepted by the leader. If a follower crashes, it does not affect durability because the recovered followers directly copy *log* from the leader via state transfer (Step 7 in §A.2) before serving new requests. Hence, we consider the durability property during leader crashes.

(1) If the client/proxy commits *req* in the fast path, it means the request has been replicated to the leader and at least  $f + \lceil f/2 \rceil$  followers. When the leader crashes, among any group of  $f + 1$  replicas, *req* exists in at least  $\lceil f/2 \rceil + 1$  of them because of quorum intersection. Hence, *req* will be added to the *new-log* in Step 6 in §A.3, and eventually recovered.

(2) If the client/proxy commits *req* in the slow path, it means *req* has been synced with the leader by at least  $f + 1$  replicas, i.e., there are at least  $f + 1$  replicas containing a *sync-point* whose *deadline* is greater than or equal to *req*'s *deadline*. Due to quorum intersection, there will at least one replica which has the *sync-point* in Step 4 of §A.3. Therefore, *req* will be directly copied to *new-log* in Step 5 of §A.3, and eventually recovered.

#### B.2 Proof of Consistency

Without considering the acceleration of recovery mentioned in §6, we prove consistency. It is also easy to check that the recovery acceleration is a performance optimization that does not affect the consistency property. So, ignoring acceleration of recovery for simplicity, followers do not execute requests. Thus, we only need to consider the leader's crash and recovery. We assume the client/proxy has committed *req* before the leader crash.

(1) If the client/proxy commits *req* in the fast path, it means at least  $f + \lceil f/2 \rceil$  followers have consistent log entries with the leader up to this request *req*. Therefore, on the old leader, all the log entries *before req* are *also* committed, because they also form a super quorum with consistent hashes. So, they can survive crashes and be recovered in Steps 5 and 6 of §A.3. Additionally, consider an uncommitted request *ureq*, which is not commutative to *req* and has a smaller *deadline* than *req*, it cannot be appended by any of the  $f + \lceil f/2 \rceil + 1$  replicas which have appended *req*, because the *early-buffer* of DOM only accepts and releases requests in the ascending order of deadlines (§4). Even if all the other  $\lfloor f/2 \rfloor$  have appended *ureq*, they fail to satisfy the condition in Step 5/Step 6, so *ureq* cannot appear in the recovered *logs* to affect the execution result of *req*.

(2) If the client/proxy commits *req* in the slow path, it means at least  $f$  followers have consistent log entries with the leader up to *req*, i.e., the *deadlines* of their *sync-points* are greater than or equal to the *deadline* of *req*. Therefore, on the old leader, all the log entries *before req* are committed, and they can survive crashes and be recovered in Step 5 of §A.3. Additionally, if the follower's *log* contains the request *ureq*, which is not commutative to *req* and has a smaller *deadline* than *req*, but does not exist on the leader, then *ureq* cannot appear in the recovery *log* of the new leader. This is because, based on the protocol in §5.4, the follower advances its *sync-point* strictly following *sync* messages from the leader. Since the *sync* message does not include the *ureq*'s 3-tuple  $\langle \text{client-id}, \text{request-id}, \text{deadline} \rangle$ , the follower will delete *ureq* before updating its *sync-point*. Therefore, it is impossible for *ureq* to appear the recovered *logs* and affect the execution result of *req*.

After recovery, the survived log entries will be executed by the new leader according to the ascending order of their *deadlines*, thus the same execution order is guaranteed and provides the consistent execution result for *req*.

#### B.3 Proof of Linearizability

We assume there are two committed requests, denoted as *req-1* and *req-2*. The submission of *req-2* is invoked after the completion of *req-1*, i.e. the client has observed the execution of *req-1* before submitting *req-2*. We want to prove that no contrary observation can occur after crash and recovery. Here we assume *req-1* and *req-2* are not commutative with each other, because the execution of commutative requests cause no effect on each other, regardless of their execution order.

Since *req-2* is invoked after the completion of *req-1*, *req-2* must have a larger *deadline* than *req-1*, otherwise, it cannot be appended to the *log*. Based on the durability property, *req-1* and *req-2* will be both recovered after a crash. According to the recovery algorithm,

the new leader still executes the two requests based on their *deadlines*. Therefore, the execution of *req-1* on the new leader cannot observe the effect of *req-2*. By contrast, while executing *req-2*, the effect of *req-1*'s execution has already been reflected in the leader's replica state. Therefore, no contrary observation (i.e., revert or recorder) can occur after the crash and recovery.

## C EVALUATION OF NEZHA UNDER DIFFERENT WORKLOADS

We adopt the similar approach as [84] to conduct extensive evaluation under different workloads: we maintain 1 million unique keys and choose different values of read ratio and skew factors to generate different workloads. As for the read ratio, we choose three different values, i.e. read-10% (write-intensive), read-50% (medium) and read-90% (read-intensive). As for the skew factor, we also choose three different values, i.e. skew-0.0 (evenly distributed), skew-0.5 (medium) and skew-0.99 (highly skewed). The combination of the two dimensions create 9 different workloads. We measure the median latency and throughput under each workload, as shown in Figure 17. Considering the variance in cloud environment, we run each test case for 5 times and plot the average values.

Although the latency in the cloud can vary over time [38, 69, 91] and introduces some noise to performance results, in general, the commutativity optimization remains effective across all workloads and helps reduce the latency by 7.7%-28.9%: under low throughput, the effectiveness of the commutativity optimization is not distinct because the No-Commutativity variant can also keep a high fast commit ratio (~75%). However, as the throughput increases, the fast commit ratio of No-Commutativity variant drops distinctly but the commutativity variant can still maintain a high fast commit ratio (80%-97%), so the commutativity optimization becomes more effective. Then, as the throughput continues to grow, it reaches closer to the capacity of the replicas and even overloads the replicas, so the reduction of latency becomes less distinct again and eventually negligible.

## D EVALUATION UNDER DIFFERENT CLOCK VARIANCE

### D.1 Explanation of Clock Assumption

Nezha depends on clock synchronization for performance but not for correctness. An accurate clock synchronization provides favorable conditions for Nezha to achieve high performance. Here "accurate" means the clocks among replicas (DOM-Rs) and proxies (DOM-Ses) are synchronized with a small error bound *in most cases*, but note that Nezha does not assume a deterministic worst-case error bound, which is impractical because Huygens is built atop a probabilistic model (SVM), and the Huygens agents (or other clock synchronization algorithms) can also fail while the consensus protocol is still running without awareness of that.

Besides, Nezha's correctness does not require the assumption of monotonously increasing clock time either. In other words, even if the local clock time goes back and forth (this can happen because Huygens or other clock synchronization algorithms may correct the clocks with some negative offset), Nezha's correctness is still preserved thanks to the entrance condition of the *early-buffer*. Recall

in §4, the eligibility check to enter the *early-buffer* is to compare the incoming request's deadline with the deadline of the last released one (rather than the replica's local clock time). Requests in the *early-buffer* are organized with a priority queue and released according to their deadline order. The clock skew can only cause requests to be released prematurely, but the released requests all follow the ascending order of deadlines, therefore, the invariant of uniform ordering is preserved by DOM. Establishing protocol correctness independent of clock skew is desirable and we will show in §G that the other protocol, Domino, loses its correctness due to clock skew (i.e. it can lose committed requests permanently if replica clocks go from large values to small values).

### D.2 Quantifying the Effect of Bad Clock Synchronization on Nezha Performance

Although we have not experienced significant clock skew in our evaluation, it is worthwhile to quantify the effect on Nezha performance imposed by different clock synchronization quality. To simplify the discussion below, we consider most VM/server's clocks are synchronized to the reference clock time within a tight bound, whereas the other ones suffer from distinct skew and are not well synchronized with the reference clock time. Thus, we mainly focus on three categories.

- (1) The leader replica's clock is badly synchronized with the other VMs.
- (2) The follower replica's clock is badly synchronized with the other VMs.
- (3) The proxy's clock is badly synchronized with the other VMs.

**Method.** To create the effect of bad clock synchronization, we choose one or multiple target VMs (i.e. the leader replica, or the follower replica, or the proxies) and inject artificial offsets when the clock APIs are called on the VM. To be more specific, we generate random offsets based on normal distribution  $N(\mu, \sigma)$ . For each test case, we choose different mean values ( $\mu$ ) and standard deviation ( $\sigma$ ) for the distribution to mimic bad clock synchronization of different degrees. When the clock API is called, instead of returning the clock value, we take an offset sample from the distribution and add it to the clock value, and then return this summed value, to make its clocks faster/slower than the others.

**Test Setting.** Similar to the setting in §7.2, we set up 3 replicas and 2 proxies, and use 10 open-loop clients to submit at 10K reqs/sec each, thus yielding a throughput ~100K reqs/sec. We measure the latency for each test case and study how the latency evolves as the clock synchronization quality varies. We maintain the same parameters for the adaptive latency bound formula (refer to §4). Specifically, the sliding window size is 1000 to calculate the moving median  $M$ ;  $\beta = 3$ ;  $D = 200\mu s$ . During our tests, we observe the  $\sigma_S$  and  $\sigma_R$  returned by Huygens are both very small, typically  $1 - 2\mu s$ . We choose 10 different normal distributions (as shown in Figure 18) to mimic bad clock synchronization of different degrees, from the slowest clock to the fastest clock.

