Nezha: Deployable and High-Performance Consensus Using Synchronized Clocks

[Technical Report]
Jinkun Geng*, Anirudh Sivaraman*, Balaji Prabhakar*, Mendel Rosenblum*  
*Stanford University, *New York University

ABSTRACT
This paper presents a high-performance consensus protocol, Nezha, which can be deployed by cloud tenants without any support from their cloud provider. Nezha bridges the gap between protocols such as Multi-Paxos and Raft, which can be readily deployed and protocols such as NOPaxos and Speculative Paxos, that provide better performance, but require access to technologies such as programmable switches and in-network prioritization, which cloud tenants do not have.

Nezha uses a new multicast primitive called deadline-ordered multicast (DOM). DOM uses high-accuracy software clock synchronization to synchronize sender and receiver clocks. Senders tag messages with deadlines in synchronized time; receivers process messages in deadline order, on or after their deadline.

We compare Nezha with Multi-Paxos, Fast Paxos, Raft, a NOPaxos version we optimized for the cloud, and 2 recent protocols, Domino and TOQ-EPaxos, that use synchronized clocks. In throughput, Nezha outperforms all baselines by a median of 5.4× (range: 1.9–20.9×). In latency, Nezha outperforms five baselines by a median of 2.3× (range: 1.3–4.0×), with one exception: it sacrifices 33% latency compared with our optimized NOPaxos in one test. We also prototype two applications, a key-value store and a fair-access stock exchange, on top of Nezha to show that Nezha only modestly reduces their performance relative to an unreplicated system. Nezha is available at https://github.com/Steamgjk/Nezha.

1 INTRODUCTION
Our goal in this paper is to build a high-performance consensus protocol which can be deployed by cloud tenants with no help from their cloud provider. We are motivated by the fact that 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 motivating our work below.

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 [13, 53, 62], thereby losing consistency guarantees over the whole data. Second, microsecond-scale applications are pushing the limits of computing [3, 27, 31, 39, 42]. Such applications often have stateful components that must be made fault-tolerant (e.g., the matching engine within a fair-access cloud stock exchange [21], details in §10). To effectively support such applications on the public cloud, we need the consensus protocol to provide low latency and high throughput.

Despite significant improvements in consensus protocols over the years, the status quo falls short in 2 ways. First, protocols such as Multi-Paxos [37] and Raft [58] can be (and are) widely deployed without help from the cloud provider. However, they only provide modest performance: latency in the millisecond range and throughput in the 10K requests/second range [17]. Second, high-performance alternatives such as NOPaxos [41], Speculative Paxos [63], NetChain [30], NetPaxos [30], and Mu [3], require technologies such as programmable switches, switch multicast, RDMA, priority scheduling, and control over routing—most of which are out of reach for the cloud tenant.1

Here, we develop a protocol, Nezha, that provides high performance for cloud tenants without access to such technologies. Our starting point in designing Nezha is to observe that a common approach to improve consensus protocols is through optimism: in an optimistic protocol, there is a common-case fast path that provides low client latency, and there is a fallback slow path that suffers from a higher latency. Examples of this approach include Fast Paxos [36], EPaxos [56], Speculative Paxos [63], and NOPaxos [41].

For optimism to succeed, however, the fast path must indeed be the common case, i.e., the fraction of client requests that take the fast path should be high. For a sequence of client requests to take the fast path, these requests must arrive in the same order at all servers involved in the consensus protocol. In the public cloud, however, cloud tenants have no control over paths from clients to these servers. As we empirically demonstrate in §3, this leads to frequent cases of reordering: client requests arrive at servers in different orders. Thus, for an optimistic protocol to improve performance in the public cloud, reordering must be reduced. This observation influenced the design of Nezha, which has 3 key ideas.

Deadline-ordered multicast. Nezha uses a new network primitive, called deadline-ordered multicast (DOM), designed to reduce the rate of reordering in the public cloud. DOM is a type of multicast 2 that works as follows. The senders’ and receivers’ clocks are synchronized to each other to produce a global time shared by the sender and all receivers. The sender attaches a deadline in global time to its message and multicasts the message to all its receivers. Receivers process a message on or after its deadline, and process multiple messages in the increasing order of deadline. Because the deadline is a message property and common across all receivers of a message, ordering by deadline provides the same order of processing at all receivers and undoes the reordering effect. DOM is best effort: messages arriving after their deadlines or lost

1We note that many of these technologies are available to cloud providers, but in most cases they are not exposed to tenants of the cloud. RDMA instances [54] are an exception, but such instances are expensive.

2Unless otherwise specified, “multicast” in this paper refers to application-based multicast, because switch-based multicast is not supported in cloud environment.
messages are no longer DOM’s responsibility. Thus, for DOM to be effective, the deadline should be set so that most messages arrive before their deadlines—despite variable network delays and despite clock synchronization errors. However, if messages arrive after their deadlines, correctness is still maintained because Nezha falls back to the slow path. Here, DOM follows Liskov’s suggestion of using accurate clock synchronization for performance improvements, but not as a necessity for correctness [43].

**Speculative execution.** DOM combats reordering and increases the fraction of client requests that take the fast path. Our next idea reduces client latency of Nezha in the slow path, by decoupling execution of a request from committing the request. Consensus protocols like Multi-Paxos and Raft wait until the request is committed at a quorum of servers before executing the request at one of them (typically the leader). However, the leader in Nezha executes the request before it is committed and sends the execution result to the client. The client then accepts the leader’s execution result only if it also gets a quorum of replies from other servers that indicate commitment; otherwise, the client just retries the request. Thus a leader’s execution is speculative in that the execution result might not actually be accepted by a client because (1) the leader was deposed after sending its execution result and (2) the new leader executed a different request instead.

**Proxy for deployability.** Performing quorum checks, multicasting, and clock synchronization at the client creates additional work for a Nezha client relative to a typical client of a protocol like Multi-Paxos or Raft. To address this, Nezha uses a proxy (or a fleet of proxies if higher throughput is needed), which multicasts requests, checks the quorum sizes, and performs clock synchronization—on the client’s behalf. Because Nezha’s proxy is stateless, it is easy to scale with the number of clients and it is easier to make fault tolerant.

**Evaluation.** We compare Nezha to six baselines in the public cloud: Multi-Paxos, Fast Paxos, our optimized version of NOPaxos, Raft, Domino and TOQ-EPaxos under closed-loop and open-loop workloads. In a closed-loop workload, commonly used in the literature [41, 52, 56, 63], 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 [73], clients submit new requests according to a Poisson process, without waiting for replies for previous ones. We find:

1. In closed-loop tests, Nezha (with proxies) outperforms all the baselines by 1.9–20.9× in throughput, and by 1.3–4.0× in latency at close to their saturation throughputs.
2. In open-loop tests, Nezha (with proxies) outperforms all the baselines by 2.5–9.0× in throughput. It also outperforms five baselines by 1.3–3.8× in latency at close to their saturation throughputs. The only exception is that, it sacrifices 33% of latency compared with our optimized version of NOPaxos.
3. Nezha can achieve better latency without a proxy, if clients perform multicasts and quorum checks. In open-loop tests, Nezha (without proxies) outperforms all the baselines by 1.3–6.5× in latency at close to their respective saturation throughputs. In closed-loop tests, Nezha (without proxies) outperforms them by 1.5–6.1×.
4. We also use Nezha to replicate two applications (Redis and a prototype financial exchange, CloudEx [21]) 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://github.com/Steamgik/Nezha.

### 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 [20], 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 [37]/Raft [58], Fast Paxos [36], Speculative Paxos [63], and NOPaxos [41]. Table 1 summarizes the basic properties of them and Nezha. §11 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 [63] or the OUM primitive [41]), Fast Paxos performs much worse (§9.2) because most requests can only be committed in the slow path, but its slow path (2.5 RTTs) is slower than Multi-Paxos/Raft and causes even more work for its leader because the leader needs to reconcile the inconsistent sequences of requests among followers.

**High-performance but needs in-network functionality.** Speculative Paxos [63] and NOPaxos [41] 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.
Table 1: Basic Properties of Typical Consensus Protocols (F: fast path; S: slow path)

| Protocols             | Message Delays | Load on Leader | 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 + [f/2] + 1 | S: f + 1    | F/S: Leader Medium    | No special requirement |
| Speculative Paxos     | F:2 S:6        | 2              | F: f + [f/2] + 1 | S: f + 1    | F: Client S: Leader   | High (rollback) Priority scheduling, SDN control, etc |
| NO Paxos              | F: 2 or 3 S:4 or 5 | 2              | F: f + 1 | S: f + 1 | F: Client S: Leader   | Medium Priority scheduling, programmable switch, SDN control, etc |
| Mencius               | F:4 S:6        | 2(2f + 1)/l    | F: f + 1 | S: f + 1 | F: Leaders S: Leaders | Medium No special requirement |
| EPaxos                | F:4 S:6        | 2(2f + 1)/l    | F: f + 1 | S: f + 1 | F: Leaders S: Leaders | Medium No special requirement |
| CURP                  | F:2 S:6        | 2f + 2         | F: f + 1 | S: f + 1 | F: Client S: Leader   | Low NVM, a non-faulty configuration manager |
| Nezha (No proxy)      | F: 2 S: 3      | 2 + 2f/m      | F: f + [f/2] + 1 | S: f + 1    | F/S: Client S: Proxy | Low Clock synchronization |
| (With proxy)          | F: 4 S: 5      |                |             |             |                       |                        |

1 We illustrate the number of message delays for each protocol in Appendix J.
2 Load on Leader indicates the total number of messages the leader needs to send or receive in order to commit one client request.
3 When NO Paxos 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.
4 Programmable switch serves as the hardware sequencer, and it is unnecessary when NO Paxos uses software sequencer. However, software sequencers can reduce throughput.
5 Mencius and EPaxos use multiple leaders to amortize the load. Given the number of leaders as l, the load on one leader becomes 2(2f + 1)/l.
6 Here, we consider EPaxos in a single datacenter environment. When EPaxos is deployed in WAN environments, and clients are co-located with some replica, the 4 message delays of EPaxos include 2 LAN message delays and 2 WAN message delays.
7 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 (m is the batch size).

Speculative Paxos achieves this by (1) controlling routing to ensure the same path length for all 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 submitted via the slow path (3 RTTs); the slow path also requires application-specific rollback. Further, Speculative Paxos is very sensitive to packet reordering. Its throughput drops by 10× with a 1% reordering rate [63] [Figure 9]; such rates can easily occur in cooperate to present a fault-tolerant service to clients. In the RSM setting, the goal of consensus protocols is to get multiple servers/replicas to reach agreement on the contents of an ordered log, which represents a sequence of operations issued to the RSM. This amounts to 2 requirements, one for the order of the log and one for the contents of the log. We state these 2 requirements below.

For any two replicas R1 and R2:
- **Consistent ordering.** If R1 processes request a before request b, then R2 should also process request a before request b, if R2 received both a and b.
- **Set equality.** If R1 processes request a, then R2 also processes request a.

Many optimistic protocols leverage the fact that the ordering of messages from clients to replicas is usually consistent at different locations: they employ a fast path during times of consistent ordering and fall back to a slow path when ordering is not consistent [36, 41, 63, 83]. However, for an optimistic protocol to actually improve performance, the fast path should indeed be the common case. If not, such protocols can potentially hurt performance [36, 63] relative to a protocol that doesn’t optimize for the common case like Raft or Multi-Paxos.

