Nezha: Deployable and High-Performance Consensus Using Synchronized Clocks

Jinkun Geng*, Anirudh Sivaraman+, Balaji Prabhakar*, Mendel Rosenblum*
*Stanford University, +New York University

ABSTRACT

This paper presents a high-performance consensus protocol, Nezha, designed for single-cloud-region environments, 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 tags 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-based 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% of latency performance 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://gitlab.com/steamgjk/nezhav2.

1 INTRODUCTION

Our goal in this paper is to build a high-performance consensus protocol for a local-area network such as a single cloud zone, 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 [7, 36, 43]. Second, microsecond-scale applications are pushing the limits of computing [1, 18, 20, 24, 27]. Such applications often have stateful components that must be made fault-tolerant (e.g., the matching engine within a fair-access cloud stock exchange [12], details in §9). 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 [23] and Raft [41] 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 [8]. Second, high-performance alternatives such as NOPaxos [26], Speculative Paxos [44], NetChain [19], NetPaxos [19], and Mu [1], 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 [22], EPaxos [39], Speculative Paxos [44], and NOPaxos [26].

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 §2, 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 multicast that works as follows. The sender's 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 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 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

[1] We 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 [37] are an exception, but such instances are expensive.

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 [28].

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 overhead on a Nezha client relative to a typical client of a protocol like Multi-Paxos or Raft. This overhead arises because the client is now doing additional work per client request relative to a typical consensus client. 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 public cloud: Multi-Paxos, Fast Paxos, our optimized version of NOPaxos, Raft, Domino and TOQ-based EPaxos under closed-loop and open-loop workloads. In a closed-loop workload, commonly used in the literature [26, 35, 39, 44], 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 [50], 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 outperform five baselines by 1.3–3.8× 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) and show that Nezha can provide fault tolerance with only a modest performance degradation: compared with the unreplicated system, Nezha sacrifices 5.9% throughput for Redis; it saturates the processing capacity of CloudEx and prolongs the order processing latency by 4.7%.

Nezha is open-sourced at https://gitlab.com/steamgjk/nezhav2.

2 MOTIVATION

Consensus protocols are often used to provide the abstraction of a replicated state machine (RSM) [47], where multiple servers cooperate to present a fault-tolerant service to clients. In the RSM setting, the goal of consensus protocols is to get multiple servers 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 as below.

For any two replicas R_1 and R_2:

- **Consistent ordering.** If R_1 processes request a before request b, then R_2 should also process request a before request b, if R_2 received both a and b.
- **Set equality.** If R_1 processes request a, then R_2 also processes request a.

Many *optimistic* protocols leverage the fact that the ordering of messages from client 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 [22, 26, 44, 59]. 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 [22, 44] 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 reordering rate 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. We use these sequence numbers to check how reordered R_2 is. We count a message received by R_2 as reordered if (1) its sequence number is smaller than the sequence number of R_2's previous message, or (2) the message does not exist in R_1's sequence (because it was dropped at R_1). Figure 1 shows that when we vary the submission rate, keeping the number of senders fixed at 2, the reordering rate quickly exceeds 20%. Further in Figure 2, when we vary the number of senders, keeping the submission rate fixed at 10K messages/second, the reordering rate increases rapidly up to 36% with the number of senders.

In the public cloud, with such high reordering rates, optimistic protocols are forced to take the slow path often, which reduces their performance (§8.2). 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