For example,  $N(-300, 30)$  indicates that the mean value of the normal distribution is  $-300\mu s$  with a standard deviation of  $30\mu s$ . When we choose an offset (typically a large negative value) from this distribution and add it to the clock value, it will make the clock

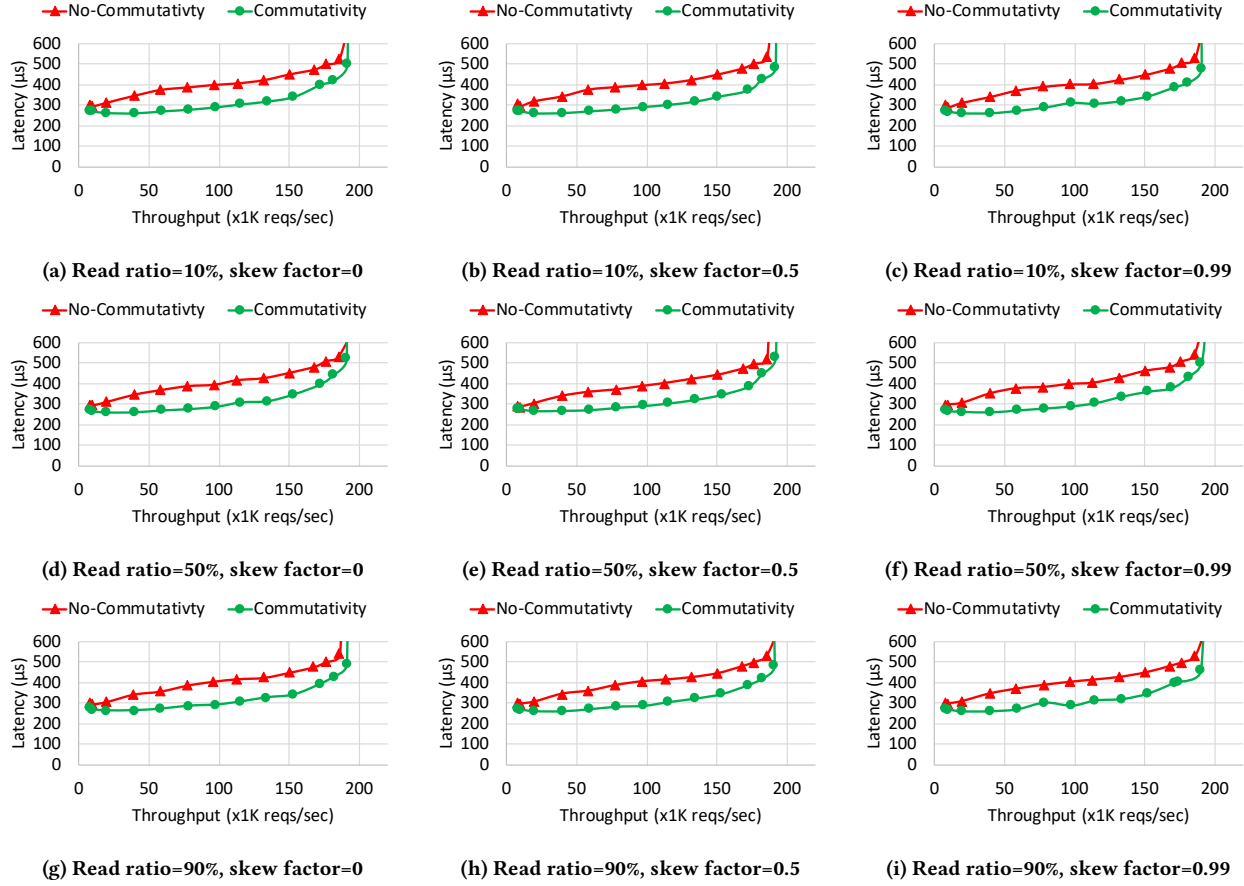


Figure 17: Latency vs. throughput (open-loop)

value smaller than the synchronized clock value by hundreds of microseconds, i.e., the clock becomes slower than the other clocks due to the offset we have added.

**D.2.1 Bad Clock Synchronization of Leader Replica.** As shown in Figure 18a, when leader’s clock fails to be synchronized with the other VMs and goes faster or slower, it will inflate the latency performance of Nezha. Comparing the faster-clock cases and the slower-clock cases, we can see that a slower clock on the leader replica causes more degradation than a faster clock.

**When the leader replica has a slower clock,** it will accept most requests into its *early-buffer* but keep them for much longer time. The requests cannot be committed until the leader releases it. Therefore, the slower the leader’s clock is, the long latency Nezha will suffer from.

**When the leader replica has a faster clock,** it will cause two main effects. First, the leader replica will prematurely release requests with large deadlines, causing the subsequent requests unable to be accepted by its *early-buffer*, so the subsequent requests can only be committed in the slow path. Second, the leader will provide overestimated one-way delay (OWD) values and piggyback them to the proxies (recall that the OWD is calculated by using leader’s receiving time to subtract the proxies’ sending time) and cause the proxies to use large latency bound (i.e. the max of

the estimated OWDs from all replicas) for its following requests multicast. However, the second effect is mitigated by DOM-Rs, because we use the clamping function: when the estimated OWD goes beyond the scope of  $[0, D]$ , it will use  $D$  as the estimated value. Therefore, the negative impact due to the leader’s slower clock is constrained. The major impact is that more requests can only be committed in the slow path, which can degrades the latency performance, but the degradation is bounded.

**D.2.2 Bad Clock Synchronization of Follower Replica.** As shown in Figure 18b, similar to the cases in Figure 18a, follower’s bad clock synchronization also inflates the latency. However, the negative impact of follower’s bad clock synchronization is less distinct than leader’s bad clock synchronization: both a faster clock and a slower clock of the follower only cause bounded degradation of latency performance.

**When the follower has a faster clock,** it may prematurely release requests with large deadline and cause subsequent requests not accepted by the *early-buffer* (similar to the case where the leader has a faster clock). But eventually the request can be committed in the slow path, so the slow-path latency will bound the degradation.

**When the follower has a slower clock,** it will hold the requests in its *early-buffer* for longer time. However, if the leader and the other follower(s) have well synchronized clocks, they can



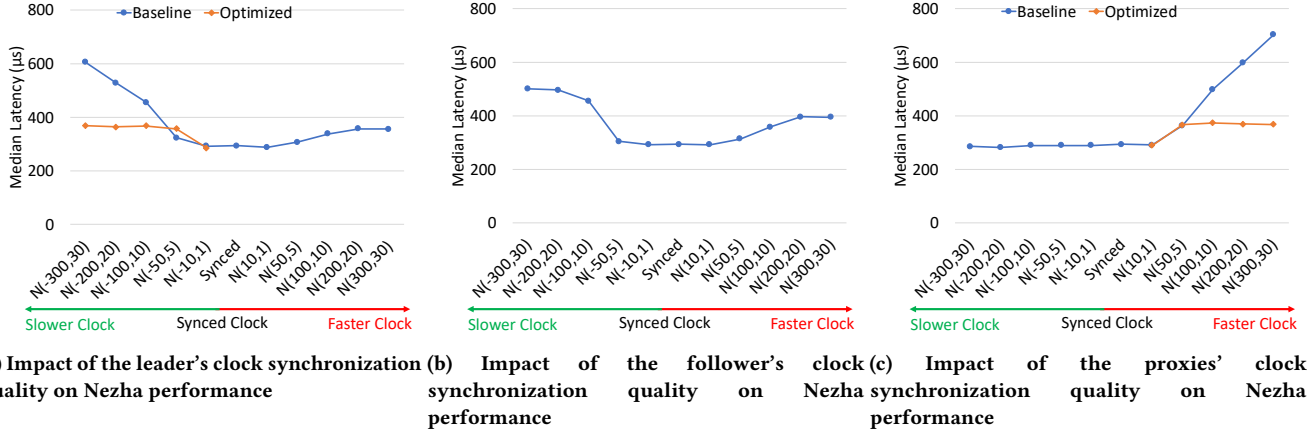


Figure 18: Nezha latency vs. clock synchronization quality

still form a simple majority to commit the request in the slow path. Therefore, this follower's slower clock can not degrade the latency without bounds (whereas the leader's slower clock can).

The major negative impact caused by the follower's faster/slower clock is that, it will lead to inaccurate estimation of OWDs. If the follower has a faster clock, it will piggyback large OWDs to the proxies, thus causing the proxies to choose large latency bound for the following requests. If the follower has a slower clock, it will piggyback small OWDs (or even negative OWDs) to the proxies. However, thanks to the clamping operation during the latency bound estimation, the latency bound will fall back to  $D$  ( $D = 200 \mu s$ ) when the estimated OWD goes too large or negative. In this way, the negative impact of follower's faster/slower clock is constrained.

**D.2.3 Bad Clock Synchronization of Proxy (DOM-S).** As shown in Figure 18c, the proxies' having slower clocks do not cause degradation of the latency performance, so long as replicas have well synchronized clocks. However, the proxies' having faster clocks can lead to unbounded degradation of latency performance.

**When the proxies (DOM-Ses) have slower clocks,** it does not affect the latency so long as replicas are still well synchronized with each other. This is because, although proxies' slower clocks cause smaller sending time, it also leads to larger OWD, which is calculated by the replicas using its local clock time to subtract the sending time. The OWDs are piggybacked to the proxies and eventually lead to large latency bound. Therefore, although the clocks of proxies (DOM-Ses) lag behind, the over-estimated latency bound compensate the lag, and summing up the sending time and latency bound still yields a proper deadline. Therefore, the latency performance does not degrade when proxies have slower clocks.