Consistent ordering is violated if messages arrive in different orders at different receivers. This situation is especially common in the public cloud where there is frequent reordering: messages from one or more senders to different receivers take different network paths and arrive in different orders at the receivers.
We measure the reordering with a simple experiment on Google Cloud. We use two receiver VMs, denoted as $R_1$ and $R_2$. We use a variable number of sender VMs to multicast messages to $R_1$ and $R_2$. We vary the rate of a Poisson process used by each sender to generate multicast messages (Figure 1) or vary the number of multicasting senders (Figure 2). After the experiment, $R_1$ receives a sequence of messages, which serves as the ground truth: each message is assigned a sequence number based on its arrival order at $R_1$. Based on these sequence numbers, we calculate a metric called reordering score to check how reordered $R_2$ is. We calculate the length of the longest increasing subsequence (LIS) [33, 68] in $R_2$’s sequence, and the reordering score is calculated as:

$$\text{reordering score} = (1 - \frac{\text{Length of } R_2's \text{ LIS}}{\text{Total Length of } R_2's \text{ Sequence}}) \times 100\%$$

A higher reordering score indicates more reordering occurring in the public cloud. Figure 1 shows that when we vary the submission rate, keeping the number of senders fixed at 2, the reordering score quickly exceeds 28%. Further in Figure 2, when we vary the number of senders, keeping the submission rate fixed at 10K messages/second, the reordering score increases rapidly up to 43% with the number of senders, because more senders lead to more concurrency in request submission but the network of public cloud has no way to serialize the concurrent requests into a commonly-agreed sequence.

In the public cloud, with such high reordering rates, optimistic protocols are forced to take the slow path often, which reduces their performance (§9.2). In order to design a high-performance protocol, we need to reduce the rate of reordering. This motivates us to design the deadline-ordered multicast (DOM) primitive to guarantee consistent ordering among replicas. DOM does not guarantee set equality. This is intentional and is also why we need a consensus protocol, Nezha, to go along with DOM because guaranteeing both requirements has been shown to be as hard as consensus itself [8].

### 4 DEADLINE-ORDERED MULTICAST

Informally, deadline-Ordered Multicast (DOM) is designed to reduce the rate of reordering by (1) waiting to process a message at a receiver until the message’s deadline is reached and (2) delivering messages to the receiver in deadline order. This gives other messages with a lower deadline the ability to “catch up” and reach the receiver before a message with a later deadline is processed.

Formally, in DOM, a sender wishes to send a message $M$ to multiple receivers $R_1$, $R_2$, ..., $R_n$. The sender attaches a deadline $D(M)$ to the message, where $D(M)$ is specified in a global time that is shared by senders and receivers because their clocks are synchronized. Then the DOM primitive attempts to deliver $M$ to receivers within $D(M)$. Receivers (1) can only process $M$ on or after $D(M)$ and (2) must process messages in the order of $D(M)$’s regardless of $M$’s sender.

We stress that DOM is a best-effort primitive: a sequence of messages is processed in order at a receiver if they all arrive before their deadline, but DOM does not guarantee that messages arrive reliably at all receivers either before the deadline or at all. There are two situations that cause DOM messages to arrive late or be lost.

The first is network variability: messages may not reach some receivers or reach them so late that other messages with larger deadlines have been processed. The second is a temporary loss of clock synchronization. If clocks are poorly synchronized, the deadline on a message might be set much earlier in time than the actual time at which the receiver receives the message.

While DOM is a general primitive, we comment briefly on its specific use for consensus as in Nezha. When DOM is used for consensus, because DOM makes no guarantees on reliable or timely delivery, it is up to the slow path of the consensus protocol to handle lost or late messages. If client requests are lost because of drops in the network and haven’t been received by a quorum of replicas, it is up to clients to retry the requests. These weaker guarantees in DOM are important because providing both reliable delivery and ordering of multicast messages is just as hard as solving consensus [8]. The use of clock synchronization for performance (i.e., increasing the frequency of the fast path) rather than correctness (i.e., linearizability) is also in line with Liskov’s suggestion on how synchronized clocks should be used [43].

**Setting DOM deadlines.** Setting deadlines is a trade-off between avoiding message reordering and adding too much waiting time to a message before it can be processed. In the public cloud, where VM-to-VM latencies can be variable and reordering is common, these deadlines should be set adaptively based on recent measurements of one-way delays (OWDs), which are also enabled by clock synchronization. We pick the deadline for a message by taking the maximum among the estimated OWDs from all receivers and adding it to the sending time of the message. The estimation of OWD is formalized as below:

$$\overline{OWD} = \begin{cases} \beta(\sigma_S + \sigma_R), & 0 < \overline{OWD} < W \\ \frac{p}{W} & \end{cases}$$
To track the varying OWDs, each receiver maintains a sliding window for each sender, and records the OWD samples by subtracting the message’s sending time from its receiving time. Then the receiver picks a percentile value from the samples in the window as \( P \). 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 use percentiles for robust estimation. The percentile is a DOM parameter set by the user of DOM.

Besides \( P \), DOM also obtains from the clock synchronization algorithm, Huygens [20] the standard deviation for the sending time and receiving time, denoted as \( \sigma_S \) and \( \sigma_R \). \(^3\) \( \sigma_S \) and \( \sigma_R \) provide an approximate error bound for the synchronized clock time, so we add the error bound with a factor \( \beta \) to \( P \) 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. Moreover, in case that clock synchronization goes wrong and provides invalid OWD values (i.e., very large or even negative OWDs), we further adopt a clamping operation: If the estimated OWD goes out of a predefined scope \([0, W]\), we will use \( W \) as the estimated OWD. The estimated OWD will be sent back to the sender to decide the deadlines of subsequent requests.

To illustrate DOM’s benefits, we redo our experiments from §3 with 10 Poisson senders, each submitting 10K requests/sec to 2 receivers. Figure 3 shows different percentiles (i.e., 50th, 75th, 90th, and 95th) for DOM to decide its deadlines. We can see that a higher percentile leads to more reduction of reordering. However, a higher percentile also means a longer holding delay for messages in DOM, which in turn undermines the latency benefit of Nezha protocol.

### 5 Nezha Overview

We use DOM as a building block to develop a consensus protocol, called Nezha, atop DOM. We overview the protocol here and describe it in detail in subsequent sections. Recall that DOM maintains consistent ordering across replicas by ordering messages based on their deadlines. This allows Nezha to use a fast path that assumes consistent ordering across replicas. When DOM fails to deliver a message to enough replicas before the message’s deadline (either because of delays or drops), Nezha uses a slow path instead.

**Model and assumptions.** Nezha assumes a fail-stop model and does not handle Byzantine failures. It uses \( 2f + 1 \) replicas: 1 leader and \( 2f \) followers, where at most \( f \) can be faulty and crash. Nezha guarantees safety (linearity) at all times and liveness under the same assumptions as Multi-Paxos/Raft (i.e., “the majority of servers are up and communicating with reasonable timeliness” [60]).

However, Nezha’s performance is improved by DOM, whose effectiveness depends on accurate clock synchronization among VMs and the variance of OWDs between proxies and replicas. Here “accurate” means the clocks among proxies and replicas are synchronized with a small error bound in most cases. But Nezha does not assume the existence of a worst-case clock error bound that is never violated because clock synchronization can also fail [43, 45, 46].

**Nezha Architecture.** Nezha uses a stateless proxy/proxies (Figures 4 and 5) interposed between clients and replicas to relieve clients’ computational burden of quorum checks and multicasts. Using a stateless proxy also makes Nezha a drop-in replacement for Raft/Multi-Paxos because the client just communicates with a Nezha proxy like it would with a Raft leader. This proxy serves as the DOM sender, while the replicas serve as DOM receivers. The DOM deadline is set to be the sum of (1) the request sending time from the proxy and (2) the maximum of a sliding window median (50th percentile) of OWD estimates between the proxy and each replica; these deadlines also take into account the current estimate of clock synchronization errors (§4). Another benefit of a proxy is that it is sufficient if the proxy’s clock is synchronized with the receivers; the client can remain unsynchronized.

**Nezha Fast Path.** We very briefly describe Nezha’s fast path and slow path, leaving details to later sections. Figure 4 shows the fast path. The request is multicast from the proxy \( @ \). If the request’s deadline is larger than the last request released from the early-buffer, the request enters the early-buffer \( @ \). It will be released from the early-buffers at the deadline, so that replicas can append the request to their logs \( @ \). The log of requests is ordered by request deadline. After that, followers immediately send a reply to the proxy without executing the request \( @ \), whereas the leader first executes the request \( @ \) and sends a reply including the execution result. The proxy considers the request as committed after receiving replies from the leader and \( f + \lfloor f/2 \rfloor \) followers. The proxy also obtains the execution result from the leader’s reply, and then replies with the execution result to its client. The fast path requires a super quorum \((f + \lfloor f/2 \rfloor + 1)\) rather than a simple quorum \((f + 1)\) for the same reason as Fast Paxos [36]: Without leader-follower communication, a simple quorum cannot persist sufficient information for a new leader to always distinguish committed requests from uncommitted requests (details in §6.3).

**Nezha Slow Path.** Figure 5 shows the more involved slow path: when a multicasted \( @ \) request goes to the late-buffer because of its small deadline \( @ \), followers do not handle it. However, the leader must pick it out of its late-buffer eventually for liveness. So the leader modifies the request’s deadline to make it eligible to enter the early-buffer \( @ \). After releasing and appending this request to the log \( @ \), the leader broadcasts this request’s unique identifier (a 3-tuple consisting of client-id, request-id, and request deadline) to followers \( @ \), to force followers to keep consistent logs with the leader. On hearing this broadcast \( @ \), the followers add/modify entries from their log to stay consistent with the leader: as an optimization, followers can retrieve missing requests from their late-buffers without having to ask the leader for these entries \( @ \). After this, followers send replies to the proxy \( @ \). Meanwhile, the leader has executed the request \( @ \) and replied to the proxy \( @ \). After collecting \( f + 1 \) replies (including the leader’s reply), the proxy considers the request as committed. Notably, Nezha differs from other optimistic protocols (e.g. [36, 41, 63]): in the slow path, to improve performance, Nezha decouples request execution (at the leader) from commit. The request is immediately executed by the leader and included in the leader’s reply. The request is only committed by the proxy if a quorum exists, i.e., if the leader’s speculation is safe.

\(^3\) \( \sigma \) values are calculated based on the method in [19][Appendix A], \( \beta = 3 \) in our setting, i.e., a 3\( \sigma \) confidence interval for the sending/receiving time provided by Huygens.
6 The Nezha Protocol

We first describe the state maintained by Nezha, Nezha’s message formats and Nezha’s fast and slow path. In addition, Figure 6 illustrate the life cycle of a request after arriving at a replica. Algorithms 1 and 2 describe the replica and proxy algorithms with object-oriented and event-driven methods.

6.1 Replica State

Figure 7 summarizes the state variables maintained by each replica. We omit some variables (e.g., crash-vector) related to Nezha’s recovery (§7). Below we describe them in detail.

replica-id: Each replica is assigned with a unique replica-id, ranging from 0 to 2f. The replica-id is provided to the replica during the initial launch of the replica, and is then persisted to stable storage, so that the replica can get its replica-id after crash and relaunch.

view-id: Replicas leverage a view-based approach [44]: each view is indicated by a view-id, which is initialized to 0 and incremented by one after every view change. Given a view-id, this view’s leader’s replica-id is view-id% (2f + 1).