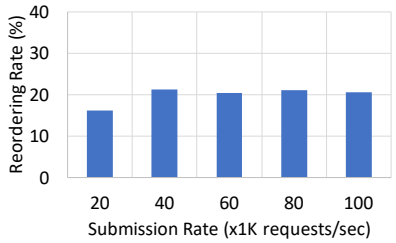


Figure 1: Packet reordering vs. submission rate on Google Cloud

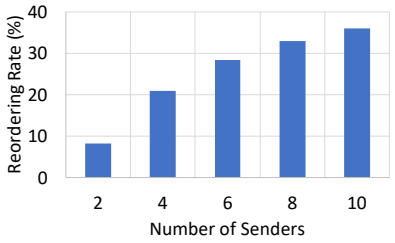


Figure 2: Packet reordering vs. number of senders on Google Cloud

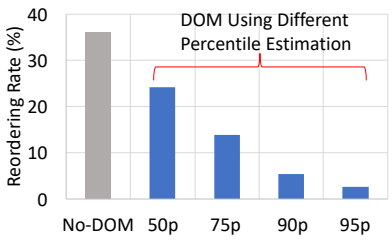


Figure 3: Effectiveness of DOM on packet reordering on Google Cloud

protocol, Nezha, to go along with DOM because guaranteeing both requirements has been shown to be as hard as consensus itself [4].

3 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 has passed 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 $D(M)$ 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 ever. 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 the 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 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 late or lost messages, it is up to the slow path of the consensus protocol to handle such 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 [4]. The use of clock synchronization for performance (i.e., increasing the frequency of the fast path) rather correctness (i.e., linearizability) is also in line with Liskov's suggestion on how synchronized clocks should be used [28].

Setting DOM deadlines. Setting deadlines is a trade-off between avoiding message reordering and adding too much waiting time. 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.

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

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 with its receiving time. Then 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 [11] the standard deviation for the sending time and receiving time, denoted as σ_S and σ_R [2]. σ_S and σ_R provide an *approximate* error bound for the synchronized clock time, so we add the error bound with a factor β 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 $[0, D]$, we will use D as the estimated OWD. The estimated OWD will be replied to the sender to decide the deadlines of subsequent requests.

To illustrate DOM's benefits, we redo our experiments from §2 with 10 Poisson senders, each submitting 10K requests/sec to 2 receivers. Figure 3 shows different percentiles (i.e., 50^{th}, 75^{th}, 90^{th}, and 95^{th}) 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.

4 NEZHA OVERVIEW

We use DOM as a building block to develop a consensus protocol, called Nezha, atop DOM. We overview the protocol here and

[2] σ_S and σ_R are calculated based on the method in Appendix A of [10].

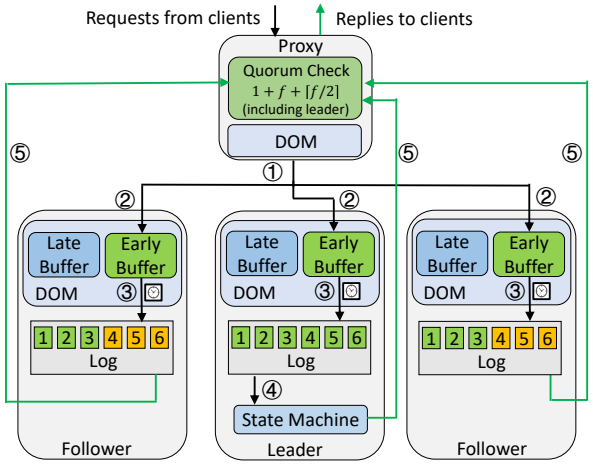


Figure 4: Fast path of Nezha

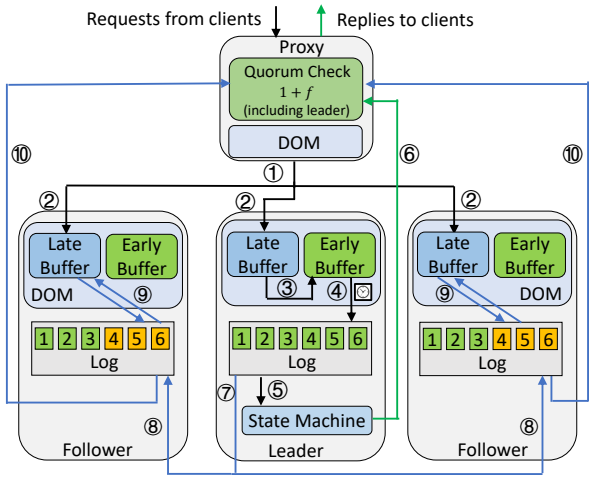


Figure 5: Slow path of Nezha

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 (linearizability) at all times and liveness under the same assumptions as Multi-Paxos/Raft. 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 [28, 30, 31].