**When the proxies (DOM-Ses) have faster clocks,** the latency can go up without bound. When there is only a small clock offset occurring (e.g.  $N(10, 1)$ ), i.e. the proxies' clocks are not fast enough, it will not degrade the latency performance of Nezha. This is because, although the faster clock leads to a larger sending time, it also leads to a smaller latency bound, summing them up still yields a proper deadline. However, when proxies' clocks are too fast (e.g.  $N(300, 30)$ ), the sending time becomes even larger than the receiving time obtained at replicas. In this case, replicas (DOM-Rs) will get negative OWD values, so the estimated OWD will

be clamped to  $D$  and piggybacked to the proxies. Then proxies will use  $D$  as the latency bound. Since the proxies' clocks are already faster than the replicas' clocks, the request deadline will become much larger than the replicas' clock time when the request arrives at replicas, leading to long holding delay in the *early-buffer*, and eventually causing much degradation of Nezha's latency performance.

**D.2.4 Optimization: Bounding Latency Degradation.** Reviewing the cases described in §D.2.1-§D.2.3, we note that the leader's slower clock and proxies' faster clocks can cause unbounded latency performance degradation to Nezha. Although such cases do not affect Nezha's correctness and can hardly be long-lasting in practice (because Huygens will keep monitoring its agents and correct the error bounds), we propose an optimization strategy to bound the latency even when such cases of bad clock synchronization become long-lasting. The key idea of the optimization is to let leader force the request to be committed in the slow path.

Recall in the design of DOM (§4), DOM-R will not accept the request into *early-buffer* only if its deadline is smaller than the last released one, which is not commutative to this request (§5.3). We can enforce the entrance condition of *early-buffer*: If the request's deadline is much larger than the current clock time of the leader, which means the request will suffer from a long holding delay if it is put into the *early-buffer*, then the leader also modifies its deadline to be slightly larger than the last released one and then put it into the *early-buffer*. This step is similar to ③ in Figure 4. The difference is, here we modify a large deadline to a smaller one so as to make it release from the *early-buffer* earlier. By contrast, step ③ in Figure 4 is to modify a small deadline to a larger one, so that it will not violate uniform ordering with previously released requests from the *early-buffer*.

The effectiveness of the optimization is shown in Figure 18a and Figure 18c. We configure a threshold for the leader replica: if the request's deadline is larger than the replica's current clock time by  $50 \mu s$ , then the request will not be directly put into the *early-buffer* (as the baseline does). Instead, the leader replica modifies the request's deadline to be slightly larger than the deadline of the last released request (which is not commutative to this

request), so that the request can enter the leader’s *early-buffer* and be released much earlier without violating uniform ordering. Eventually the request can be committed in the slow path. After equipped with the optimization strategy, we can see from Figure 18a and Figure 18c that, the degradation of the latency performance becomes bounded, which provides Nezha with stronger resistance to bad clock synchronization.

In this section, we only discuss the three typical cases. Theoretically, there exists some possibility that these cases can happen simultaneously, which create even more complicated scenarios. For example, when proxies and the leader replica both have slower clocks, the effect due to the bad clock synchronization can be counteracted to some extent. However, the optimization strategy discussed here is still effective to bound the latency degradation and help Nezha to resist the impact of bad clock synchronization.

## E MICRO BENCHMARK OF REORDERING IN PUBLIC CLOUD

### E.1 Measurement of Packet Reordering and Evaluation of DOM Effectiveness

To evaluate the packet reordering in public cloud, we made a micro benchmark as follows. We launch two receivers (denoted as Receiver-0 and Receiver-1) and multiple senders. After the receivers complete receiving the requests, we take Receiver-0’s sequence of requests as the reference, so we get the sequence number of each request based on Receiver-0’s sequence. Then, we use these sequence numbers to check how much reordering has occurred on Receiver-1. We count the request received by Receiver-1 as “reordered request” if (1) its sequential number is smaller than the sequential number of its previous request, or (2) the request does not exist in Receiver-0’s sequence (due to packet drop).

We evaluate in two settings. First, we make each sender send at the rate of 10K reqs/sec (following the Poisson distribution), and vary the number of senders. Second, we only use 2 clients, and varying their submission rates. Each case is run for 5 times and we average the value of the reported number. Figure 19 and Figure 20 show the reordering rate in both settings. Figure 19 shows that, the reordering rate becomes higher as more senders are involved. Even 2 senders (with 10K reqs/sec each) yield the reordering rate of 8.1%. Figure 20 shows the increase of reordering rate as the submission rate increases from 20K reqs/sec to 40K reqs/sec. Although there is not a continuous trend of reordering rate increase, we can see the reordering rate maintains more than 20% when the submission rate varies from 40K reqs/sec to 100K reqs/sec.

After the measurement of packet reordering in public cloud, as shown in Figure 20 and Figure 19, we would like to study the effectiveness of DOM to reduce packet reordering. We use 10 open-loop clients with each submitting at 10K reqs/sec, so the total submission rate is 100K reqs/sec. Recall that DOM is using a sliding window to estimate the OWD (i.e.,  $M$  in the OWD estimation formula in §4). We evaluate the effectiveness of DOM using different percentiles of OWD estimation. In Figure 19 we already measured that the reordering rate in this setting can reach 36%. Then, after we have installed DOM<sup>8</sup>, we can see from Figure 21 that DOM

can effectively reduce the reordering rate. Since a larger percentile will leads to a larger estimated value of OWD and further a longer holding delay of the *early-buffer*, it can enable DOM to rectify more reordering cases. On the other hand, the longer holding delay will be added to the latency to commit the requests.

In order to avoid using a large percentile, Nezha seeks commutativity as an optimization, which has been described in §5.3. Given a fixed latency bound, the reordering cases occur more frequently as the throughput grows. But when commutativity is considered, reordering cases does not matter for commutative requests, which essentially makes DOM handle a “downsampled” throughput. Therefore, DOM does not need the high percentile estimation and DOM-50p is already sufficient for Nezha to achieve a high fast commit ratio (80 %-97 %).

### E.2 Discussion on OWD Variance and Latency Bound Estimation

Essentially, the estimation of latency bound is a trade-off between overall commit/execution latency and the fast commit ratio. A proper latency bound aims to accommodating the variance for *most* conflicting (non-commutative) requests so that they can achieve a low latency overall. The latency bound does not target at cover the latency variance for *all* request (in that case, the latency bound will be determined by the heavy tail of OWD samples). Thus setting the latency bound is essentially an open question of hyper-parameter tuning, which aims to find the sweet spot between improving fast commit ratio and reducing pending delay in *early-buffers*. In general, we have considered two turning approaches.

**Approach 1: Target-based tuning.** We pre-define a target of fast commit ratio, and then we let the proxies monitor the change of the fast commit ratio. When the fast commit ratio becomes lower than the target, the proxies increase the latency bound by a step size. Otherwise, it reduces the latency bound by a step size.

**Approach 2: Measurement-based tuning.** We let the replicas collect the OWD samples from proxies, and then the replicas piggyback an estimation of OWD to the proxies (e.g., by using the sliding window mechanism), so that proxies can decide the latency bound based on the estimated OWDs from each replica (e.g., by taking the maximum as the decided latency bound).

Neither approaches can bypass the tuning of some hyper-parameters. Approach 1 needs to tune the target fast commit ratio and the step size whereas Approach 2 needs to decide the proper percentile used for estimation. In the current implementation of Nezha, we choose Approach 2 because its estimation of latency bound is more intuitive and only includes one hyper-parameter. However, just like the other hyper-parameter tuning problems, there is no “one-size-fits-all” approach to decide a proper latency bound for every scenario. For instance, when commutativity is unavailable and the submission rate is very high (e.g., more than 100K reqs/sec), DOM-50p may be not the best choice for OWD estimation, and choosing a larger percentile (DOM-75p) may be desirable. Nezha is also compatible to any other intelligent tuning

<sup>8</sup>Note that in this experiment, both receivers are following the same way as the Nezha leaders, that is to say, when the packet comes late and cannot enter the *early-buffer*, the receiver will modify its deadline to make it eligible to enter the *early-buffer*, instead of leaving it in the *late-buffer* forever.

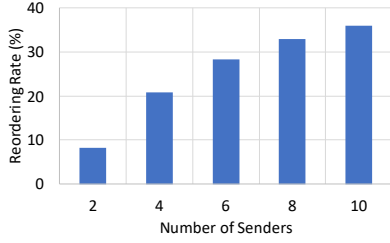


Figure 19: Packet reordering vs. number of senders

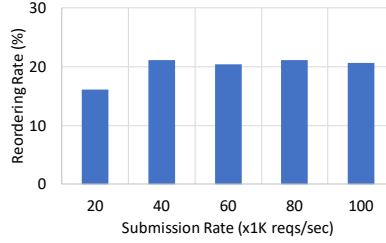


Figure 20: Packet reordering vs. submission rate

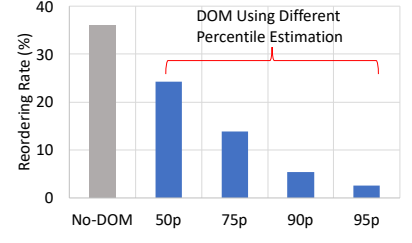


Figure 21: Effectiveness of DOM on packet reordering

algorithms for *adaptive latency bound*, and we leave the study of better tuning algorithms in our future work.

## F DERECHO IN PUBLIC CLOUD

Derecho [40, 81] is a recent high-performance state machine replication system. It works with both RDMA and TCP, and achieves very high throughput with RDMA. Since Derecho is also deployable in public cloud (with TCP), we intend to compare Neza with Derecho in public cloud.

We follow the guidelines from the Derecho team [1]: First, we try to tune the configuration parameters for Derecho and reproduce the performance number in [81] by using bare-metal machines. We set up a cluster in Cloudlab [24]. We use 3 c6525-100g instances (equipped with 100GB RDMA NICs) to form a Derecho subgroup size of 3. We use ramfs [19] as the storage backend for Derecho to avoid disk writes. Then, we evaluate the throughput of Derecho in all-sender mode and one-sender mode. As for the all-sender mode, Derecho yields the throughput of 626K reqs/sec with 1KB message size and 634K reqs/sec with 100B message size. As for the one-sender mode, Derecho yields the throughput of 313K reqs/sec with 1KB message size and 305K reqs/sec with 100B message size. These numbers are close to the reported number in [81], which convinces us that the configuration parameters have been properly set.