Figure 4: Fast path of Nezha

Figure 5: Slow path of Nezha

Figure 6: Life cycle of one request on the replica

status: Replicas switch between three different statuses: NORMAL, VIEWCHANGE, and RECOVERING. Replicas are initially launched in
Algorithm 1 Replica Actions

Member Variables:

\(eb\), \(lb\), \(eb\), \(lb\) 

\(synced-log\), \(unsynced-log\), \(committed-log\), \(request\), \(status\), \(f\) 

1. on receive request \(do\)
   2. \(eb\) insert \(request\) \(\triangleright eb\) is a priority queue
   3. if \(request\) , then \(return\) 
   4. \(eb\) insert \(request\) \(\triangleright eb\) is a priority queue
   5. 
   6. \(lb\) insert \(request\) \(\triangleright lb\) is a map
   7. if \(request\) then \(return\) 
   8. \(newDdl = max(cloctime(), lastReq.deadline + 1)\) 
   9. \(request\) deadline = \(newDdl\) \(\triangleright Modify its deadline\)
   10. \(eb\) insert \(request\) ) \(\triangleright Can enter eb with the new deadline\)
   11. upon \(eb\) empty() = false and \(eb\) top(). \(\triangleright\) The request to be released from \(eb\)
   12. \(eb\) erase \(request\) \(\triangleright\) Replica is leader
   13. \(result = execute(request)\) \(\triangleright\) Only leader executes request
   14. if \(request\) then \(return\) 
   15. \(synced-hash = calciincrementhash(synced-log)\) 
   16. \(unsynced-hash = calciincrementhash(unsynced-log)\) 
   17. \(sendFastReply(result, hash)\) \(\triangleright\) In parallel with sending fast-reply, leader conducts broadcast
   18. \(broadcastLogModification(request)\) 
   19. 
   20. if \(request\) then \(return\) 
   21. \(synced-hash = calciincrementhash(synced-log)\) 
   22. \(unsynced-hash = calciincrementhash(unsynced-log)\) 
   23. \(sendFastReply\) (null, synced-hash, XOR unsynced-hash) \(\triangleright\) Only followers will receive log-modification messages
   24. upon \(log-modification\) do
   25. \(\triangleright Both synced part and unsynced part will be modified, details at §6.4\)
   26. \(ModifySynced\) (synced-log, synced-hash) 
   27. \(ModifyUnSynced\) (unsynced-log, unsynced-hash) 
   28. \(sendSlowReply()\) \(\triangleright\) Only followers send slow-reply

normal status. When the leader is suspected of failure, followers switch from normal to view-change and initiate the view change process. They will switch back to normal after completing the view change. For a failed replica to rejoin the system, it starts from recovering status and will switch to normal after recovering its state from the other replicas.

early-buffer: early-buffer is implemented as a priority queue, sorted by requests’ deadlines. early-buffer is responsible for (1) conducting eligibility checks of incoming requests: if the incoming request’s deadline is larger than the last released one from early-buffer, then the incoming request can enter the early-buffer; and (2) release its accepted requests in the order of their deadlines, thus maintaining DOM’s consistent ordering across replicas.

late-buffer: late-buffer is implemented as a map using the <client-id, request-id> as the key. It is used to hold those requests which are not eligible to enter the early-buffer. Replicas maintain

\(^{3}\)When deadlines are equal, the tie is broken by <client-id, request-id>.

Algorithm 2 Proxy Actions

Member Variables:

\(f\), \(reply\), \(reply\) 

1. upon receive request from client \(do\)
   2. \(tag request with the sending time and the deadline\)
   3. for \(r \leftarrow 0 to 2f\) do
   4. \(send request to replica r\) 
   5. upon receive reply from replica \(do\)
   6. if reply is duplicate or from previous view then \(return\)
   7. if reply is from new view then \(\triangleright Replica experienced view change, all previous replies are stale\)
   8. \(replySet.clear()\)
   9. replySet.insert(reply)
   10. committedReply = checkCommitted(reply) 
   11. if committedReply \(\neq null\) then 
   12. \(replyToClient(committedReply)\)
   13. function checkCommitted (reply) \(\triangleright\) If the proper quorum is established, return the leader’s reply because it contains the execution result
   14. \(\triangleright\) If the proper quorum is established, return the leader’s reply because it contains the execution result
   15. \(\triangleright\) If the proper quorum is established, return the leader’s reply because it contains the execution result
   16. \(\triangleright\) If the proper quorum is established, return the leader’s reply because it contains the execution result
   17. \(\triangleright\) If the proper quorum is established, return the leader’s reply because it contains the execution result
   18. \(\triangleright\) If the proper quorum is established, return the leader’s reply because it contains the execution result
   19. \(\triangleright\) If the proper quorum is established, return the leader’s reply because it contains the execution result
   20. \(\triangleright\) If the proper quorum is established, return the leader’s reply because it contains the execution result
   21. \(\triangleright\) If the proper quorum is established, return the leader’s reply because it contains the execution result
   22. \(\triangleright\) If the proper quorum is established, return the leader’s reply because it contains the execution result
   23. \(\triangleright\) If the proper quorum is established, return the leader’s reply because it contains the execution result
   24. \(\triangleright\) If the proper quorum is established, return the leader’s reply because it contains the execution result
   25. \(\triangleright\) If the proper quorum is established, return the leader’s reply because it contains the execution result
   26. \(\triangleright\) If the proper quorum is established, return the leader’s reply because it contains the execution result
   27. \(\triangleright\) If the proper quorum is established, return the leader’s reply because it contains the execution result
   28. \(\triangleright\) If the proper quorum is established, return the leader’s reply because it contains the execution result
   29. \(\triangleright\) If the proper quorum is established, return the leader’s reply because it contains the execution result

such a buffer because those requests may later be needed in the slow path (§6.4). In that case, replicas can directly fetch those requests locally instead of asking remote replicas.

log: Requests released from the early-buffer will be appended to the log of replicas. The requests then become the entries in the log. The log is ordered by request deadline.

sync-point: Followers modify their logs to stay consistent with the leader (§6.4). sync-point indicates the log position up to which this replica’s log is consistent with the leader. Specially, the leader always advances its sync-point after appending a request.

commit-point: Requests (log entries) up to commit-point are considered as committed/stable, so that every replica can execute requests up to commit-point and checkpoint its state up to this position. commit-point is used in an optional optimization (§8.3).

6.2 Message Formats

There are five types of messages closely related to Nezha. We explain their formats below. Since Nezha uses a view-based approach for leader change, we omit the description of messages
- replica-id—replica identifier (0, 1, · · · , 2f).
- view-id— the view identifier, initialized as 0 and increased by 1 after every view change.
- status— one of normal, viewchange, or recovering.
- 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, which is searchable by <client-id, request-id> of the request.
- log— a list of requests, which are appended in the order of their deadlines.
- sync-point— the log position indicating this replica’s log is consistent with the leader up to this point.
- commit-point— the log position indicating the replica has checkpointed the state up to this point.

**Figure 7: Local state of Nezha replicas**

related to leader changes; these messages have been defined in Viewstamped Replication [44].

**request:** request is generated by the client and submitted to the proxy. The proxy will attach some necessary attributes and then submit request to replicas. request is represented as a 5-tuple:

request=<client-id, request-id, command, s, d>

client-id represents the client identifier and request-id is assigned by the client to uniquely identify its own request. On one replica, client-id and request-id combine to uniquely identify the request. command represents the content of the request, which will be executed by the leader. s and d are tagged by proxies. s is the sending time of the request and d is the estimated deadline that the request is expected to arrive at all replicas. When the request arrives at the replica, the replica can also derive the proxy-replica OWD by subtracting s from its receiving time.

**fast-reply:** fast-reply is sent by every replica after they have appended or executed the request, and it is used for quorum checks in the fast path. fast-reply is represented as a 6-tuple:

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

view-id and replica-id are from the replica state variables (see §6.1). client-id and request-id are from the appended request that lead to this reply. result is only valid in the leader’s fast-reply, and is null in followers’ fast-replies. The proxy can recognize the leader’s reply by checking whether its replica-id is view-id%(2f + 1). hash captures a hash of the replica’s log (the hash calculation is explained in §8.1). Proxies can check the hash values to know whether the replicas involved in a quorum have consistent logs.

**log-modification:** log-modification message is broadcast by the leader to convey the log entry’s deadline, client-id and request-id to followers, making the followers modify their logs to keep consistent with the leader. Meanwhile, log-modification also doubles as the leader’s heartbeat. log-modification is represented as a 5-tuple:

log-modification=<view-id, log-id, client-id, request-id, deadline>

view-id is from the replica state. log-id indicates the position of this log entry (request) in the leader’s log. client-id and request-id uniquely identify the request. deadline is the request’s deadline shown in the leader’s log, which is either assigned by proxies on the fast path (i.e., § in Figure 4) or overwritten by the leader on the slow path (i.e., § in Figure 5). log-modification messages can be batched under high throughput to reduce the leader’s burden of broadcast.

**slow-reply:** slow-reply is sent by followers after all the entries in their logs have become the same as the leader’s log entries up to this request. It is used by the client to establish the quorum in the slow path. slow-reply is represented as a 4-tuple:

slow-reply = <view-id, replica-id, client-id, request-id>

The four fields have the same meaning as in the fast-reply.

**log-status:** log-status is periodically sent from the followers to the leader, reporting the sync-point of the follower’s log, so that the leader can know which requests have been committed and update its commit-point. log-status is represented as a 3-tuple:

log-status=<view-id, replica-id, sync-point>

The three fields come from the followers’ replica state variables.

### 6.3 Fast Path

Nezha relies on DOM to increase the frequency of its fast path. As shown earlier, the percentile at which DOM estimates OWDs is a parameter set by the DOM user. A lower percentile will set smaller deadlines, which improve fast path latency, but reduce the frequency of the fast path. Higher percentiles have the opposite problem. For Nezha, we use the 50th percentile to strike a balance between the two. This does reduce the fast path frequency compared with using a higher percentile; hence, Nezha compensates for this by optimizing its slow path for low client latency as well.

To commit the request in the fast path (Figure 4), the proxy needs to get the fast-reply messages from both the leader and f + [f/2] followers. (1) It must include the leader’s fast-reply because only the leader’s reply contains the execution result. (2) It also requires the f + [f/2] + 1 replicas have matching view-ids and the same log (requests). In §8.1 we will show how to efficiently conduct the quorum check by using the hash field included in fast-reply. If both (1) and (2) are satisfied, the proxy can commit the request in 1 RTT.

As briefly explained in the sketch of the fast path (§5), the fast path requires a super quorum (f + [f/2] + 1) rather than a simple quorum (f + 1), because a simple quorum is insufficient to guarantee the correctness of Nezha’s fast path. Consider what would happen if we had used a simple majority (f + 1) in the fast path. Suppose there are two requests request-1 and request-2, and request-1 has a larger deadline. request-1 is accepted by the leader and f followers. They send fast-replies to the proxy, and then the proxy considers request-1 as committed and delivers the execution result to the client application. Meanwhile, request-2 is accepted by the other f followers. After that, the leader fails, leaving f followers with request-1 accepted and the other f followers with request-2 accepted. Now, the new leader cannot tell which of request-1 or request-2 is committed at a provided log position. If the new leader adds request-2 into the recovered log, it will be appended and executed ahead of request-1 due to request-2’s smaller deadline. This violates linearizability [25]: the client sees request-1 executed before request-2 with the old leader but sees the reverse with the new leader.
6.4 Slow Path

The proxy is not always able to establish a super quorum to commit the request in the fast path. When requests are dropped or are placed into the late-buffers on some replicas, there will not be sufficient replicas sending fast replies. Thus, we need the slow path to resolve the inconsistency among replicas and commit the request. We explain the details of the slow path (Figure 5) below in temporal order starting with the request arriving at the leader.

Leader processes request. After the leader receives a request, the leader ensures it can enter the early-buffer: if it is not eligible due to its small deadline Ω, the leader will modify its deadline to make it eligible. Specifically, the modified deadline should be larger than the last released request’s deadline. Therefore, we choose the max between (a) the replica’s current clock time and (b) the last released request’s deadline + 1 μs. The leader then conducts the same operations as in the fast path (i.e., appending the request Ω, applying it to the state machine Ω, and sending fast-reply Ω).