Nezha architecture. Nezha uses a stateless proxy/proxies (Figures 4 and 5) interposed between clients and replicas to relieve clients of the 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 the maximum of a sliding window median (50^{th} percentile) of OWD estimates between the proxy and each replica; these deadlines also take into account the current estimate of clock synchronization errors (§3). 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.

Fast/Slow path sketch. 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 ①. If the request's deadline is larger than the last request released from the *early-buffer*, the request enters the *early-buffer* ②. It will be released from the *early-buffer*s at the deadline, so that replicas can append the request to their *log*s ③. The *log* list is ordered by request deadline. After that, followers immediately send a reply to the proxy without executing the request ⑤, whereas the leader first executes the request ④ and sends a reply including the execution result. The proxy considers the request as committed after receiving replies from the leader and $f + \lceil f/2 \rceil$ 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 ($f+\lceil f/2 \rceil+1$) rather than a simple quorum ($f+1$) for the same reason as Fast Paxos [22]: 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 §5.3).

Figure 5 shows the more involved slow path: when a multicasted ① request goes to the *late-buffer* because of its small deadline ②, 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* ③. After releasing and appending this request to the log ④, the leader broadcasts this request's identifier (a 3-tuple consisting of *client-id*, *request-id*, and request deadline) to followers ⑦, to force followers to keep consistent logs with the leader. On hearing this broadcast ⑧, 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-buffer*s without having to ask the leader for these entries ⑨. After this, followers send replies to the proxy ⑩. Meanwhile, the leader has executed the request ⑤ and replied to the proxy ⑥. After collecting $f+1$ replies (including the leader's reply), the proxy considers the request as committed. Notably, Nezha differs from the other optimistic protocols (e.g. [22, 26, 44]): it also decouples the request execution and quorum check in the slow path. Such a decoupling design enables a *faster* slow path for the proxy to commit requests in the slow path. Besides, through the quorum check, the proxy can ensure that the speculative execution result from the leader replica is safe to use.

5 THE NEZHA PROTOCOL

5.1 Replica State

Figure 6 summarizes the state variables maintained by each replica. We omit some variables (e.g., *crash-vector*) related to Nezha's recovery (§6). Below we describe them in detail.

- *replica-id*— replica identifier ($0, 1, \cdots, 2f$).
- *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 6: Local state of Nezha replicas

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 process, 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 [29]: 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*$\%(2f+1)$.

status: Replicas switch between three different *status*es. Replicas are initially launched in NORMAL status. When the leader is suspected of failure, followers switch from NORMAL to VIEWCHANGE 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 their deadlines' order, 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 such a buffer because those requests may later be needed in the slow path (§5.4). 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 *log*s to keep consistent with the leader (§5.4). *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 (§7.3).

5.2 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 related to leadership changes; these messages have been defined in Viewstamped Replication [29].

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:

$$request=<client\text{-}id, request\text{-}id, command, s, l>$$

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 l are tagged by proxies. s is the sending time of the *request* and l is the estimated latency bound. When the request arrives at the replica, the replica can derive the request's deadline as $s+l$. Meanwhile, 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:

$$fast\text{-}reply = <view\text{-}id, replica\text{-}id, client\text{-}id, request\text{-}id, result, hash>$$

view-id and *replica-id* are from the replica state variables (see §5.1). *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*. *hash* captures a hash of the replica's *log* (the hash calculation is explained in §7.1). Proxies can check the *hash* values to know whether the related replicas have consistent *log*s.