Then, we keep using the cluster and the configuration files for Derecho, but switch the backend from RDMA to TCP. After switching to TCP, we find Derecho’s performance drops much: with 100B message size, the all-sender mode achieves the throughput of 17.4K reqs/sec with the median latency of 2.33 ms; the one-sender mode achieves the throughput of 5.68K reqs/sec with the median latency of 2.35 ms. The throughput becomes even lower after we move back to Google Cloud: with 100B message size, the all-sender mode achieves the throughput of 16.5K reqs/sec with the median latency of 2.0 ms; the one-sender mode achieves the throughput of 4.93K reqs/sec with the median latency of 2.54 ms.

We speculate that the low performance of Derecho is due to libfabric it uses for communication. Although libfabric supports both RDMA and TCP communication, it is mainly optimized for RDMA, and the TCP backend is mainly used for test and debug [75]. We expect Derecho can achieve much higher performance if equipped with a better TCP backend. Therefore, we think the comparison is unfair to Derecho and do not include it.

## G ERROR TRACES OF DOMINO

Domino [94] is a recently proposed solution to achieve consensus with clock synchronization. In this section, we show that Domino can violate durability (i.e., the committed requests can be permanently lost) when clock skew occurs. Besides, we conduct a comprehensive discussion on the linearizability violation of Domino.

### G.1 Durability Violation

Domino is derived from Fast Paxos and tries to use clock synchronization to achieve lower commit latency. As what will be shown, Domino does not guarantee the durability property, and further cannot guarantee the consistency and linearizability properties. There are two sources of durability violation that damage Domino’s correctness.

(1) Clock skew/failure. Unlike Fast Paxos, Domino no longer relies on the leader replica to decide the log positions for its requests. Instead, it allows the client to decide log positions for its requests, based on the clock time. Below we will use an error trace to show that, when clock skew happens, client may consider the request as committed, but eventually the request is lost from the replicas.

(2) Stray messages. Fast Paxos is disk-based protocol and writes every request to stable storage before sending the replies, so that the failure of the replica becomes “indistinguishable from its simply pausing” [48]. Domino removes the stable storage and is implemented as a diskless protocol [93], which can also cause durability violation due to stray messages. Since this is a common problem for a group of diskless protocols, including Viewstamped Replication, Speculative Paxos, NOPaxos, etc. We do not extend the discussion in this section, and explain the stray message effect in §A.1 and §H.2 (with NOPaxos as an example).

**Error Trace 1:** There are 5 replicas in Domino, and we denote them as R0-R4. Suppose R0 is the DFP coordinator. There are two requests included in the trace, denoted as request-1 and request-2.

- (1) R1-R5’s clocks are synchronized. R1-R4 report their current clock time  $T$  to the coordinator R0, indicating they have accepted no-ops for all log positions before  $T$  (as described in 5.3.2 of Domino paper [94]).
- (2) R0 receives request-1 with predefined arrival time  $T+1$ . so R0 accepts this request.
- (3) R0 intends to execute request-1. Before execution, R0 syncs request-1 with the other replicas.
- (4) R1 and R2 also accept request-1 and reply to R0, whereas R3 and R4 do not receive request-1 from either the client or the replica due to packet drop.

- (5) R0 considers request-1 is committed because it has received the majority of replies (R1, R2 and itself). R0 considers it safe to execute the request, because R1-R4 have reported T to R0, and R1 and R2 also accept request-1.
- (6) R0 executes the request, but has not broadcast the execution to learners (i.e. the other replicas).
- (7) R1 and R2 fail (i.e. so the NTP services of R1 and R2 also fail). When R1 and R2 are relaunched, the NTP services on their nodes are reinitialized, but the reinitialized NTP gives a time T1, which is smaller than T.
- (8) R3 and R4's NTP services encounter a skew and get a clock time T2, which is smaller than T.
- (9) The client submits request-2, which has a pre-arrival time smaller than T but larger than both T1 and T2.
- (10) R1-R4 all accept request-2 and send replies to the client. The client considers request-2 as committed.
- (11) R1-R4 waits for the notification from the coordinator R0, when the notification arrives, R1-R4 will do either (1) replace request-2 with no-op and only execute request-1 or (2) execute both requests but with request-2 first and request-1 second.

The choice between (1) and (2) in Step 11 depends on how Domino implements the coordination between the leader and the other learners (followers), which is not revealed in the Domino paper [94]. We reviewed the Domino implementation [93] and find that, followers will choose (1) because the DFP leader will also broadcast the log positions (refer to NonAcceptTime variable in [93]) which the leader fills no-ops. When followers choose (1), Domino violates durability because request-2, which have been considered committed, is lost permanently. As an alternative, if followers choose (2), consistency will be violated: After the DFP leader (coordinator) fails and one replica among R1-R4 becomes the new leader, it will have different system state (which executes both request-2 and request-1) from the old leader (which only executes request-1).

The durability violation of Domino is because it overly trusts clocks and "weakens" the leadership of the leader replica in original Fast Paxos. Fast Paxos is still a leader-based protocol, and the leader (which is termed as "coordinator" in the original paper of Fast Paxos [48]) plays a crucial role for its durability/consistency, but is overlooked by Domino during its adaption.

As explained in §4.4 of [78], Fast Paxos requires 1.5 RTTs to commit the request and reply the execution result to clients. The message flow can be described as below:

Client→Replicas→Leader→Client.

As an optimization, when commit and execution are decoupled, it is possible for Fast Paxos to commit in 1 RTT, without the execution result (as explained in §B3.3 of [76]). Domino tries to use this optimization to reduce commit latency. However, the difference between the proper Fast Paxos optimization and Domino design is, even if the request is committed in the fast path (1 RTT), Fast Paxos can only decide the log position for the request until the leader receives a super quorum of consistent Replicas→Leader messages. By contrast, *Domino client takes the leadership from the replica and has already decided the log position of the request during its submission*. In fact, the log positions of Domino are decided by the clock time returned by NTP, but NTP can go wrong and fail to

guarantee the monotonous property. When this happens, Domino's correctness is also damaged.

Nezha suffers no such error cases and the reasons can be summarized into three main aspects.

First, Nezha does not decouple commit and execution, so the history of each request is fixed after the request is committed. In other words, after a request is committed, its log position and all the previous log positions has all been decided. There is no chance for the subsequent requests to overwrite the previous log positions. By contrast, in Domino a later-committed request can also take a log position before the earlier comers, which introduces the potential risks for violating durability/consistency.

Second, the log positions for requests in Nezha are still decided by replicas, not by clients. Although the heavy tasks (i.e. request multicast and quorum check) have been offloaded, Nezha still maintains a strong leadership for the leader replica. When follower's logs have inconsistency with the leader, the leader's logs are considered to be the correct reference and followers are forced to keep consistent with the leader. This also explains why Nezha always requires the leader's reply message during quorum check. By contrast, Domino can bypass the DFP coordinator to commit a request, which weakens its leadership, and eventually causes Error Trace 1.

Third, Nezha exploits synchronized clocks to *reduce packet reordering in the network, rather than directly decide ordering with clock time*. The design of the *early-buffer* maintains the invariant of consistent ordering regardless of clock skew/failure, because the eligibility check for request to enter the *early-buffer* is to compare its deadline with the last released one (§4). Even after the replica fails and recovers, the consistent ordering invariant is still guaranteed: in this case, the *last released request* is the last appended entry in the recovered log (Step 9 in §A.3). By contrast, *Domino directly uses the clock time for ordering*, and does not expect that the clocks can also give a wrong time (Step 7 and Step 8 in Error Trace 1), which leads to the incorrectness for the protocol.

The durability property is a necessary condition for the consistency and linearizability properties:

- Because one committed request can affect the execution result of the subsequent requests, the loss of it will lead to different execution results for the subsequent requests, thus violating consistency.
- Because the committed request can be observed by clients, the loss of it causes contrary observation afterwards, thus violating linearizability.

Error Trace 1 have shown that Domino can violate durability, so the consistency and linearizability properties are not guaranteed either. In §G.2 we will continue to show that, even without considering the violation of durability/consistency property, Domino still violates linearizability.

## G.2 Linearizability Violation

The formal definition of linearizability [35] can be described as: each operation appears to take effect instantaneously and exactly once at some point between its invocation and its response.

Most consensus protocols, such as Multi-Paxos, Raft, Fast Paxos (without commit optimization), Speculative Paxos, NOPaxos,

CURP etc., do not decouple commit and execution, so there is no difference between commit reply and execution reply. To evaluate linearizability with these protocols, “invocation” refers to the point when the client submits the request and “response” refers to the point when the client acknowledges the request has been committed/executed. It can be easily checked that all of them satisfy the definition of linearizability<sup>9</sup>.

However, the other consensus protocols, such as EPaxos, Fast Paxos (with commit optimization), Mencius, etc., decouple commit and execution, so commit and execution no longer happen at the same point. Therefore, it may cause ambiguous understanding of “response” in the original definition of linearizability. The correctness proof [60, 70] of these protocols do not explicitly use the definition of “linearizability”. Instead, they define clearer correctness properties: Mencius [60] defines “total order” whereas EPaxos [70] defines “execution linearizability” to prove correctness. We will use the trace below to show that Domino violates both “total order” and “execution linearizability”.