Leader broadcasts log-modification. In parallel with Ω-Ω, the leader also broadcasts a log-modification message to followers Ω after appending each request. Every time a follower receives a log-modification message Ω, it checks its log entry at the position log-id included in the log-modification message. (1) If the entry has the same 3-tuple <client-id, request-id, deadline> as that included in the log-modification message, it means the follower has the same log entry as the leader at this position. (2) If only the 2-tuple <client-id, request-id> is matched with that in the log-modification 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 log-modification message. (3) Otherwise, the entry has different <client-id, request-id>, which means the follower has placed a wrong entry at this position. In this third 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 <client-id, request-id>. As a rare case, when the entry does not exist on this replica because the request was dropped or delayed, the follower fetches it from other replicas and puts it at the position.

Follower sends slow-reply. After the follower has processed the log-modification message, and has ensured the requests in its log are the same as the leader’s log entries, the follower updates its sync-point, indicating it has the same log entries as the leader up to the log position represented by the sync-point. The leader itself can directly advance its sync-point after appending the request to log. Then, the follower sends a slow-reply message for every synced request Ω. The slow-reply will be used to establish the quorum in the slow path. 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 log is consistent with the leader. By contrast, the follower’s fast-reply cannot replace its slow-reply for the quorum check in the slow path.

Proxy conducts quorum check. The proxy considers the request as committed when it receives the fast-reply from the leader and the slow-replies from f followers. The execution result is obtained from the leader’s fast-reply. Decoupling (leader’s) execution from commit enables the proxy to know whether the request is committed even earlier than the leader replica. Meanwhile, replicas can continue to process subsequent requests and are not blocked by the quorum check in the slow path, which proves to be an advantage compared to other opportunistic protocols like NOPaxos (see §9.2). Unlike the quorum check of the fast path (§6.3), the slow path does not need a super quorum (1 + f + ⌈f/2⌉). 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 (§7), they are directly copied to the new leader’s log.

In the background: followers report sync-statuses. In response to log-modification messages, followers send back log-status messages to the leader to report their sync-points. The leader can know which requests have been committed by collecting the sync-points from f + 1 replicas including itself: the requests up to the smallest sync-point among the f + 1 ones are definitely committed. Therefore, the leader can update is commit-point and checkpoint its state at the commit-point. It can also broadcast the commit-point to followers, which enables followers to checkpoint their states for acceleration of recovery (§8.3). Note that the followers’ reporting sync-status is not on the critical part of the client’s latency on the slow path; it happens in the background. Therefore, the slow path only needs three message delays (1.5 RTTs) for the proxy to commit the request. Besides, the log-status messages only serve for an optional optimization to accelerate recovery (details in §8.3). The correctness and performance of Nezha will not be affected even if all log-status messages are lost.

6.5 Other Concerns

Proxy Failure. Proxy failures do not hurt Nezha’s correctness: proxy failures cause the same effect as packet drops, which is already handled by consensus protocols because consensus protocols do not assume reliable communication [9, 35].

Client Timeout and Retry. The client starts a timer while waiting for the proxy’s reply. If the timeout is triggered (due to packet drop or proxy failure), the client eventually 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. As in traditional distributed systems, replicas maintain at-most-once semantics. When receiving a request with duplicate <client-id, request-id>, the replica resends the previous reply without re-execution.

7 RECOVERY

Assumptions. We assume replica processes can fail because of process crashes or a reboot of the replica’s 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
Avoiding disk writes during normal processing. While designing the recovery protocol, we aim to avoid disk writes as much as possible. This is because disk writes add significant delays (0.5ms~20ms per write), significantly increasing client latency. At the same time, we also want to preserve the correctness of our protocol from stray messages [34], which have caused bugs to multiple diskless protocols (e.g., [44][63][41][83]). Nezha adopts the crash-vector technique invented by Michael et al. [49, 50], to develop Nezha’s recovery protocol. While Nezha still uses stable storage to distinguish whether it is the first launch or reboot, it does not use disk writes during normal processing and preserves its correctness from the stray message effect. In Appendix §A we describe how to use crash-vector to prevent stray-messages for Nezha.

Correctness. In Appendix §B, we have proved Nezha’s three correctness properties. The three properties have also been model-checked in our TLA+ specification (Appendix §I).

- 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).

8 OPTIMIZATIONS IN NEZHA

8.1 Incremental Hash

In Nezha’s fast path, fast-replies from replicas can form a super quorum only if these replies indicate that the replicas’ ordered logs are identical. This is because—unlike the slow path—replicas do not communicate amongst themselves first before replying to the client. One impractical way to check that the ordered logs are identical is to ship the logs back with the reply. A better approach is to perform a hash over the sequence corresponding to the ordered log, and update the hash every time the log grows. However, if the log is ever modified in place (like we need to in the slow path), such an approach will require the hash to be recomputed from scratch, starting from the first log entry.

Instead, we use a more efficient approach by decomposing the equality check of two ordered logs into two components: checking the contents of the 2 logs and checking the order of the 2 logs. Because logs are always ordered by deadline at all our replicas, it suffices for us to check the contents of the 2 logs. The contents of the logs can be checked by checking equality of the 2 sets corresponding to the entries of the 2 logs: this requires only a hash over a set rather than a hash over a sequence.

To compute this hash over a set, we maintain a running hash value for the set. Every time an entry is added or removed from this set, we compute a hash of this entry (using SHA-1) and XOR the hash with the running hash value. This allows us to rapidly update the hash every time a log entry is appended (an addition to the set) or modified (a deletion followed by an addition to the set). The proxy checks for equality of this set hash across all replicas, knowing that equality of the set of log entries guarantees equality of the ordered logs because logs are always ordered by deadlines.

To be more specific, when the replica sends the fast-reply for its nth request, the hash, denoted as $H_n$, represents the set of all the stable storage, the replica notices it is the first launch and persists its replica-id to stable storage, and then it enters the system with normal status. Otherwise, the replica also fetches the same value of replica-id from the stable storage, then the replica knows this is a reboot, so it will first go through the recovery process with recovering status, and switch back to normal status after recovery.
previously appended entries using an incremental hash function [6]:

\[ H_n = \bigoplus_{1 \leq i \leq n} h(request_i) \]

Here, \( h(\cdot) \) 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 the stray message effect [49, 50], we also XOR \( H_n \) with the hash of the 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 (recall that crash-vector consists of \( 2f + 1 \) integers) and transforming it into a hash value. The inclusion of crash-vector is necessary for the correctness of Nezha and we explain this in Appendix §A.1.

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 whether they have the same requests.

### 8.2 Commutativity Optimization

To enable a high fast commit ratio without a long holding delay of DOM, we employ a commutativity optimization in Nezha. As an example, commutative requests refer to those requests operating on different keys in a key-value store, so that the execution order among them does not matter [11, 61]. The commutativity optimization enables us to choose a modest percentile (50\% percentile) while still achieving a high fast commit ratio, because it eases the fast path in two aspects.

First, it relaxes the eligibility check condition of the early-buffer. 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). Otherwise, DOM’s consistent ordering property is violated. However, the execution results of commutative requests are not affected by their order [61]. Hence, consistent ordering is only required among non-commutative requests, which enables the 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, it refines the hash computation, making hash consistency among replicas becomes easier. 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 also evaluate across a range of workloads in Appendix §C, with different read/write ratios and skew factors. The result shows that the commutativity optimization helps reduce the latency by 7.7\% - 28.9\%.

### 8.3 Periodic Checkpoints

To (1) accelerate the recovery process after leader failure and (2) enable a deposed leader to quickly catch up with the fresh state, we integrate a periodic checkpoint mechanism in Nezha.

Since Nezha only allows the leader to execute requests during normal processing, it can lead to inefficiency during leader change, either caused by leader’s failure or network partitions. This is because the new leader is elected from followers, and it has to execute all requests from scratch after it becomes the leader. To optimize this, we adopt an idea from NOPaxos [41] and conduct synchronization of the replica state 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. Both the leader and followers checkpoint state at their commit-points. The periodic checkpoints bring acceleration benefit in two aspects: (1) When the leader fails, the new leader only needs to recover and execute the requests from its commit-point onwards. (2) When network partitions happen, the leader is deposed and it later notices the existence of a higher view. Instead of abandoning its complete state (as what we described in §7), it can start from its latest checkpoint state to stable storage, and only retrieve from another replica (in the fresh view) requests beyond its commit-point.

### 9 EVALUATION

We answer the following questions during the evaluation:

1. How does Nezha compare to the baselines (Multi-Paxos, Fast Paxos, NOPaxos) in the public cloud? (§9.2)
2. How does Nezha compare to the recent protocols which also use clock synchronization (i.e., Domino and TOQ-EPaxos)? (§9.3)
3. How effective are the proxies in saving clients’ CPU cost, especially when there is a large number of replicas? (§9.7)
4. How fast can Nezha recover from the leader failure? (§9.9)
5. How does Nezha compare to Raft when both are equipped with log persistence to stable storage? (§9.10)
6. Does Nezha provide sufficient performance for replicated applications? (§10)

### 9.1 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 in a single cloud zone (except §9.8). Huygens is installed on all VMs and has an average 99\% percentile clock offset of 49.6 ns.

**Baselines.** We compare with Multi-Paxos, Fast Paxos and NOPaxos. For the 3 baselines, we use the implementation from the NOPaxos repository [40] with necessary modification: (1) we change switch multicast into multiple unicasts because switch multicast is
unavailable in cloud. (2) we use a software sequencer with multithreading for NOPaxos because tenant-programmable switches are not yet available in cloud. We also added two recently proposed protocols that leverage synchronized clocks for comparison, i.e., Domino [83] and TOQ-EPaxos [73]. We choose to compare them with Nezha because they also use clock synchronization to accelerate consensus. Domino is previously tested with clocks synchronized by Network Time Protocol (NTP) in [83], but in our test, we also provide Huygens synchronization for Domino to give it more favorable conditions because Huygens has higher accuracy than NTP [20]. However, since both Domino and TOQ-EPaxos target WAN settings, we do not expect them to perform better than Nezha or our other baselines in LAN settings, which is verified by our experiments (§9.2). We intended to compare with Derecho [29]. However, its performance degrades significantly in the public cloud due to a lack of RDMA (see Appendix §E). We think the comparison is not fair to Derecho and do not include it in the main body.

**Metrics.** We measure execution latency: the time between when a client submits a request to the system and receives an execution result from it along with a confirmation that the request is committed. We also measure throughput. To measure latency, we use median latency because it is more robust to heavy tails. We 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 [26, 35, 80] despite a median of only hundreds of microseconds. This is unlike the WAN setting where tails can be more reliably estimated [73]. We run each experiment 5 times and average values before plotting.

**Evaluation method.** We follow the method of NOPaxos [41] and run a null application with no execution logic. Traditional evaluation of consensus protocols [41, 52, 56, 57, 63, 76] 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 [73] 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 §9.2, we use 3 replicas. For the closed-loop test, we increase load by adding more clients until saturation. 6 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 [23] of 0.5. We also evaluate Nezha under various read ratios and skew factors in Appendix §C, which verifies the robustness of its performance.

### 9.2 Comparison with Multi-Paxos, Fast Paxos and NOPaxos

The closed-loop and open-loop evaluation results are shown in Figure 8. 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 the public cloud, in comparison with their published numbers from highly-engineered networks [41]. This is not surprising since the public cloud environment without network support is not the main focus of [41, 63]. Fast Paxos suffers the most and reaches only 4.0K requests/second at 425 µs in open-loop test (not shown in Figure 8b). When clients send at a higher rate, Fast Paxos suffers from heavy reordering, and the reordered requests force Fast Paxos into its slow path, which is even more costly than Multi-Paxos.

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 significant CPU cycles. We realized this issue and developed an optimized version (NOPaxos-Optim in Figure 8) 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. When the gap occurs, NOPaxos needs at least one RTT for the leader to coordinate with followers to fetch the missing request or mark no-op at the gap position. During the gap handling process, all the incoming requests have to be pending and can no longer be processed (i.e., the normal request processing logic is blocked). Thus, all these follow-up requests will make the gap handling cost as part of their latencies, and they can also continue to cause more gaps. Meanwhile, the system’s overall throughput is also degraded because no more requests are processed until the gap handling is completed.