log-modification: *log-modification* message is broadcast by the leader to convey its *log* identifier (*deadline+client-id+request-id*) to followers, making the followers modify their *log*s to keep consistent with the leader. Meanwhile, *log-modification* also doubles as the leader's heartbeat. *log-modification* is represented as a 5-tuple:

$$log\text{-}modification=<view\text{-}id, log\text{-}id, client\text{-}id, request\text{-}id, deadline>$$

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 on each replica. *deadline* is the request's deadline shown in the leader's *log*, which is either assigned by proxies on the fast path or overwritten by the leader on the slow path (i.e., ③ 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 *log*s have become consistent with the leader's *log* 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:

$$slow\text{-}reply = <view\text{-}id, replica\text{-}id, client\text{-}id, request\text{-}id>$$

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.

5.3 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 50^{th} 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 $f + \lceil f/2 \rceil$ 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 $f + \lceil f/2 \rceil + 1$ replicas have matching *view-id*s and the same *log* (requests). In §7.1 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 fast path (§4), the fast path requires a super quorum ($f + \lceil f/2 \rceil + 1$) 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. 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 [16]: the client sees *request-1* executed before *request-2* with the old leader and sees the reverse with the new leader.

5.4 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 ②, the leader will modify its deadline to make it eligible ③. The leader then conducts the same operations as in the fast path (i.e., appending the request ④, applying it to the state machine ⑤, and sending *fast-reply* ⑥). The leader also broadcasts the *log-modification* message ⑦ in parallel with ⑤-⑥.

Leader broadcasts log-modification. Every time the leader appends a request to its *log*, it broadcasts a *log-modification* message to followers ⑦. Every time a follower receives a *log-modification* message ⑧, 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 that case, the follower removes the wrong entry and tries to put the right one. It first searches its *late-buffer* for the right entry with matching *<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 consistent with the leader, the follower updates its *sync-point*, indicating its *log* is consistent with the leader up to the log position indicated 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 ⑩. 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 related *fast-reply* from the leader and the *slow-replies* from f followers. The execution result is still obtained from the leader's *fast-reply*. Our decoupling design 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 §8.2). Unlike the quorum check of the fast path (§5.3), the slow path does not need a super quorum ($1 + f + \lceil f/2 \rceil$). This is because, before sending *slow-replies*, the followers have updated their *sync-point*s and ensured that all the requests (log entries) are consistent with the leader up to the *sync-point*s. A simple majority ($f + 1$) is sufficient for the *sync-point* to survive the crash. All requests before *sync-point* are committed requests, whose log positions have all been fixed. During the recovery (§6), they are directly copied to the new leader's *log*.

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-point*s. The leader can know which requests have been committed by collecting the *sync-point*s 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 checkpoints its state at the *commit-point*. It can also broadcast the *commit-point* to followers, which enables them to checkpoint their states for acceleration of recovery (§7.3). 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.

5.5 Timeout and Retry

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

6 RECOVERY

Assumptions. We assume replica processes can fail because of process crashes or a reboot of its server. When a replica process fails, it will be relaunched on the same server. However, we assume that there is some stable storage (e.g., disk) that survives process crashes or server reboots. A more general case, which we do not handle, is to relaunch the replica process from a different server with a new disk where the stable storage assumption no longer holds. We also do not handle the case of changing Nezha's f parameter by adding or removing replicas from the system. Both cases are handled by the literature on reconfigurable consensus [29, 54], which we believe can be adapted to Nezha as well.

Recovery protocol. Nezha's recovery protocol consists of two components: replica rejoin and leader change. After a replica fails, it can only rejoin as a follower. If the failed replica happens to be the leader, then the remaining followers will stop processing requests after failing to receive the leader's heartbeat for a threshold of time. Then, they will initiate a view change to elect a new leader before resuming service. In [9], we describe the protocol with both pseudo-code and a model-checked TLA+ specification, and also include the correctness proof. Here, we only sketch the major steps for the new leader to recover its state (*log*).