**Error Trace 2:** There are 5 replicas in Domino, denoted as R0-R4, and R0 is the DFP coordinator. There are two clients, namely C1 and C2, submitting requests to replicas. There are two requests involved, denoted as request-1 and request-2. The two requests are not commutative.

- (1) When  $t = 0$ , C1 submits request-1 to R1 to commit it in the slow path. R1 serves as a DM leader.
- (2) When  $t = 1$ , R1 assigns a timestamp  $t = 10$  to request-1, and ask other replicas to accept it (described in Para. 4 of §5.5 in Domino paper).
- (3) When  $t = 2$ , R1 receives the acceptance reply from R2 and R3, so the majority (R1-R3) have been formed and R1 considers the request is committed and send reply to C1.
- (4) When  $t = 3$ , C1 receives the reply from R1 and also knows request-1 is committed in the slow path.
- (5) When  $t = 4$ , C2 submits request-2 with a pre-estimated arrival time  $t_2 = 9$  for request-2.
- (6) When  $t = 5$ , all replicas receive request-2, so they immediately reply to C2 (described in Para. 6 of §5.4 in Domino paper)
- (7) When  $t = 6$ , C2 receives the replies and knows request-2 is committed in the fast path.
- (8) When  $t = 7$ , R0, the DFP coordinator begins to sync the requests according to the order of log positions, so it first confirms the commit of request-2, and then confirms the commit of request-1.
- (9) When  $t = 8$ , R0 broadcast the information of committed requests to all the other other replicas (learners).
- (10) When  $t = 11$ , every replica has executed the two requests, with request-2 executed first and request-1 executed second.

It can be checked that Error Trace 2 violates “execution linearizability” described in EPaxos [70, 71]: if two interfering commands  $\gamma$  and  $\delta$  are serialized by clients (i.e.,  $\delta$  is proposed only after  $\gamma$  is committed by any replica), then every replica will execute  $\gamma$  before  $\delta$ ). However, Error Trace 2 just goes to the opposite way: the early committed request (request-1) is executed later.

<sup>9</sup>Here we are not considering *stray message* effect for Speculative Paxos and NOPaxos. In §H we will show that they can lose durability and thus violate linearizability due to stray messages.

It can also be checked that Error Trace 2 violates “total order” described in Mencius [60]: If two correct servers  $p$  and  $q$  both commit request  $r_1$  and  $r_2$ , then  $p$  commits  $r_1$  before  $r_2$  if and only if  $q$  commits  $r_1$  before  $r_2$ . Regarding Error Trace 2, let  $p=R0$ ,  $q=R1$ ,  $r_1=\text{request-1}$  and  $r_2=\text{request-2}$ , we show that “total order” is also violated, because R0 commits request-2 first and request-1 second, whereas R1 commits request-1 early in Step 3.

To evaluate whether Domino violates the formal definition of linearizability defined in [35], we need to first clarify the definition of “response” point of Domino. Without considering the violation of durability/consistency in §G.1, one could claim Domino does not violate linearizability if the “response” point is defined as the point when the client receives the execution result from replicas. However, such a definition nullifies the motivation of Domino, which aims to optimize commit latency and assumes the client does not care about the execution results (refer to §5 in Domino paper). Besides, Domino’s design actually sacrifices execution latency because it chooses large values (95th percentile of the probed one-way delays) to decide the pre-defined arrival time for requests to reduce conflict, so its overall execution latency is expected to be larger than that of typical Multi-Paxos or EPaxos. Therefore, we think a meaningful linearizability evaluation for Domino should define the “response” point as the point when the client receives the commit acknowledgement rather than the execution acknowledgement. Below, we can show that Domino still violates linearizability even without the durability/consistency violation described in §G.1.

Formally, the linearizability violation of Domino can be derived as follows: consider the two commit events, the event sequence can be summarized as:

Invocation(C1)→Response(C1)→Invoke(C2)→Response(C2),

and there is no overlap. According to the formal definition of linearizability [35], C1’s request should take effect before C2’s request, which is opposite to the execution order conducted by Domino.

Furthermore, if we consider C1 and C2 as the same client in Error Trace 2, we can find that the error trace also violates sequential consistency [45], which is a weaker correctness condition than linearizability (as mentioned in [10, 35]). According to the description from [10], “sequential consistency requires that all of the data operations appear to have executed atomically, in some sequential order that is consistent with the order seen at individual process”. Obviously, if C1 and C2 are the same client, this individual client sees an order which is opposite of the execution order of its two requests. Therefore, Domino’s design also violates sequential consistency.

The violation of linearizability and sequential consistency is because Domino decouples commit and execution, but has no mechanism to ensure the consistency between commit order and execution order. Given a sequence of committed requests, the execution order of these requests can be arbitrary in Domino because they may be assigned with different arrival-time and lead to arbitrary execution order. Decoupling commit and execution is possible for consensus protocols, such as Fast Paxos, EPaxos and Mencius. However, the consistency between commit order and execution order must be guaranteed:

- (1) Fast Paxos relies on its leader to achieve this. In the original Fast Paxos [48], the leader (coordinator) can only launch new rounds



with larger number than those already started, and the acceptors (followers) can only participate the rounds with growing round numbers (refer to Phase 1(b) of §2.2.1 and Observation 1 of §2.2.2 in [48]). The “round number” in the original paper can be mapped to the “log position” here. Therefore, Fast Paxos does not have such cases that replicas accept a request in higher round (larger log position) and then continue to accept another request in lower round (smaller log position), which is what Domino does.

(2) EPaxos relies on the dependency chain. The dependency chain includes all interfering (non-commutative) requests related to the current request to commit, and the dependency chain is persisted together with the committed request. The dependency chains thus provide enough information for EPaxos to eventually execute the requests in the same order as the commit order.

(3) Mencius partitions the log positions in advance to multiple leaders, but “servers (leaders) cannot commit requests before all previous requests are committed” (Rule 1 in Mencius paper [61]) and fast leaders must wait for slow leaders to guarantee “total order” of the requests. In this way, requests can only be committed with monotonously increasing indices (log positions), and the execution thread can later execute them in the consistent order. Compared with the existing protocols, Domino does not have any mechanism to maintain such consistency between the commit order and the execution order, so it violates linearizability and sequential consistency.

Nezha uses the *deadline*, which is similar to the concept of *expected arrival time* in Domino, to decide the execution order of requests, but Nezha does not violate linearizability because of two design principles.

(1) Nezha does not decouple commit and execution, and the leader serves as the linearization point. The commit of every request cannot bypass the leader. This principle is achieved by the special design of quorum check in fast/slow paths. Regarding Error Trace 2, request-1 will not be prematurely committed without the leader’s agreement in Nezha.

(2) Contrary to Domino, replicas in Nezha do not immediately reply to clients after they receive the request. Instead, replicas in Nezha reply to clients after the replica’s local clock time has passed the *deadline* of the request. In Error Trace 2, if request-1 arrives before  $t = 10$ , it will be held until  $t = 10$ . Meanwhile, if request-2 arrives before  $t = 9$ , it will be replied before request-1. In this way, linearizability is still preserved (i.e. the two commit events overlap and they appear to take effect instantaneously with request-2 first and request-1 second). It is true that the *early-buffer* can cause additional pending time for some requests if the latency bound is too large. Therefore, unlike Domino which chooses the 95th tail latency as the estimation, Nezha chooses moving median as the latency bound and exploits commutativity optimization (rather than rely on a very large latency bound) to improve fast commit/execution ratio and reduce commit/execution latency (details in §5.3).

## H ERROR TRACE OF NOPAXOS RECOVERY

Michael et al [63, 64] discover the incorrectness in the diskless recovery of viewstamped replication [56] and present an error trace in §B.1 of [64]. The cause for the incorrectness is later defined as *stray message* in [44]. We are inspired by the error trace of

viewstamped replication and find that, the similar bugs also exist in the recovery of Speculative Paxos [78] and NOPaxos [53]. It can appear during both view change process (due to *stray messages* between replicas) and the normal quorum check process (due to *stray messages* between replicas and clients). We use two error traces to illustrate them respectively.

### H.1 Stray Messages During View Change

In the terminology of NOPaxos protocol (refer to §5.2.3 in [53]), we describe an error trace to illustrate how *stray messages* can damage the correctness of NOPaxos. Such errors commonly exist in all protocols that inherit the diskless recovery of viewstamped replication whereas Nezha avoids the problem by using *crash-vector* [63, 64] during the recovery process (§A).

**Error Trace 1:** There are 3 replicas in NOPaxos, denoted as R0, R1, and R2.

- (1) Initially, the view is 0, and R0 is the leader. We assume the sequencer does not fail during the following process, and the *session-num* is always 0.
- (2) R1 suspects R0 fails, so R1 initiates a view change: it sets its *status* to VIEWCHANGE, increments its *view-id* from 0 to 1. Then it broadcasts <VIEW-CHANGE-REQ, *view-id*> to all the other replicas. After that, R1 crashes, and the messages are delayed in the network.
- (3) R1 is relaunched and contacts R0 and R2 for recovery (R0 and R2 forms a majority). R0 and R2 have not received the VIEW-CHANGE-REQ messages sent by R1 before crash, so R0 and R2 are still NORMAL in view 0 with R0 as the leader. Then, R1 is recovered as a follower, with *view-id*=0.
- (4) R2 receives the VIEW-CHANGE-REQ message (this is a *stray message* and the recovered R1 has forgotten it!) from R1, so it sets *status* as VIEWCHANGE. Then, it also broadcasts VIEW-CHANGE-REQ messages to all the other replicas, and sends a VIEW-CHANGE message to the leader in the new view (R1). After sending the messages, R2 crashes. The messages are delayed in the network.
- (5) R2 is relaunched and contacts R0 and R1 for recovery (R0 and R1 forms a majority). R0 and R1 have not received the VIEW-CHANGE-REQ messages sent by R2 before crash, so R0 and R1 are still NORMAL in view 0. Then, R2 is recovered as a follower, with *view-id*=0.
- (6) The VIEW-CHANGE-REQ message is lost to R0, but both the VIEW-CHANGE-REQ message and VIEW-CHANGE message arrives at R1 (this is another *stray messages*, and the recovered R2 has forgotten it!). R1 then initiates the view change: incrementing *view-id* from 0 to 1, broadcasting VIEW-CHANGE-REQ messages and sending a VIEW-CHANGE message to the new leader. It notices itself is the new leader, so it collects VIEW-CHANGE from itself and R2, thus forming a majority.
- (7) R1 thinks it has got the VIEW-CHANGE messages from the majority (though R2 has forgotten it has sent that), so it merges the logs from the two VIEW-CHANGE messages, resets its *status* to NORMAL, and becomes the leader in view 1. Meantime, R0 and R2 are both in view 0 and R2 still believes R0 is the leader.