Third, 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 improved throughput and latency come 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 separates request execution and quorum check, letting clients/proxies undertake quorum check instead of the leader, which effectively relieves leader’s burden and enables better pipelining (i.e., avoid the blocking problem in NOPaxos). (3) The use of commutativity further reduces the latency by allowing more requests to be committed in fast path. To verify the benefit of each component, we further conduct an ablation study in §9.4.

### 9.3 Comparison with Domino and TOQ-EPaxos

For a fair comparison of Domino and TOQ-EPaxos with Nezha, we originally wanted to plot their execution latencies in Figure 8. However, both Domino and TOQ-EPaxos decouple commit from execution, and execution happens much later than commit, which causes high execution latencies. In our experiments, we found

---

6Specially, when the system is saturated, the throughput can drop instead of continuously increasing [18, 77], as shown in Figure 8.
that Domino’s execution latency exceeds 10 ms and TOQ-EPaxos’ execution latency ranges from 1.3 to 3.3 ms, which are significantly larger than Nezha (as well as our other baselines). This makes it hard to show them in our figure, and hence we plot their commit latencies instead.

When comparing the commit latency of Domino and TOQ-EPaxos to Nezha’s execution latency, we still find that Nezha performs better. We believe this is because of differences in implementation: Domino and TOQ-EPaxos are implemented in Golang with gRPC [67, 82] whereas Nezha and the other baselines are implemented in C++ with UDP [40]). These differences in implementation likely arise from the fact that the additional latency incurred by Golang+gRPC is tolerable for wide-area use cases that typically have higher latencies. We also compare Nezha with Domino and TOQ-EPaxos below from a design perspective.

9.3.1 Nezha compared with Domino. Nezha and Domino both use synchronized clocks with deadlines attached to messages. They differ in 3 ways.

(1) Unlike Domino, Nezha does not decouple commit from execution, which makes it easier for Nezha to be a drop-in replacement for Paxos/Raft, where applications can directly get the execution result from the commit reply.

(2) Nezha’s correctness is independent of clock skew; whereas Domino can violate durability when the clock values are not monotonically increasing. When a request arrives later than its deadline, Domino replicas are expected to reject it. But if clock skew occurs at this moment, the replicas will accept it and make the client consider it as committed, even though this request may later be replaced by a no-op. We explain with error traces in Appendix §F.

(3) Nezha guarantees the linearizability, whereas Domino does not specify the consistency model it uses. We suspect Domino uses a weak consistency model, i.e., eventual consistency. However, the durability violation in (2) also causes the eventual consistency not guaranteed (details in Appendix §F).

9.3.2 Nezha compared with TOQ-EPaxos. (1) TOQ-EPaxos only synchronizes replicas to reduce reordering of messages between replicas, so that replicas are more likely to process non-commutative requests in the same order (refer to §4 in [73]). Nezha synchronizes replicas and proxies to reduce the reordering from proxies to replicas. Compared with TOQ-EPaxos, Nezha controls more paths in the consensus workflow, which makes Nezha’s acceleration more effective: when clients and replicas are located in different zones, TOQ provides little benefit, whereas Nezha can still reduce latency (§9.8).

![Figure 8: Latency vs. throughput](image-url)
Figure 9: Ablation study of Nezha

(2) TOQ does not guarantee consistent ordering but DOM does. In TOQ, when one replica multicasts the requests with a ProcessAt time, if some requests arrive at some replicas very late, then different replicas can still have different message orders. By contrast, DOM prioritizes consistent ordering over set equality, which means, any two replicas can never release the same requests in different order. DOM adopts such design because our Nezha protocol can rapidly fix set inequality based on its single leader design, unlike EPaxos which involves multiple leaders in its design.

(3) TOQ does not improve EPaxos performance in a LAN. As shown in [4], even implemented under the same framework, EPaxos is less performant than Multi-Paxos in LAN. By contrast, Nezha is a generally high-performance protocol which yields good performance in both LAN (Figure 8) and WAN (Figure 13) settings.

9.4 Ablation Study

During the ablation study of Nezha, we remove one component from the full protocol of Nezha each time to yield three variants, shown as No-DOM, No-QC-Offloading, No-Commutativity in Figure 9. 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 still perform request multicast). No-Commutativity variant disables Nezha’s commutativity optimization. We run all protocols under the same setting as Figure 8b.

Figure 9 shows that, removing any of the three components degrades performance (i.e., throughput and/or latency).

(1) The No-DOM variant makes the fast path meaningless: requests are no longer ordered by their deadlines without DOM, so consistent ordering is not 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 yield 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 a distinct impact on the throughput. However, compared with the full protocol, the lack of commutativity optimization degrades the latency performance by up to 24.2%.

9.5 DOM’s Trade-off at Different Percentiles

At the level of DOM primitive, the percentile used by DOM makes a trade-off between reordering rate and latency (Figure 3). When it comes to Nezha, the percentile makes a trade-off between how fast the request can be committed via the fast path and how frequently the request can be committed via the fast path. We measure these two aspects with fast path latency (FPL) and fast commit ratio (FCR) respectively. FPL is the median latency for requests to commit in fast path; FCR is the ratio of requests committed in fast path. A larger percentile leads to higher (better) FCR but longer (worse) FPL, but both FCR and FPL affect the overall commit latency (OCL) of all requests.

To measure this, in Figure 10 we run Nezha with/without commutativity optimization in the open-loop test (20K requests/second).

(1) Without commutativity optimization (Figure 10a), as we use larger percentiles (from 50p to 75p), the improvement of FPL outweighs the increase of FPL, thus leading to lower OCL. However, as we continue to use larger percentiles (e.g., 95p and 99p), though FCR keeps growing, FPL also becomes longer due to the increasing holding delay in early-buffer, which undermines the benefit of fast path, and no longer helps reduce OCL.

(2) After adding the commutativity optimization (Figure 10b), Nezha already reaches a high FCR using 50p. Therefore, using larger percentiles brings little FCR improvement, but only increases FPL, and further FPL. Therefore, we choose 50p in DOM and have verified its robustness under different workloads (details in [18]).

9.6 Scalability

Figure 11 shows that, Nezha achieves much higher throughput than the baselines with different number of replicas. However, in open-loop tests with only 10 clients (Figure 11b), the throughput of Nezha-Non-Proxy distinctly degrades from 187.8K requests/sec to 148.7K requests/sec, as the number of replicas grows. This indicates that the clients become the new bottleneck when submitting at
9.7 Proxy Evaluation

Figure 12a and Figure 12b compare the two versions of Nezha with 10 open-loop clients and 9 replicas. Nezha-Proxy employs 5 proxies. As clients increase their submission rates, 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 usage at the client side. It even achieves lower latency as the throughput grows, because Nezha-Non-Proxy makes the clients CPU-intensive.

Figure 12c compares the maximum throughput achieved by one client with/without proxies. Given the same CPU resource, the throughput of the client without proxies declines distinctly as the number of replicas increases. Such bottlenecks can also occur in the other protocols with similar offloading design (e.g., Speculative Paxos, NOPaxos, Domino, CURP). By contrast, when equipped with proxies, the client retains a high throughput regardless of the number of replicas.

9.8 Comparison in WAN

We continue to compare Nezha with the baselines in a wide-area network (WAN). We run the open-loop tests across 5 zones: the 3 replicas are located in europe-north1-a, asia-northeast1-a and southamerica-east1-a, respectively; the 10 open-loop clients are divided into two groups, and distributed in us-east1-b and us-west1-a; correspondingly, the 2 proxies are also distributed in us-east1-b and us-west1-a to serve the clients in their zones.

Compared to the LAN evaluation (Figure 8), Nezha outperforms all the baselines even more in WAN. As shown in Figure 13, NOPaxos-Optim is the best among the four baselines, but Nezha still outperforms NOPaxos-Optim by 1.51× in latency and 2.55× in throughput. For TOQ-EPaxos, TOQ provides little help to reduce.

---

1Each of the 9 replicas send a fast-reply to the client; each of the 8 followers also send a slow-reply to the client.
2Every client uses one thread for request submission and another for reply handling.
the latency for EPaxos when clients and replicas are located in different zones, because TOQ only leverages clock synchronization to reduce conflicts among replicas, and its fast path still costs 2 WAN RTTs when replicas and clients are separated. By contrast, since Nezha’s proxies are stateless and generally deployable, they can be deployed in the same zone as clients, making client-proxy latency as LAN message delay. Therefore, Nezha can achieve 1 WAN RTT in the fast path. We include more discussion in Appendix H.

9.9 Failure Recovery

We evaluate failure recovery as shown in Figure 14 and Figure 15. Since follower’s crash and recovery do not affect the availability of Nezha (until more than \( f \) failures), 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 use 3 replicas and 10 open-loop clients, and vary per-client submission rate from 1K requests/second to 20K requests/second, so the total submission rate varies from 10K requests/second to 200K requests/second. We are using a null application during our measurement. We leave for future work the evaluation of Nezha’s failure recovery performance for more complex replicated applications.

Under different submission rates, we kill the leader and measure the time taken for a view change. As shown in Figure 14, the time taken for a view change 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 ($8.3$) integrated in Nezha.

The time cost to recover the same throughput level (Figure 15) is larger than the time cost of view change, because there are other tasks to complete after the replicas enter the new view. For example, replicas need to refresh the working threads and reinitialize the contexts; replicas need to handle clients’ retried requests, which were not 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 15. Figure 15 shows that Nezha takes longer to recover to a higher throughput. 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 requests/sec, 100K requests/sec, 200K requests/sec, respectively. As a reference to compare, Figure 3.20 in [75] evaluates the recovery time for an industrial Raft implementation [24], which takes about 6 seconds to recover to 18K requests/sec.

9.10 Disk-based Comparison: Nezha vs. Raft

Raft establishes its correctness on log persistence and relies on 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 [58] (Raft-1 in Figure 16), which is written in C++, but uses a slower 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 requests/sec on Google Cloud VMs equipped with zonal standard persistent disk [22]. Hence, we implement and optimize Raft (Raft-2), by using the Multi-Paxos code from [40] 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 test (Figure 16) and open-loop test (Figure 17). We also see that there is little difference in latency with or without a proxy in Nezha because latencies are now dominated by disk writes, not message delays.

10 APPLICATION PERFORMANCE

In order to measure Nezha in the context of a replicated application, we port two applications to Nezha and the baseline protocols. Each replicated application uses 3 replicas.

Redis. Redis [64] is a typical in-memory key-value store. We choose YCSB-A [81] as the workload, which operates 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 18 illustrates the maximum throughput of each protocol under 10 ms service level objective (SLO). Nezha outperforms all the baselines on this metric: it outperforms Fast Paxos by 2.9×, Multi-Paxos by 1.9×, and NOPaxos by 1.3×. Its throughput is within 5.9% that of the unreplicated system.

CloudEx. CloudEx [21] is a research fair-access financial exchange system for the 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 improved the performance of CloudEx, compared with the version in [21], by multithreading and replacing ZMQ [84] with raw UDP transmission. We first run the unreplicated CloudEx with its dynamic delay bounds (DDP) strategy disabled [21]. We configure a fixed sequencer delay parameter \((d_s)\) of 200μs. Similar to [21], 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/second, achieving an inbound unfairness ratio of 1.49%.

We then run CloudEx atop the four protocols with the same setting. In Figure 19, only Nezha reaches the throughput (42.93K orders/second) 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 20, Nezha prolongs the end-to-end latency by 19.7% (344µs vs. 288µs), but achieves very close order processing latency to the unreplicated version (426µs vs. 407µs).