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

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

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

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

In some cases, the leader change can happen not only because of a process crash but also because of a network partition, where followers fail to hear from the leader for a long time and start a view change to elect the new leader. When the deposed leader notices the existence of a higher view, it needs to abandon its current state, because its current state may have diverged from the state of the new leader. In other words, the state of the deposed leader may include the execution of some uncommitted requests, which do not exist in the new view. To maintain correct state, the deposed leader transfers the state from another replica in the fresh view.

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 can 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* [21], which proved to cause bugs to multiple diskless protocols (e.g., [29][44][26][59]). Nezha adopts the *crash-vector* technique invented by Michael et al. [33, 34], 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. We omit the details due to space limit, and include them in [9].

7 OPTIMIZATIONS IN NEZHA

7.1 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 this 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.

7.2 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 [6, 42]. The commutativity optimization enables us to choose a modest percentile (50^{th} 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*. The request can enter the *early-buffer*, so long as its deadline is larger than the last released request *which is not commutative with the incoming request*. Second, it refines the hash computation. While sending the *fast-reply* related to a request, the replicas only need to XOR the hashes of all *write* requests which have been previously appended and *are not commutative with this incoming request*.

We explain more details about our commutativity optimization and also conduct evaluation across a range of workloads in [9], with different read/write ratio and skew factors. The result shows that the commutativity optimization helps reduce the latency by 7.7 %-28.9 %.

7.3 Periodic Checkpoints

To (1) accelerate the recovery process after leader failure and (2) enable the deposed leader to quickly catch up with the fresh state, we can integrate periodic checkpointing into 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 partition. 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 conduct synchronization between the leader and followers in the background.

Periodically, the followers report their *sync-point*s 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-point*s. 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 partition happens, 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 §6), it can start from its latest checkpoint state, and only retrieve from another replica (in the fresh view) the requests beyond its *commit-point*. The exact implementation of checkpointing state is application-specific and we leave this to future work. When evaluating our failure recovery strategy, we use a *null* application where the state is the log itself.

8 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?

(2) How does Nezha compare to the recent protocols which also use clock synchronization (i.e., Domino and TOQ-based EPaxos)?

(3) How effective are the proxies, especially when there is a large number of replicas?

(4) How fast can Nezha recover from the leader failure?

(5) How does Nezha compare to Raft when both are equipped with log persistence to stable storage?

(6) Does Nezha provide sufficient performance for replicated applications?

8.1 Settings

Testbed. We run experiments in Google Cloud. We employ `n1-standard-4` VMs for clients, `n1-standard-16` VMs for replicas and NOPaxos sequencer, and `n1-standard-32` VMs for Nezha proxies. All VMs are in a single cloud zone. Huygens is installed on all VMs and has an average 99^{th} 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 [25] with necessary modification: (1) we change multicast into multiple unicasts because network-support multicast is unavailable in cloud. (2) we use a software sequencer with multi-threading 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 [59] and TOQ-based EPaxos [50]. We choose to compare them with Nezha because they also use clock synchronization to accelerate consensus. For Domino, it is previously tested with clocks synchronized by Network Time Protocol (NTP) in [59], but in our test, we also provide Huygens synchronization for Domino to give it more favorable conditions because Huygens has higher accuracy than NTP [11]. However, since both Domino and TOQ-based 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 (§8.2).

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 have attempted to measure tail latency at the 99th and 99.9th percentile. But we find it hard to reliably measure these tails because tail latencies within a cloud zone can exceed a millisecond [17, 38, 56]. This is unlike the WAN setting where tails can be more reliably estimated [50]. We run each experiment 5 times and average values before plotting.

Evaluation method. We follow the method of NOPaxos [26] and run a *null application* with no execution logic. Traditional evaluation of consensus protocols [26, 35, 39, 40, 44, 52] 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 [50] 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 §8.2, we use 3

replicas. For the closed-loop test, we increase load by adding more clients until saturation [3]. 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 [14] of 0.5. We also evaluate Nezha under various read ratios and skew factors in [9], which verifies the robustness of its performance.