- (8) A request is multicast to all replicas but is lost to R1. Both R0 and R2 receive the request and reply to the client. The client gets replies from a majority of replicas with matching *view-ids* (0), and the leader (R0) is also included, so the client considers it as committed.
- (9) R0 crashes and is relaunched. It contacts R1 and R2 for recovery, and notice the highest *view-id* is 1, so it does a state transfer with the leader (R1) in view 1, and copies R1's state, which does not include the committed request.
- (10) R2 also crashes after R0 is recovered. It follows the same workflow to recover and also copies the state from R1, and finally becomes a follower in view 1. Then, the committed request is lost forever.

During the whole process, there is no violation of NOPaxos assumptions, i.e. at most  $f = 1$  replicas fail at the same time. However, the final result does not guarantee durability of committed request, and can further damage consistency and linearizability.

## H.2 Stray Messages During Quorum Check

The stray reply messages can also affect client's quorum check and cause permanent loss of committed requests. We use the following error trace to illustrate this.

**Error Trace 2:** There are 3 replicas in NOPaxos, denoted as R0, R1, and R2, and R0 is the leader.

- (1) The client submits a request to the replicas, and the request is tagged with the sequential number of 1 at the sequencer, and then multicast to all replicas.
- (2) The request is received by R1, and is lost to R2, and is delayed in the network while transmitting to R0.
- (3) R1 sends a reply to the client. After that, R1 crashes and is relaunched. It requests the state transfer from the leader, R0. Since R0 has an empty log list, R1 also has an empty log list.
- (4) The delayed request finally arrives at R0, so R0 sends a reply to the client, then the client has got replies in matching *view-ids* and *log-slot-nums* from  $f + 1$  replicas, including the leader, so the client considers the request as committed.
- (5) R0 crashes, then R1 and R2 launch the view change process and finally elects R1 as leader. Both R1 and R2 have empty log lists.
- (6) R0 is relaunched, and requests state transfer from R1, so R0 also gets an empty log list.
- (7) A new request is multicast by the sequencer (with the sequential number of 2) and arrives at all replicas. Then every replica notices there is a gap occurring. Therefore, the leader starts gap agreement and fill a NOOP in the first slot.
- (8) So far, the request with sequential number of 1, which has been committed, is lost permanently.

Error Trace 2 shows that, even if the view change process is executed correctly, the stray messages between replicas and clients can still violate the durability property. Compared with Error Trace 1, Error Trace 2 is even more common and can happen for many diskless protocols which conduct client-side quorum check. The general pattern for such error cases can be summarized as follows:

**General Error Pattern.** Imagine a request is delayed in the network, whenever it arrives at one replica, that replica sends a

reply and immediately crashes afterwards, then the crashed replica recover from the others and gets an empty log list (because the other replicas have not received the request). After each replica completes such behavior, the client gets replies from all the replicas but actually none of them is holding the request. Such a pattern does not violate the failure model, but causes permanent loss of committed requests.

Reviewing the existing protocols, Speculative Paxos, NOPaxos and Domino all suffer from such cases. CURP [76] can avoid the *stray message* effect by assuming the existence of a configuration manager, which never produces stray messages (e.g., by using stable storage). Whenever the witnesses crash and are relaunched, the configuration manager need to refresh the information for the master replica as well as the clients, so that clients can detect the stray messages during quorum check and avoid incorrectness.

Nezha avoids such error cases by including the information of *crash-vector* in the hash of *fast-replies*, which prevent stray reply messages from forming the superquorum in the fast path and creating an *illusion* to the proxies/clients. We analyze in more details below.

Regarding the general pattern above,

- (1) When the follower(s) fail, they need to contact the leader and complete the state transfer before their recovery (Algorithm 1).

- If the leader has already received the request before the state transfer, then after the follower's recovery, it can remember the *fast-reply* that it has sent before crash, and can replay it. In this case, the *fast-reply* is not a stray message.
- If the leader has not received the request before the state transfer, then the leader's *crash-vector* will be updated after receiving the follower's STATE-TRANSFER-REQ (line 74-75 in Algorithm 1), which includes a different *crash-vector* (the follower has incremented its own counter). Therefore, the *hash* of the leader's *fast-reply* is computed with the aggregated *crash-vector*, and will be different from that included in the *fast-reply* (stray message) sent by the follower before crash, i.e. the leader's *fast-reply* and the followers' stray *fast-replies* cannot form a superquorum.

- (2) When the leader fails, based on Algorithm 2, the view change will elect a new leader. *crash-vectors* ensure the view change process is not affected by *stray messages*. After the view change is completed, the *view-id* is incremented. At least  $f + 1$  replicas after the view change will send *fast-replies* with higher *view-ids*. Because the quorum check requires reply messages have matching *view-ids*, the *stray fast-replies* (sent by the old leader) can not form a super-quorum together with the *fast-replies* sent by the replicas after the view change.

Nezha's slow path does not suffer from *stray message* effect, because there is causal relation between leader's state update (advancing its *sync-point*) and followers' sending *slow-replies*.

- (1) When followers crash and recover, they copy the state from the leader. The followers' state before crash is no fresher than their recovered state, so the followers have no *stray slow-replies*, i.e. the followers can remember the *slow-replies* they have sent before crash and can replay them.

- (2) When the leader crashes and recovers, it can only rejoin as a follower replica after the new leader has been elected (§A.2), so

the old leader’s reply messages before crash have smaller *view-ids*, compared with the *slow-replies* of replicas after the view change. With matching *view-ids*, these reply messages cannot form a quorum together in the slow path.

## I WHY IS STABLE STORAGE NECESSARY FOR RAFT?

Paxos and Viewstamped Replication (VR) have developed diskless version in [15] and [56], respectively, and the internal bugs of both diskless protocols can be fixed using crash vectors [63, 64]. In contrast, Raft, since it was invented [74], has not developed a diskless version so far. Its correctness has more dependency on the stable storage than the other two protocols. In other words, Raft will no longer be correct if stable storage is removed from the protocol. Here we provide an error trace of Raft without stable storage.

There are 5 replicas in Raft, denoted as R0-R4. R0 is the leader.

- (1) R0 receives a request from the client, and appends it to the other replicas.
- (2) R1 and R2 reply to the AppendEntries RPC from R0. Before replying, R1 and R2 should have persisted the request to its stable storage. In this error trace, we will see how Raft goes incorrect if it does not have stable storage for log persistence.
- (3) The AppendEntries RPC is lost to R3 and R4.
- (4) R0 receives the replies from R1 and R2. It forms a quorum including R1, R2 and itself, so it replies to the client, notifying it of the commit result.
- (5) The client considers the request as committed.
- (6) R1 fails, and recovers immediately. Since there is no stable storage, its recovered log is empty. Note that Raft does not have a state transfer mechanism like Paxos and VR, so R1 does not ask logs from the other replicas during recovery. It just recovers with an empty log list, forgetting it has once appended the request and replied to R0.
- (7) R0 is network partitioned or fails. The other replicas fail to hear from R0 for sufficiently long time, so they decide to elect a new leader.
- (8) R3 (with an empty log list) becomes a candidate and sends RequestVote RPC to the others.
- (9) R4 and R1 give their votes to R3, so R3 gets the votes from the majority including itself, and becomes the new leader.
- (10) R3 overwrites the logs of all the other replicas, then the committed request is lost forever.

The root cause for this error trace comes from Raft’s leader election mechanism, which takes log persistence as a necessity. Unlike VR, which takes a round-robin way to decide the leader (so that the leader is deterministic in every new view), Raft gives more freedom for replicas to compete for the leadership in each term. The voters decide whether the candidate is qualified by checking whether the candidate’s log is more *up-to-date* than itself.

In the error trace, R1 forgets its log in Step 6 without stable storage. Therefore, it mistakenly considers R3 is qualified to become the new leader in Step 9. By contrast, if it had stable storage and remembered its log after recovery, R1 should have rejected the

RequestVote RPC from R3, because R3’s log is not as *up-to-date* as its log. In that way, R3 could not become the new leader in Step 9.

## J FORMAL COMPARISON OF DIFFERENT PRIMITIVES

Concretely, the mostly ordered multicast (MOM) primitive [78] used by Speculative Paxos creates a network environment to make most requests arrive at all replicas in the same order. The ordered unreliable multicast (OUM) primitive [53] used by NOPaxos ensures ordered delivery of requests without a reliability guarantee using a programmable switch as a request sequencer. By contrast, the deadline-ordered-multicast (DOM) primitive used by Nezha leverages clock synchronization to guarantees uniform ordering, so as to ease the work for replication protocols to achieve state consistency (i.e. to satisfy both uniform ordering and set equality). In this section, we aim to make a formal comparison among the three primitives.