11 RELATED WORK

Consensus protocols. Classical consensus protocols, e.g., Multi-Paxos, Raft, and Viewstamped Replication make no distinction between a fast and slow path: all client requests incur the same latency. Nezha uses an optimistic approach to improve latency in the common case. Eve [32] adopts execute-verify architecture, which is similar to Nezha’s speculative execution design, but Eve requires application-specific rollback when replicas’ states diverge, whereas Nezha does not require such rollback mechanism. Menciùs [48] exploits a multi-leader design to mitigate the single-leader bottleneck in Multi-Paxos. However, it introduces extra coordination cost among multiple leaders and further, the crash of any of the leaders temporarily stops progress. By contrast, Nezha reduces the leader’s bottleneck using proxies and followers’ crash does not affect progress. EPaxos [56] can achieve optimal WAN latency in the fast path, but when it comes to the LAN scenarios where we focus on, it performs worse than Multi-Paxos [4]. CURP [61] can complete commutative requests in 1 RTT, but doesn’t take advantage of consistent ordering: hence, it costs up to 3 RTTs even if all witnesses process the non-commutative requests in the same order. SPaxos [7], BPaxos [79] and Compartmentalized Paxos [52] address the throughput scaling of consensus protocols with modularity, trading more latency for throughput improvement. The proxy design in Nezha is similar to compartmentalization [52], but Nezha’s proxies are stateless. By contrast, [7, 52, 79] use stateful proxies, which complicates fault tolerance.

Network primitives to improve consensus. Recent works consider building network primitives to accelerate consensus protocols. 4 other primitives are closely related to DOM, namely, mostly-ordered multicast (MOM) [63], ordered unreliable multicast (OUM) [41], timestamp-ordered queuing (TOQ) [73] and sequenced broadcast (SB) [70]. 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 environments like the cloud. On the other hand, requests output from MOM and
TOQ can still result in inconsistent ordering. By contrast, DOM and OUM guarantee consistent ordering of released requests. DOM’s guarantees are stronger than MOM because MOM can occasionally reorder requests, but are weaker than OUM because OUM also provides gap detection. We include a formal comparison in Appendix §6. SB is a new primitive for Byzantine fault tolerance. It works in an epoch-based manner and achieves high throughput through load balancing. However, its latency can reach seconds.

Clock synchronization applied to consensus protocols. CRAfT [75] and CockRoachDB [71] 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 high-accuracy clock synchronization [43, 45, 46]. Domino [83] and TOQ [73] use clock synchronization to accelerate Fast Paxos and EPaxos respectively. We compare them with Nezha in §9.3, and include more details in Appendix §F and §H.

12 CONCLUSION AND FUTURE WORK
Recent development of accurate software clock synchronization techniques brings us new opportunities to develop novel consensus protocols to achieve high performance in the 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.

We are considering two lines of future work. First, we intend to replace the Multi-Paxos/Raft backend used by some industrial systems (e.g., Kubernetes, Apache Pulsar, etc) so as to boost their performance. Second, we believe DOM can also be applied to other domains. We plan to integrate DOM with existing concurrency control algorithms (e.g., Two-Phase Locking, Optimistic Concurrency Control, etc) to improve their performance, or invent new concurrency control protocols based on DOM.

13 ACKNOWLEDGEMENTS
We thank Cisco Systems, Facebook (now Meta), Google, Nasdaq and Wells Fargo for sponsoring our research at Stanford’s Platform Lab. This work was also supported by NSF-2008048. We thank Shiyu Liu, Feiran Wang, Yilong Geng, Deepak Merugu, Dan Ports, Jialin Li, Ellis Michael, Jinyang Li, Aurojit Panda, Seo Jin Park, Zhaoguo Wang, Ken Birman, WeiJia Song and Eugene Wu for their feedback on our work and help with benchmarking.

REFERENCES
[1] [n.d.]. Derecho Discussion Issue 237. https://github.com/Derecho-Project/derecho/discussions/237.
[2] [n.d.]. RFE: Trans-reboot Monotonic Timers. https://github.com/systemd/systemd/issues/3107.
[3] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xylokris, and Igor Zablotski. 2020. Microscope Consensus for Microsecond Applications. In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, 599–616. https://www.usenix.org/conference/osdi20/presentation/aguilera.
[4] Ailidani Alijilang, Aleksey Charapko, and Murat Demirbas. 2019. Dissecting the Performance of Strongly-Consistent Replica 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/10.1145/3299689.3313893.
[5] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt on Causal Consistency. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (New York, New York, USA) (SIGMOD ’13). Association for Computing Machinery, New York, NY, USA, 761–772. https://doi.org/10.1145/2484586.2484579.
[6] Mihr Bellare and Daniele Micciancio. 1997. A New Paradigm for Collision-Free Hashing: Incrementality at Reduced Cost (EUROCRYPT ’97). Springer-Verlag, Berlin, Heidelberg, 163–192.
[7] Martin Biely, Zarko Milesevic, 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/SRDSD.2012.66.
[8] 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.
[9] Aleksey Charapko, Aliidani Alijilang, 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/10.1109/3448016.3452834.
[10] Indho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. 2020. Overload Control for µ-scale RPCs with Breakwater. In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, 299–314. https://www.usenix.org/conference/osdi20/presentation/cho.
[11] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. 2013. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP ’13). Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/2517349.2527112.
[12] Debian community. [n.d.]. ramsf. https://wiki.debian.org/ramsfs.
[13] Ben Darnell. [n.d.]. Scaling Raft. https://www.cockroachlabs.com/blog/scaling-raft.
[14] Xavier Défago, André Schiper, and Peter Urban. 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.
[15] Dmytri Duplyakin, Robert Bacx, Aleksander Marieg, Gary Wong, Jonathan Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangxing Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Singhshwain Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC ’19). USENIX Association, Renton, WA, 1–14. https://www.usenix.org/conference/atc19/presentation/duplyakin.
[16] Ramakrishnan Dururajan, Sathiya Kumar Mani, Joel Sommers, and Paul Barford. 2015. Time’s Forgotten: Using NTP to Understand Internet Latency. In Proceedings of the 14th ACM Workshop on Hot Topics in Networks (Philadelphia, PA, USA) (HotNets-XIV). Association for Computing Machinery, New York, NY, USA, Article 18, 7 pages. https://doi.org/10.1145/2834050.2834068.
[17] etcd. [n.d.]. Benchmarking etcd v3. https://etcd.io/docs/v3.5/etcd/etcd-v3-benchmarking/.
[18] Jinkun Gong, Aniruddh Sivaraman, Balaji Prabhakar, and Mendel Rosenblum. 2012. Nezha: Deployable and High-Performance Consensus Using Synchronized Clocks [Technical Report]. https://arxiv.org/abs/2206.03285.
[19] Yilong Gong. 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=14826.
[20] Yilong Gong, 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 Networks System Design and Implementation (Renton, WA, USA) (NSDI’18). USENIX Association, Berkeley, CA, USA, 81–94.
[21] Ahmad Ghaliyini, Jinkun Gong, Vighnesh Sashidaranada, Vinay Sri ram, 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/10.1145/3458536.3458578.
[22] Google. [n.d.]. Storage options. https://cloud.google.com/compute/docs/disks.
[23] Jim Gray, Prakash Sadasuresan, Susanne Engler, Ken Backlowski, 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.
[24] HashiCorp. [n.d.]. HashiCorp Raft. https://github.com/hashicorp/raft. Accessed 2022-03-31.
[25] 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/10.1145/98697.99072.
[26] Keon Jiang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable Message Latency in the Cloud. SIGCOMM Comput. Commun. Rev. 45, 4 (Aug. 2015), 342–356. https://doi.org/10.1145/2829988.2874779.
APPENDICES
In this appendix, we include the following:
- Explanation of Nezha’s recovery (§A).
- Correctness proof of Nezha (§B).
- Further evaluation of Nezha under different workloads (§C).
- Detailed discussion and evaluation about the effect of clock variance on Nezha’s performance (§D).
- Comparison with Derecho [29] (§E).
- Analysis of the incorrectness of Domino due to clock skews (§F).
- Formal comparison between DOM, MOM, and OUM (§G).
- Discussion about deploying Nezha in WAN and comparing Nezha to EPaxos in WAN environments (§H).
- TLA+ specification of Nezha (§I).
- Illustration of message delays regarding the protocols listed in Table 1 (§J).

A RECOVERY PROTOCOL AND ALGORITHMS
We explain how Nezha leverages the diskless crash recovery algorithm [44] from Viewstamped Replication in 3 steps. First, we explain how we adopt the recent concept of crash-vectors [49, 50] 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 [34] (i.e., messages that are sent out but not delivered before replica crash, so that the relaunched replicas forget them) using the crash-vector [49, 50]. 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.2) 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 [50] for detailed description of crash-vector).

Nezha uses crash-vectors to avoid two types of stray messages, i.e. (1) the stray messages during the view change process and (2) the stray messages (fast-replies) during quorum check. (1) has been clearly explained in [50], so here we only sketch how crash-vector works preserve the protocol correctness during the view change process. (2) has not been disclosed in prior works, so we will explain more details in §A.4 after we complete the explanation of the replica rejoin (§A.2) and leader change (§A.2).

A.1.1 Stray Message during View Change. There can be stray messages during the view change process: replicas mistakenly elect a leader, whose state falls behind the others, finally causing permanent loss of committed requests. 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.

A.1.2 Stray Message during Quorum Check. Stray messages can also occur during the quorum check in the fast path: some replicas send fast-replies and crash after that. The reply messages in the fast path (i.e., fast-reply) 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).

In brief, the crash-vector prevents the effect of such stray fast-replies, because we include the information of crash-vectors in the fast-replies (§6.3). When a failed replica rejoins the system (Algorithm 3), 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. After we describe the replica rejoin (§A.2) and leader change (§A.2.1), we come back to explain the details in §A.4.

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 3, 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.

Step 2: After receiving the crash-vector-req, replicas with normal status reply to the recovering replica with <crash-vector-req, 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. $cv[\text{replica-idx}] = cv[\text{replica-idx}] + 1$ (line 8).

There are many options available to generate the locally unique nonce string [44, 50]. Nezha uses the universally unique identifier (UUID) (generate-uuid in line 4), which have been widely supported by modern software systems.
Algorithm 3 Replica rejoin