8.2 Comparison with Multi-Paxos, Fast Paxos and NOPaxos

The closed-loop and open-loop evaluation results are shown in Figure 7. We plot two versions of Nezha. Nezha-Proxy uses standalone proxies whereas Nezha-Non-Proxy lets clients undertake proxies' work. Below we discuss three main takeaways.

First, all baselines yield poorer latency and throughput in public cloud, in comparison with published numbers from highly-engineered networks [26]. Fast Paxos suffers the most and reaches only 4.0K requests/second at 425 µs in open-loop test (not shown in Figure 7b). 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 the 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 most CPU cycles. We realize this issue and develop an optimized version (NOPaxos-Optim in Figure 7) 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 10 open-loop clients submit 10K requests/sec each, NOPaxos replicas trigger gap handling (slow path) for more than 30% of requests. Besides, the gap handling also *blocks the processing of follow-up requests* because NOPaxos still relies on the leader replica to do quorum checks in the slow path. Once a previous request triggers the slow path, all the follow-up requests will be made to wait for at least 1 RTT before the leader completes gap agreement. Thus, all these follow-up requests will count the gap handling cost into their latencies, and they can also continue to cause more gaps.

Last, Nezha achieves much higher throughput than all the baselines, and Nezha-Non-Proxy also achieves the lowest latency because of co-locating proxies with clients. Even equipped with standalone proxies, Nezha-Proxy still outperforms all baselines at their saturation throughputs, except NOPaxos-Optim (open-loop). Nezha's 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 §8.4.

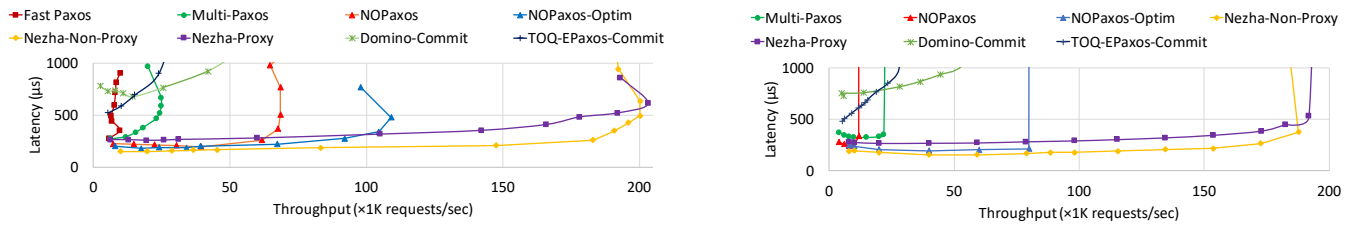
[3] Specially, when the system is saturated, the throughput can drop instead of continuously increasing [5, 53], as shown in Figure 7.

8.3 Comparison with Domino and TOQ-based EPaxos

For a fair comparison of Domino and TOQ-based EPaxos with Nezha, we originally wanted to plot their execution latencies in Figure 7. However, both Domino and TOQ-based EPaxos decouple commit from execution, and execution happens much later than commit, which causes high execution latencies. In our experiments, we found that Domino's execution latency exceeds 10 ms and TOQ-based 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-based 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-based EPaxos are implemented in Golang with gRPC [46, 58] whereas Nezha and the other baselines are implemented in C++ with UDP [25]). 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-based EPaxos below from a design perspective.

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) Domino uses a more conservative estimation of OWD at the 95^{th} percentile, rather than Nezha's estimate at the 50^{th} percentile. While a 50^{th} percentile will reduce the fast commit ratio, Nezha compensates for it by leveraging commutativity (see §7.2) and a more optimized slow path that uses speculative execution. (3) Domino directly uses a request's deadline as a position in the log, which causes 2 problems. First, it means that many log entries will be no-ops because there was no request at that particular time/position. Second, 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. This causes a violation of durability and subsequently linearizability as we explain with error traces in [9]. Compared with Domino's "timestamp-as-log-position," Nezha uses deadlines to reduce packet reordering, but the log positions of requests are eventually decided by the replicas. Therefore, Nezha's correctness is independent of clock skew.