### J.1 Notation

- Replicas:  $R_1, R_2, \dots$
- Messages:  $M_1, M_2, \dots$
- $a(M_i, R_k)$ : the arrival time of  $M_i$  at  $R_k$ . It is **reference time which is not accessible by replicas**, replicas can only get an approximate  $\hat{a}(M_i, R_k)$  by calling their local clock API once  $M_i$  arrives.
- $r(M_i, R_k)$ : the reference time at which  $M_i$  is released by the primitive to  $R_k$ ’s protocol. It is the reference time. The primitive does not deliver a reference time  $r(M_i, R_k)$  to replication protocols, instead, it delivers an approximate  $\hat{r}(M_i, R_k)$  by calling the clock API before releasing  $M_i$ .
- $S(M_i)$ : the sequential number of  $M_i$  given by Sequencer.(OUM Oracle Information).
- $D(M_i)$ : the **planned** deadline of  $M_i$  to arrive at all replicas (DOM Oracle Information). Replicas know the value of  $D(M_i)$  but cannot decide when is exactly the time point of  $D(M_i)$  by simply checking their local clock.

### J.2 Definition

- Packet drop:  $M_i$  is lost to  $R_x$  if  $r(M_i, R_x) = \infty$
- Uniform ordering:  $R_1$  and  $R_2$  are said to be *uniformly ordered* (denoted as  $UO(R_1, R_2, M_1, M_2)$ ) with respect to  $M_1$  and  $M_2$  if:
  - $r(M_1, R_1) > r(M_2, R_1)$  and  $r(M_1, R_2) > r(M_2, R_2)$
  - Or  $r(M_1, R_1) < r(M_2, R_1)$  and  $r(M_1, R_2) < r(M_2, R_2)$

For simplicity, we omit discussing the edge case  $r(M_1, R_1) = r(M_1, R_2)$  and/or  $r(M_1, R_2) = r(M_1, R_2)$ , which can be categorized into either of the two aforementioned outcomes. Similar edge cases are also omitted in the discussion of §J.3.

- Set equality:  $R_1$  and  $R_2$  are *set-equal* with respect to  $M_1$  (denoted as  $SE(R_1, R_2, M_1)$ ) if
  - $r(M_1, R_1) = \infty$  and  $r(M_1, R_2) = \infty$
  - Or  $r(M_1, R_1) < \infty$  and  $r(M_1, R_2) < \infty$

Set equality is similar to the term *reliable delivery* in NOPaxos [53]. While NOPaxos describes the property from the network perspective, our description is more straightforward by describing it from the replica perspective.

- Consistency:  $R_1$  and  $R_2$  are consistent if

$$\forall M_i, M_j : \quad UO(R_1, R_2, M_i, M_j) \\ \& \quad SE(R_1, R_2, M_i) \& \quad SE(R_1, R_2, M_j)$$

Satisfying both  $UO$  and  $SE$  property is equivalent to implementing an atomic broadcast primitive [23], which is as hard as the consensus protocol.

### J.3 Primitive Actions

Given a replica  $R_k$ , and two messages  $M_1$  and  $M_2$ , we can formally describe the actions of the three primitives as follows.

#### J.3.1 MOM.

$$\begin{aligned} r(M_1, R_k) &= a(M_1, R_k) \\ r(M_2, R_k) &= a(M_2, R_k) \end{aligned} \quad (1)$$

$r(*, *)$  is completely determined by  $a(*, *)$  without guaranteeing uniform ordering.

**J.3.2 OUM.** Without loss of generality, the OUM Oracle gives  $S(M_1) < S(M_2)$ .

If  $a(M_1, R_k) < a(M_2, R_k)$  (**Branch 1**), then

$$\begin{aligned} r(M_1, R_k) &= a(M_1, R_k) \\ r(M_2, R_k) &= a(M_2, R_k) \end{aligned} \quad (2)$$

Otherwise  $a(M_1, R_k) > a(M_2, R_k)$  (**Branch 2**), then

$$\begin{aligned} r(M_1, R_k) &= \infty \\ r(M_2, R_k) &= a(M_2, R_k) \end{aligned} \quad (3)$$

Equation 2 captures the case where  $M_1$  and  $M_2$  arrive in an order *consistent* with their sequence numbers.

Equation 3 captures the case where  $M_1$  and  $M_2$  arrive in an order *inconsistent* with their sequence numbers, in which case  $M_1$  is immediately declared lost.

Uniform ordering is guaranteed by OUM because messages arrive at different replicas either consistent with their sequence numbers (which are unique to a message and not a replica) or messages are declared lost.

**J.3.3 DOM.** To simplify the following comparison analysis, we assume the local clock of each replica is monotonically increasing, which is a common assumption in clock modelling [50, 57, 59]. However, it is worth noting that, the correctness of Nezhā does not require this assumption: recall that the replicas compare the deadlines of the incoming requests with the *last released request* to decide whether the incoming request can enter the *early-buffer* (§4). Therefore, even if the local clock time goes back and forth, the entrance condition of the *early-buffer* still preserves uniform ordering among the released requests. The clock behavior only affects the performance of Nezhā but not its correctness.

DOM can satisfy the monotonically increasing property as follows: DOM tracks the returned value every time it calls the clock API. If the returned value is smaller than the last one (i.e. violating the monotonically increasing property), DOM disposes the value and retries the clock API. When the replica fails, DOM can rely on the replication protocol to recover the committed logs, and then it starts using the clock time which is larger than the

deadline of the last log entry. In this way, DOM guarantees that each replica clock follows monotonically increasing property.

The *monotonically increasing clock time* leads to the following fact:

$$\begin{aligned} r(M_1, R_k) < r(M_2, R_k) &\iff \hat{r}(M_1, R_k) < \hat{r}(M_2, R_k) \\ a(M_1, R_k) < a(M_2, R_k) &\iff \hat{a}(M_1, R_k) < \hat{a}(M_2, R_k) \end{aligned} \quad (4)$$

Without loss of generality, assume DOM Oracle decides two deadlines  $D(M_1)$  and  $D(M_2)$ , satisfying  $D(M_1) < D(M_2)$ .

If  $\hat{a}(M_1, R_k) < \hat{a}(M_2, R_k)$  or  $\hat{a}(M_1, R_k) < D(M_2)$  (**Branch 3**), then

$$\begin{aligned} \hat{r}(M_1, R_k) &= \max\{D(M_1), \hat{a}(M_1, R_k)\} \\ \hat{r}(M_2, R_k) &= \max\{D(M_2), \hat{a}(M_2, R_k)\} \end{aligned} \quad (5)$$

Based on the condition and the formula, it is easy to check that  $\hat{r}(M_1, R_k) < \hat{r}(M_2, R_k)$ , thus  $r(M_1, R_k) < r(M_2, R_k)$ .

Otherwise  $\hat{a}(M_1, R_k) > \hat{a}(M_2, R_k)$  and  $\hat{a}(M_1, R_k) > D(M_2)$  (**Branch 4**), then

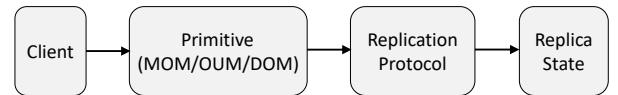
$$\begin{aligned} \hat{r}(M_1, R_k) &= \infty \\ \hat{r}(M_2, R_k) &= \max\{D(M_2), \hat{a}(M_2, R_k)\} \end{aligned} \quad (6)$$

Equation 5 captures the cases where either  $M_1$  arrives before  $M_2$  arrives, or  $M_1$  arrives before  $M_2$ 's deadline (based on the local clock of the replica). In both cases,  $M_1$  can be released to the protocol before  $M_2$  is released.

Equation 6 captures what happens when  $M_1$  arrives after both  $M_2$ 's deadline and  $M_2$ 's arrival. Here,  $M_1$  has to be declared lost (so that both  $\hat{r}(M_1, R_k)$  and  $r(M_1, R_k)$  are  $\infty$ ) and  $M_2$  is released to the protocol.

Uniform ordering is guaranteed by DOM because messages are released to replicas according to their deadline's order. Those which have violated the increasing deadline order will not be released by DOM and should be handled by the replication protocol.

### J.4 Understanding Difference between MOM, OUM and DOM



**Figure 22: General Model of Speculative Paxos/NOPaxos/Nezhā**

As shown in Figure 22, primitives are decoupled from the replication protocol. None of the primitives guarantees consistency defined in §J.2. The primitives are just used to create favorable message sequences for the replication protocol to achieve consistency more efficiently.

Considering two replicas  $R_1$  and  $R_2$ , with  $M_1$  and  $M_2$  arriving at both replicas. We aim to study the question: When equipped with MOM/OUM/DOM, how likely (easily) can  $R_1$  and  $R_2$  reach consistency without sacrificing liveness (i.e. both  $M_1$  and  $M_2$  should appear in the consistent replica state) given the message sequences output from the primitive?

- MOM simply relies on the highly-engineered network to remove inconsistency flavor, and its output is exactly the output of the network. There is no guarantee on either uniform ordering or set equality. When it comes to the general network (e.g. public cloud), Rarely both  $r(M_1, R_1) < r(M_2, R_1)$  and  $r(M_1, R_2) < r(M_2, R_2)$  (or the other direction  $r(M_1, R_1) > r(M_2, R_1)$  and  $r(M_1, R_2) > r(M_2, R_2)$ ) are satisfied at the same time, thus most consensus work still needs to be undertaken by the replication protocol.