Local State: 
- nonce, \( m.\) - A locally unique string on this replica
- \( C, m.\) - Reply set of CRASH-VECTOR
- \( R, m.\) - Reply set of RECOVERY-REP
- \( r, m.\) - short for replica-id (the message sender)
- \( cv, m.\) - short for crash-vector

status, view-id, last-normal-view, log, sync-point > Other attributes

upon RECOVER do
1. \( m.\) = RECOVERING
2. \( C = \emptyset \)
3. \( nonce = \) GENERATE-UUID
4. \( m.\) = CRASH-VECTOR
5. \( cv-set = \{ m.cv | m \in C \} \)
6. \( m.cv = \) AGGREGATE\((cv-set \cup \{ cv \})\)
7. \( m.cv[r] = cv[r] + 1 \) > Increment its own counter
8. \( R = \emptyset \)
9. \( R = READ-RECOVERY-INFO \)
10. \( highest-view = \max\{ m.o | m \in R \} \)
11. leader-id = highest-view \% \( (2f + 1) \)
12. while (leader-id = \( r \))
13. Pick \( m \in R: m.o = highest-view \)
14. state-transfer(leader-id)

function READ- CRASH-VECTOR
15. \( m.type = \) CRASH-VECTOR-REQ
16. \( m.r = r \)
17. \( m.cv = \) nonce
18. \( R = READ-RECOVERY-INFO \)
19. \( highest-view = \max\{ m.o | m \in R \} \)
20. \( R = READ-RECOVERY-INFO \)
21. \( highest-view = \max\{ m.o | m \in R \} \)
22. while (leader-id = \( r \))
23. Pick \( m \in R: m.o = highest-view \)
24. state-transfer(leader-id)

function READ-RECOVERY-INFO
25. \( m.type = \) RECOVERY-REQ
26. \( m.r = r \)
27. \( m.cv = cv \)
28. \( \) Send-message \((m, l)\) > Send message \( m \) to the replica \( l \)
29. \( \) Send-message \((m, l)\) > C is initialized as \( \emptyset \) by the caller

return

function STATE-TRANSFER\((l)\)
30. \( m.type = \) STATE-TRANSFER-REQ
31. \( m.r = r \)
32. \( m.cv = cv \)
33. \( \) Send-message \((m, l)\) > Send message \( m \) to the replica \( l \)
34. \( \) Wait until \( status = \) NORMAL

return

upon REceiving CRASH-VECTOR-REQ, \( m \) do
35. \( if (status \neq \) RECOVERING \) then
36. \( return \)
37. \( m.\) = CRASH-VECTOR-REP
38. \( m.\) = CRASH-VECTOR-REP
39. \( m.\) = CRASH-VECTOR-REP
40. \( m.\) = CRASH-VECTOR-REP
41. \( m.\) = CRASH-VECTOR-REP
42. \( m.\) = CRASH-VECTOR-REP
43. \( m.\) = CRASH-VECTOR-REP
44. \( m.\) = CRASH-VECTOR-REP
45. \( m.\) = CRASH-VECTOR-REP
46. \( m.\) = CRASH-VECTOR-REP
47. \( m.\) = CRASH-VECTOR-REP
48. \( m.\) = CRASH-VECTOR-REP
49. \( if (status \neq \) NORMAL \) then
50. \( return \)
51. \( m.\) = CRASH-VECTOR-REP
52. \( m.\) = CRASH-VECTOR-REP
53. \( m.\) = CRASH-VECTOR-REP
54. \( m.\) = CRASH-VECTOR-REP
55. \( m.\) = CRASH-VECTOR-REP
56. \( m.\) = CRASH-VECTOR-REP
57. \( m.\) = CRASH-VECTOR-REP
58. \( m.\) = CRASH-VECTOR-REP
59. \( m.\) = CRASH-VECTOR-REP
60. \( m.\) = CRASH-VECTOR-REP
61. \( m.\) = CRASH-VECTOR-REP
62. \( m.\) = CRASH-VECTOR-REP
63. \( m.\) = CRASH-VECTOR-REP
64. \( m.\) = CRASH-VECTOR-REP
65. \( m.\) = CRASH-VECTOR-REP
66. \( m.\) = CRASH-VECTOR-REP
67. \( m.\) = CRASH-VECTOR-REP
68. \( m.\) = CRASH-VECTOR-REP
69. \( m.\) = CRASH-VECTOR-REP
70. \( m.\) = CRASH-VECTOR-REP
71. \( m.\) = CRASH-VECTOR-REP
72. \( m.\) = CRASH-VECTOR-REP
73. \( m.\) = CRASH-VECTOR-REP
74. \( m.\) = CRASH-VECTOR-REP
75. \( m.\) = CRASH-VECTOR-REP
76. \( m.\) = CRASH-VECTOR-REP
77. \( m.\) = CRASH-VECTOR-REP
78. \( m.\) = CRASH-VECTOR-REP
79. \( m.\) = CRASH-VECTOR-REP
80. \( m.\) = CRASH-VECTOR-REP
81. \( m.\) = CRASH-VECTOR-REP
82. \( m.\) = CRASH-VECTOR-REP
83. \( m.\) = CRASH-VECTOR-REP
84. \( m.\) = CRASH-VECTOR-REP
85. \( m.\) = CRASH-VECTOR-REP
86. \( m.\) = CRASH-VECTOR-REP
87. \( m.\) = CRASH-VECTOR-REP
88. \( m.\) = CRASH-VECTOR-REP
89. \( m.\) = CRASH-VECTOR-REP
90. \( m.\) = CRASH-VECTOR-REP
91. \( m.\) = CRASH-VECTOR-REP
92. \( m.\) = CRASH-VECTOR-REP
93. \( m.\) = CRASH-VECTOR-REP
94. \( m.\) = CRASH-VECTOR-REP
95. \( m.\) = CRASH-VECTOR-REP
96. \( m.\) = CRASH-VECTOR-REP
97. \( m.\) = CRASH-VECTOR-REP
98. \( m.\) = CRASH-VECTOR-REP
99. \( m.\) = CRASH-VECTOR-REP
100. \( m.\) = CRASH-VECTOR-REP
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).

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 sets 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.2). 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 4, 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). 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. 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 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 ⌈f/2⌉+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 §8.2 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 consistent 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 [44]). 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
Algorithm 4 Leader change

Local State:
- \( V\), \( r\), \( cv\), \( last-normal-view\): \( > \) Reply set of view-change, \short for replica-id (the message sender), \short for crash-vector
- \( status, view-id, log, sync-point\) \( > \) The most recent view in which \( status \) is normal, \short for 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(o)
      6. status = view-change
      7. view-id = o
      8. \( V = \emptyset \)
        \( > \) Broadcast view-change-req to all replicas
      9. for \( i \leftarrow 0 \) to \( 2f + 1 \) do
         10. send-view-change-req(i)
           \( > \) Send view-change to the new leader
      11. send-view-change(o(\( 2f + 1 \))
      12. Wait until status = normal or Timeout
      13. return
      14. function send-view-change-req(i)
      15. \( m.type = view-change \)
      16. \( m.v = view-id \)
      17. \( m.cv = cv \)
      18. send-message(m, i)
      19. return
      20. function send-view-change(i)
      21. \( m.type = view-change \)
      22. \( m.v = view-id \)
      23. \( m.cv = cv \)
      24. \( m.log = log \)
      25. \( m.sp = sync-point \)
      26. \( m.lnv = last-normal-view \)
      27. send-message(m, i)
      28. return
      29. upon receive-view-change-req, m do
      30. if status = recovering then
      31. if check-crash-vector(m, cv)=false then
      32. if \( m.v > view-id \) then
      33. initiate-view-change(m.v)
      34. else
      35. if status = NORMAL then
      36. send-start-view(m.r)
      37. else
      38. \( > \) The leader is asking for fresher view-change
      39. send-view-change-req(m.r)
      40. function send-start-view(i)
      41. \( m.type = start-view \)
      42. \( m.v = view-id \)
      43. \( m.cv = cv \)
      44. \( m.log = log \)
      45. send-message(m, i)
      46. return
      47. upon receive-view-change, m do
      48. if status = recovering then
      49. return
      50. if check-crash-vector(m, cv)=false then
      51. return
      52. if status = normal then
      53. if \( m.v > view-id \) then
      54. initiate-view-change(m.v)
      55. else
      56. \( > \) The sender lags behind
      57. send-start-view(m.r)
      58. else if status = view-change then
      59. if \( m.v > view-id \) then
      60. initiate-view-change(m.v)
      61. else if \( m.v < view-id \) then
      62. send-view-change-req(m.r)
      63. else
      64. \( > \) Remove stray messages and add the fresh one
      65. \( V = \{ m' \in V \mid m.cv[m'r] < cv[m'r] \} \)
      66. \( V = V \cup \{ m \} - V' \)
      67. \( \forall m' \in V' \), resend view-change-req to \( m.r \)
      68. if \( |V| \geq f + 1 \) then
      69. \( log = merge-log(V) \)
      70. \( for i \leftarrow 0 \) to \( 2f \) do
      71. send-start-view(i)
      72. \( last-normal-view = view-id \)
      73. status = normal \( > \) Leader becomes normal
      74. function merge-log(V)
      75. \( new-log = \emptyset \)
      76. \( largest-normal-view = \max\{m.lnv|m \in V\} \)
      77. \( largest-sync-point = \max\{m.sp|m \in V\} \)
      78. Pick \( m \in V\); \( m.lnv = largest-normal-view \)
      79. \( m.sp = largest-sync-point \)
      80. \( > \) Directly copy entries up to sync-point
      81. \( for e \in m.log \) do
      82. \( > \) m.log is already sorted by deadlines
      83. \( entries = \{ e|e \in m.log \} \)
      84. \( and e.deadline > largest-sync-point.deadline \)
      85. \( and m.lnv = largest-normal-view \)
      86. \( for e \in entries \) do
      87. \( > \) Check how many replicas contain e
      88. \( S = \{ m|m \in V \land e \in m.log \} \)
      89. if \( |S| \geq \lceil f/2 \rceil + 1 \) then
      90. \( log.append(e) \)
      91. \( sort-new-log by entries' deadlines \)
      92. return new-log
      93. upon receive-view-change, m do
      94. if status = recovering then
      95. if check-crash-vector(m, cv)=false then
      96. return
      97. if \( m.v < view-id \) then
      98. \( last-normal-view = view-id = m.v \)
      99. \( log = m.log \)
      100. \( sync-point = log.last() \)
      101. status = normal \( > \) Followers become normal

24
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 in the sync messages, so that replicas can check and update their crash-vectors as soon as possible.

A.4 Why Does crash-vector Prevent Stray Message Effect during Quorum Check?

The stray messages can cause a common bug for most optimistic consensus protocols (e.g., [41, 63, 83]) when they conduct the quorum check in the fast path. Below we summarize the general pattern to cause the loss of committed requests (durability violation). Then, we explain why Nezha can avoid such problems by use of crash-vector.

General Error Pattern. Consider 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 opportunistic protocols, Speculative Paxos, NOPaxos and Domino all suffer from such cases. CURP [61] 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 re-launched, 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 (§8.1), which prevent stray reply messages from forming the super-quorum 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 3).
   - 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 75-76 in Algorithm 3), 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 super quorum.

2. When the leader fails, based on Algorithm 4, 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 the 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.

A.5 Reconfiguration

While it has not been implemented, Nezha can also use the standard reconfiguration protocol from Viewstamped Replication [44] (with its incorrectness fixed by crash-vector [49, 50]) 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.

B CORRECTNESS PROOF OF NEZHA

With the normal behavior described in §6.3–§6.4, 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 super quorum 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 by 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.2, 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.2. Therefore, req will be directly copied to new-log in Step 5 of §A.2, and eventually recovered.

B.2 Proof of Consistency

Without considering the acceleration of recovery mentioned in §7, 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.2. 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 \( \lceil f/2 \rceil \) 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.2. 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 workflow of the slow path (§6.4), 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 \(<\text{client-id, request-id, deadline}>\), the follower will delete ureq before updating its sync-point. Therefore, it is impossible for ureq to appear in 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 [73] 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 21. 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 [26, 55, 80] 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 and proxies 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 §F 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 §9.2, we set up 3 replicas and 2 proxies, and use 10 open-loop clients to submit at 10K request/sec each, thus yielding a throughput ~100K request/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 22) 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 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 22a, when the 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 a 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 degrade the latency performance, but the degradation is bounded.

**D.2.2 Bad Clock Synchronization of Follower Replica.** As shown in Figure 22b, similar to the cases in Figure 22a, the follower’s bad clock synchronization also inflates the latency. However, the negative impact of the 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 deadlines 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 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.** As shown in Figure 22c, 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 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 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 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...
The difference in Figure 5. The difference in Figure 5.

(a) Impact of the leader’s clock synchronization performance
(b) Impact of the follower’s clock synchronization performance
(c) Impact of the proxies’ clock synchronization performance

Figure 22: Nezha latency vs. clock synchronization quality

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 Huỳgens 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 (§8.2). 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 $\otimes$ in Figure 5. 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 $\otimes$ in Figure 5 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 22a and Figure 22c. We configure a threshold for the leader replica: if the request’s deadline is larger than the replica’s current clock time by 50μs, 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 installing the optimization strategy, we can see from Figure 22a and Figure 22c 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 creates 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 DERECHO IN THE PUBLIC CLOUD

Derecho [29, 66] 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 intended to compare Nezha with Derecho in the 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 [66] by using bare-metal machines. We set up a cluster in Cloudlab [15]. We use 3 c6525-100g instances (equipped with 100GB RDMA NICs) to form a Derecho subgroup size of 3. We use rams [12] 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 request/sec with 1KB message size and 634K request/sec with 100B message size. As for the one-sender mode, Derecho yields the throughput of 313K...
request/sec with 1KB message size and 305K request/sec with 100B message size. These numbers are close to the reported number in [66], 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 request/sec with the median latency of 2.33 ms; the one-sender mode achieves the throughput of 5.68K request/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 request/sec with the median latency of 2.0 ms; the one-sender mode achieves the throughput of 4.93K request/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 [59]. 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.