- OUM is potentially better than MOM, because it does the serialization between the clients and replicas with a standalone sequencer, so that the reordering occurrence in the path between clients and the sequencer does not matter. However, when reordering happens in the path between the sequencer and replicas, it leads to **Branch 2**, thus the replication protocol (e.g. NOPaxos) has to handle the loss of  $M_1$  for the affected replicas (e.g. fetching from other replicas or starting gap agreement). Although the consistency property is still satisfied if all replicas take **Branch 2**, that leads to liveness problem: the client which submits  $M_1$  has to retry a new submission. If it goes to the extreme case, when clients submit a series of requests and only the one with the largest sequential number arrives first on all replicas, then all the other requests are declared loss by OUM. In this case, the replicas reaches consistency, but little progress is made.

- DOM performs better than OUM in general network because it maintains stronger resistance to reordering. Based on the Equation 4, we can easily derive that **Branch 3** of DOM is a super set of **Branch 1** of OUM. In other words, replicas equipped with DOM are more likely to take DOM's "good" branch (i.e. **Branch 3**), whereas replicas equipped with OUM are less likely to take OUM's "good" branch (i.e. **Branch 1**). However, DOM's strong resistance is obtained at the expense of extra pending delay. According to Equation 6, even when  $M_1$  and  $M_2$  comes in order and before their deadlines, they still need to be held until  $D(M_1)$  and  $D(M_2)$ . By contrast, OUM can immediately present  $M_1$  and  $M_2$  to the replication protocol, according to Equation 2.

### J.5 Why does Clock Synchronization Matter to DOM?

Clock synchronization affects the effectiveness of DOM for two reasons. First, clock synchronization affects whether DOM can resist the reordering. Second, clock synchronization is closely related to the measurement of client-replica one-way delay, thus (indirectly) affecting whether the client can decide a proper deadline for its messages (requests). We use two cases to illustrate how bad clock synchronization and bad deadlines can affect DOM's effectiveness, and use one case to illustrate the effective DOM with good clock synchronization and proper deadlines.

**Bad Case-1:** Bad clock synchronization.  $M_1$  and  $M_2$  arrive at  $R_1$  out of order but  $a(M_1, R_1) < D(M_2)$ . Meanwhile, the two messages arrive at the other replicas in order. If  $R_1$ 's clock had been well synchronized with the reference clock,  $\hat{a}(M_1, R_1)$  should be very close to  $a(M_1, R_1)$ , leading to  $\hat{a}(M_1, R_1) < D(M_2)$ , and then DOM should be able to rectify the reordering on  $R_1$ , so that it outputs the consistent message sequence as the others. However,  $R_1$ 's clock fails at that time and gives a very large  $\hat{a}(M_2, R_1)$  that leads to  $\hat{a}(M_2, R_1) > D(M_3)$ . In this case, DOM becomes ineffective and  $R_1$

takes **Branch 4**, leaving more consensus work for the replication protocol to complete.

**Bad Case-2:** Improper deadline. Suppose the clock synchronization goes wrong on some replicas (e.g.  $R_2$ ), and the clocks on the problematic replicas are much faster than the reference clock, so the one-way delay (OWD) measurement gives very large value and elevates the latency bound estimation (§4). When  $M_1$  and  $M_2$  are given very large deadlines  $D(M_1)$  and  $D(M_2)$ . The replicas (e.g.  $R_1$ ) will take **Branch 3** and DOM is able to rectify possible reordering. However,  $M_1$  suffers from the pending time of  $D(M_1) - \hat{r}(M_1, R_1)$  whereas  $M_2$  suffers from the pending time of  $D(M_2) - \hat{r}(M_2, R_1)$  on  $R_1$  (Assume  $R_1$ 's clock is well synchronized with the reference clock).

**Good Case:** Clocks are well synchronized and  $D(M_i)$ s are properly decided, i.e.  $D(M_i)$  is close to (but slightly larger than) the arrival time  $a(M_i, R_x)$  regarding most replicas. In this case, when network is good, DOM delivers the message to the replication protocol with both uniform ordering and set equality, just like MOM and OUM. More than that, when the network causes messages reordering, both MOM and OUM will present the reordering effect to the replication protocol, and triggers the replication protocol to take extra effort. Specifically, MOM presents non-uniformly ordered messages to the replication protocol which causes Speculative Paxos to go to slow path and costly rollback; OUM presents uniformly order messages with gaps (equation 3), which also causes NOPaxos to go to slow path and make the following messages pending before the gap is resolved. By contrast, so long as the out-of-order message ( $M_1$ ) does not break the deadline ( $D(M_2)$ ) of the message ( $M_2$ ), the reordering between  $M_1$  and  $M_2$  can be rectified by DOM in equation 5 and is insensible to the replication protocol, so that the workload of replication protocol (Nezha) is much relieved.

## K NEZHA VS. EPAXOS IN WAN

EPaxos [71, 84] is a consensus protocol which proves to outperform Multi-Paxos in Wide Area Network (WAN) scenario. EPaxos fully exploits the fact that Local Area Network (LAN) message delays are negligible when compared with WAN message delays. Therefore, EPaxos distributes its replicas across multiple zones. Such design enjoys two benefits: First, the long-distance (cross-zone) communication between replicas are fully controlled by the service providers, so the service providers can use private backbone network to provide better quality of service. By contrast, if replicas are co-located together and far away from clients. The long distance from clients to replicas are out of control, which may cause longer latency and more frequent message drop. Second, Although EPaxos also incurs 2 RTTs in the fast path, one of them is LAN RTT (i.e. client→replica and replica→client message delays) that can be ignored. Therefore, EPaxos claims to achieve optimal RTT (1 WAN RTT) in the fast path and 2 RTTs in the slow path, which makes it outperform Multi-Paxos in latency. Besides, by using the multi-leader design and commutativity, EPaxos also enjoys less throughput bottleneck compared with Multi-Paxos.

While in this paper we only focus on LAN deployment and have shown that Nezha outperformed TOQ-based EPaxos in LAN (Figure 6), we would like to highlight that Nezha is also deployable in WAN environment, and we believe Nezha can also earn more advantages over EPaxos when deployed in WAN. We analyze the

advantages as below and leave the experimental evaluation as our follow-up work.

### K.1 Latency

When deployed in WAN, Nezha shares the same benefit as EPaxos: Nezha deploys its stateless proxies in every zone, so the client→proxy and proxy→client message delays are also LAN message delays that can be ignored. Therefore, Nezha also achieves 1 WAN RTT as EPaxos, but Nezha achieves only 1.5 WAN RTTs in the slow path, compared with 2 WAN RTTs achieved by EPaxos.

Besides, Nezha earns more performance advantages over EPaxos when there are more zones than replicas. For instance, consider a 3-replica consensus protocol with 10 different zones, and clients are evenly distributed in every zone, EPaxos cannot benefit all clients regarding the latency. Since there are only three replicas, at most the clients in three zones can enjoy 1 WAN RTT to commit their requests in the fast path. The majority of clients (70%) still suffer 2 WAN RTTs to commit in the fast path, and even worse (3 WAN RTTs) to commit in the slow path. The large number of zones make EPaxos lose most of its latency benefit. In order to let all clients enjoy 1 WAN RTT fast path, EPaxos has to deploy one replica in each zone (i.e. 10 replicas), but in that case, the quorum check will become much heavier and more interference/conflicts among replicas can occur. In contrast, Nezha *distributes proxies instead of replicas across zones, and proxies are highly scalable*. Regardless of the number of zones, Nezha can still maintain 1 WAN RTT for all clients, so long as sufficient proxies are deployed in every zone.

Besides, when data center failure is not considered (i.e. the number of zone failures is assumed to be 0), Nezha can even co-locate all replicas in the same zone and connect them with high-end communication (e.g. DPDK, RDMA). In this case, inter-replica communication is also LAN message delays, and Nezha can achieve optimal WAN RTT (1 WAN RTT) for both fast path and slow path, which gives Nezha more latency benefit than EPaxos.

### K.2 Throughput

While EPaxos uses multiple leaders to mitigate single-leader bottleneck, Nezha adopts an alternative design: Nezha still

maintains single leader but offload most workload to proxies. The proxy brings two major advantages for Nezha regarding the throughput. First, the inter-replica communication is much more lightweight because the leader only multicast index messages (rather than request messages) to other followers, which have much smaller sizes than requests and can be batched to amortize the communication cost. Second, replicas do not undertake quorum check, and proxies can conduct the quorum check concurrently. Although EPaxos can share the workload of request multicast and quorum check among replicas, the number of replicas is limited and it is still likely that the quorum check workload can overwhelm the capacity of multiple leaders. However, the number of proxies in Nezha can be considered without constraint (i.e. as many as Huygens can support), and Nezha can deploy as many proxies as needed to tackle the workload of request multicast and quorum check. Therefore, we expect Nezha can also achieve higher throughput than EPaxos.

### K.3 Clock Synchronization in WAN

As mentioned in our paper, the performance of Nezha is closely related to the synchronization performance of clocks. A reasonable concern about deploying Nezha in WAN is that the clock error can become very large and cause performance degradation. Such concerns prove to be unnecessary. According to the discussion with the developer team of Huygens, when deployed in public cloud across multiple data centers, the clock accuracy provided by Huygens will be in the order of 10s of microseconds, with occasional spikes if the WAN link is unstable. Such claims have been verified in [84], which evaluates Huygens in the WAN setting and observes the clock offsets between 20  $\mu$ s and 1 ms. Considering the inter-datacenter latency is usually tens of or even hundreds of milliseconds (as shown in Figure 5 of [84]), the synchronization performance of Huygens is sufficient for Nezha to achieve fast consensus in WAN.

## L NEZHA TLA+ SPECIFICATION

The TLA+ specification of Nezha is available at the anonymous repository <https://anonymous.4open.science/r/Nezha-89BF/Nezha.tla>.