F ERROR TRACES OF DOMINO

Domino [83] is a recently proposed solution to achieve consensus with clock synchronization. When clock skew happens, clients may consider the request as committed, but eventually the request is lost from the replicas, leading to durability violation. The key reason for the durability violation is because clocks cannot always maintain monotonically increasing values. This lack of monotonicity is true both for NTP, used by Domino and Huygens, which Nezha uses. However, Nezha does not rely on clock monotonicity for correctness.10 In this section, we will use an error trace to demonstrate Domino’s durability violation due to the non-monotonicity of clock time.

Error Trace 1: There are 5 replicas in Domino, and we denote them as R0-R4. Suppose R0 is the DFP (Domino’s Fast Paxos) leader and the others are the followers. 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 [83]).
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 broadcasts 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 because the request is dropped.
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. 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 (a) replace request-2 with no-op and only execute request-1 or (b) execute both requests but with request-2 first and request-1 second.

The choice between (a) and (b) in Step 11 depends on how Domino implements the coordination between the leader and the other learners (followers), which is not shown in the Domino paper [83]. We have studied the implementation [82] of Domino and found that, followers will choose (a) because DFP leader will also broadcast the log positions (refer to NonAcceptTime variable in [82]) which the leader fills no-ops. When followers choose (a), Domino violates durability because request-2, which have been considered committed, is lost permanently. As an alternative, if followers choose (b), consistency will be violated: After the DFP leader (i.e., 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).

Furthermore, the durability property is a necessary condition for the consistency and linearizability properties (as we defined in §B):

- 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.

Hence, Error Trace 1 has shown that Domino can violate durability, and consequently violate consistency and linearizability property.

Since the Domino paper [83] does not specify the consistency model it targets, we speculate that Domino is based on eventual consistency [5, 28, 72, 74], because the stronger consistency model, causal consistency, is not guaranteed by Domino’s protocol design: suppose one Domino client has two requests, i.e., request-1 and request-2. If request-1 influences the creation of request-2 but the client assigns a larger arrival time to request-1, then the other clients can see the result of request-2 earlier than the result

---

10NTP is known to be unable to guarantee monotonicity [2, 16, 65]. Huygens synchronizes clocks by tuning the clock frequency instead of adding offsets to clock time. Thus it can provide monotonically increasing clock time if Huygens does not fail. However, when Huygens fails and is restarted, or when Huygens changes its reference clock, it cannot guarantee monotonicity either.

11The three consistency models can be sorted from the weakest to the strongest: eventual consistency < causal consistency < linearizability. A more detailed comparison can be found in [28].
of request-1. Thus Domino does not provide causal consistency. However, eventual consistency cannot be guaranteed either when durability is violated [51].

Nezha avoids such error cases because Nezha exploits synchronized clocks to reduce packet reordering in the network, rather than to 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 the 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.2). Therefore, Nezha’s correctness is independent of the clock behavior. However, the clock synchronization indeed affects the performance of Nezha. For example, if the clock time of the replica becomes much faster and goes to a very large value, it can release some requests with very large deadlines. The large deadlines will be used in the eligibility check of the early-buffer, making the subsequent requests unable to enter the early-buffer and trigger the slow path more frequently. We have discussed in §D different cases of bad clock synchronization and their impact on Nezha.

By contrast, Domino directly uses the clock time for ordering, and does not expect that the clocks can also give a smaller time than before, violating monotonicity (Step 7 and Step 8 in Error Trace 1), which leads to the incorrectness of the protocol.

G  FORMAL COMPARISON OF DIFFERENT PRIMITIVES

Concretely, the mostly ordered multicast (MOM) primitive [63] 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 [41] 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 guarantee consistent ordering, so as to ease the work for replication protocols to achieve state consistency (i.e. to satisfy both consistent ordering and set equality). In this section, we aim to make a formal comparison among the three primitives.

G.1 Notation

• Replicas: R1, R2, .
• Messages: M1, M2, .
• \( a(M_i, R_k) \): the arrival time of \( M_i \) at \( R_k \). It is the reference time which is not accessible by replicas, replicas can only get an approximate \( \tilde{a}(M_i, R_k) \) 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 \( \tilde{r}(M_i, R_k) \) 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.

G.2 Definition

• Packet drop: \( M_i \) is lost to \( R_k \) if \( r(M_i, R_k) = \infty \)
• consistent ordering: \( R_1 \) and \( R_2 \) are said to be consistently 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 \( r(M_1, R_2) > r(M_2, R_2) \)
  - \( \forall r(M_1, R_1) < r(M_2, R_1) \) and \( r(M_1, R_2) < r(M_2, R_2) \)

For simplicity, we omit discussing the edge case \( r(M_1, R_1) = r(M_2, R_1) \) and/or \( r(M_1, R_2) = r(M_2, R_2) \), which can be categorized into either of the two aforementioned outcomes. Similar edge cases are also omitted in the discussion of §G.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 \( r(M_1, R_2) = \infty \)
  - \( \forall r(M_1, R_1) < \infty \) and \( r(M_1, R_2) < \infty \)

Set equality is similar to the term reliable delivery in NOPaxos [41]. 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:

\[
\forall M_i, M_j : UO(R_1, R_2, M_i, M_j) \land SE(R_1, R_2, M_i) \land SE(R_1, R_2, M_j)
\]

Satisfying both \( UO \) and \( SE \) property is equivalent to implementing an atomic broadcast primitive [14], which is as hard as the consensus protocol.

G.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.

G.3.1 MOM

\[
\begin{align*}
\langle M_1, R_k \rangle &= a(M_1, R_k) \\
\langle M_2, R_k \rangle &= a(M_2, R_k)
\end{align*}
\]

\( r(\ast, \ast) \) is completely determined by \( a(\ast, \ast) \) without guaranteeing consistent ordering.

G.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

\[
\begin{align*}
\langle M_1, R_k \rangle &= a(M_1, R_k) \\
\langle M_2, R_k \rangle &= a(M_2, R_k)
\end{align*}
\]

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

\[
\begin{align*}
\langle M_1, R_k \rangle &= \infty \\
\langle M_2, R_k \rangle &= a(M_2, R_k)
\end{align*}
\]

Equation 2 captures the case where \( M_1 \) and \( M_2 \) arrive in an order consistent with their sequence numbers.

12The example of causal consistency is adapted from the example in [5].
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.

Consistent 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.

G.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 modeling [38, 45, 47]. It is worth noting that, this assumption does not always hold in practice (we have explained this in §F). Here, we are not focusing on the protocol correctness (Nezha’s correctness does not require this assumption). Instead, we only adopt this assumption to simplify the comparison between DOM and MOM/OUM. Since the non-monotonicity of clock time occurs rarely, such rare cases will not affect the conclusions we derived on the overall performance of DOM and Nezha.

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 of 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:

$$r(M_1, R_k) < r(M_2, R_k) \iff \hat{r}(M_1, R_k) < \hat{r}(M_2, R_k)$$

$$a(M_1, R_k) < a(M_2, R_k) \iff \hat{a}(M_1, R_k) < \hat{a}(M_2, R_k)$$

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

$$\hat{r}(M_1, R_k) = \max\{D(M_1), \hat{a}(M_1, R_k)\}$$

$$\hat{r}(M_2, R_k) = \max\{D(M_2), \hat{a}(M_2, R_k)\}$$

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

$$\hat{r}(M_1, R_k) = \infty$$

$$\hat{r}(M_2, R_k) = \max\{D(M_2), \hat{a}(M_2, R_k)\}$$

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.

Consistent 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.

G.4 Understanding Difference between MOM, OUM and DOM

As shown in Figure 23, primitives are decoupled from the replication protocol. None of the primitives guarantees consistency defined in §G.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 consistent 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 reach 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 superset 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 $M_2$ come 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.

G.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, it should be very close to $a(M_1, R_1)$, leading to $a(M_1, R_1) < D(M_2)$, 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 $a(M_2, R_1)$ that leads to $a(M_2, R_1) > D(M_3)$. 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 ($\S$4). When $M_1$ and $M_2$ are given very large deadlines $D(M_1)$ and $D(M_2)$, then DOM is 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 $a(M_2, R_1)$ that leads to $a(M_2, R_1) > D(M_3)$.

**Good Case**: Clocks are well synchronized and $D(M_1)$ is properly decided, i.e. $D(M_1)$ is close to (but slightly larger than) the arrival time $a(M_1, R_1)$ according to most replicas. In this case, when the network is good, DOM delivers the message to the replication protocol with both consistent ordering and set equality, just like MOM and OUM. More than that, when the network causes message 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 the slow path and costly rollback; OUM presents consistent order messages with gaps (equation 3), which also causes NOPaxos to go to the 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.

H NEZHA VS. EPAXOS IN WAN

EPaxos [56, 73] is a consensus protocol that 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 is 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 is 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 mainly focus on LAN deployment and have shown that Nezha outperformed TOQ-EPaxos in LAN (Figure 8), we would like to highlight that Nezha is also deployable in WAN environment, and we have demonstrated in Figure 13 that, Nezha can also earn more advantages over EPaxos when deployed in WAN. We analyze the advantages below.

H.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 makes EPaxos lose most of its latency benefit. 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.

H.2 Throughput
While EPaxos uses multiple leaders to mitigate single-leader bottleneck, Nezha adopts an alternative design: Nezha still maintains single leader but offloads 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 checks, 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.

H.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 [73] and our experiments (§9.8), which evaluate Huygens in the WAN setting and observes the clock offsets between 20 µs and 1 ms. Considering the inter-datacenter latency is usually tens of or even hundreds of milliseconds (as shown in Figure 5 of [73]), the synchronization performance of Huygens is sufficient for Nezha to achieve fast consensus in WAN.

I NEZHA TLA+ SPECIFICATION
The TLA+ specification of Nezha is available at the anonymous repository https://github.com/Steamgjk/Nezha/blob/main/docs/Nezha.tla.

J ILLUSTRATION OF MESSAGE DELAYS IN DIFFERENT PROTOCOLS
We illustrate the workflow for each of the protocols we listed in Table 1.
Multi-Paxos/Raft requires 4 message delays to commit a request (Figure 24).

Fast Paxos requires 3 message delays to commit a request in the fast path (Figure 25) and 5 message delays in the slow path (Figure 26).

Speculative Paxos requires 2 message delays to commit a request in the fast path (Figure 27) and 6 message delays in the slow path (Figure 28).
NOPaxos requires 2 message delays to commit a request in the fast path (Figure 29) and 4 message delays in the slow path (Figure 30). Note that when the sequencer is a software sequencer (e.g., a standalone VM) instead of a hardware sequencer (e.g., programmable switch), there will be one extra message delay added to both the fast path and the slow path.

Mencius requires 4 message delays to commit a request in the fast path (Figure 31) and 6 message delays in the slow path (Figure 32), because the leader who is responsible to commit the request needs to take one extra round (2 message delays) to coordinate with the other leader replicas.

EPaxos requires 4 message delays to commit a request in the fast path (Figure 33) and 6 message delays in the slow path (Figure 34).

CURP requires 2 message delays to commit a request in the fast path (Figure 35) and 4 message delays in the slow path (Figure 36).
Figure 35: Message Delay of CURP (Fast Path)

Figure 36: Message Delay of CURP (Slow Path)