Nezha compared with TOQ-based EPaxos. Nezha differs from TOQ-based EPaxos in 2 ways. (1) TOQ only mitigates the conflict



(a) Closed-loop workload

(b) Open-loop workload

Figure 7: Latency vs.throughput

rate for EPaxos but does not optimize the message flow of EPaxos. When it comes to LAN, where there is no difference between LAN RTTs and WAN RTTs, the EPaxos protocol inherently uses more RTTs than Nezha. Besides, there are no proxies to assist EPaxos replicas in request multicast and quorum check, so its replicas' burden is heavier than Nezha. (2) TOQ does not maintain consistent ordering for its released requests, so the protocol (EPaxos) still needs to handle both consistent ordering and set equality to commit requests, whereas DOM enables Nezha to only focus on set equality.

8.4 Ablation Study

During the ablation study of Nezha, we remove one component from the full protocol of Nezha each time, and yield three variants, shown as No-DOM, No-QC-Offloading, No-Commutativity in Figure 8. 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 mutlicast). No-Commutativity variant disables Nezha's commutativity optimization. We run all protocols under the same setting as Figure 7b.

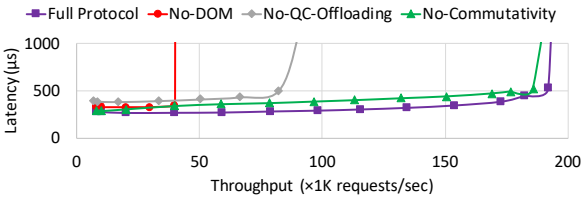
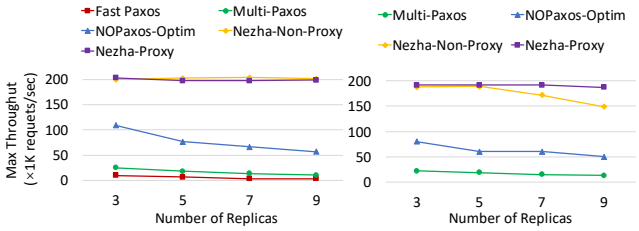


Figure 8: Ablation study of Nezha

Figure 8 shows that, removing any of the three components can degrade the performance (i.e., throughput and/or latency).

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

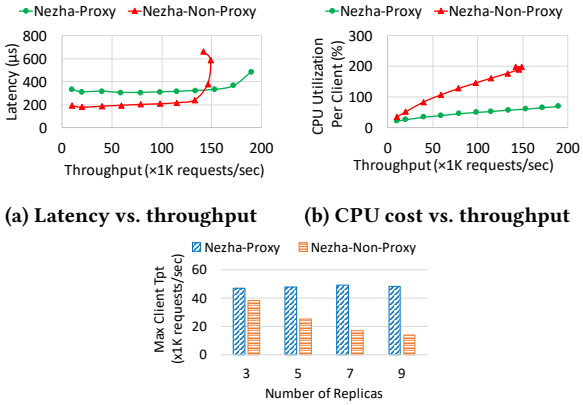
(2) The No-QC-Offloading variant still uses DOM for ordering and request multicast, but it relies on the leader to do quorum check for every request. Therefore, the leader's burden becomes much heavier than the full protocol, and the heavy bottleneck at the leader replica degrades the throughput and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



(a) Closed-loop

(b) Open-loop

Figure 9: Max throughput vs. number of replicas



(a) Latency vs. throughput

(b) CPU cost vs. throughput

(c) Max. client throughput

Figure 10: Proxy Evaluation

(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 32 %.

8.5 Scalability

Figure 9 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 9b), 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 high rates. By contrast, when equipped with proxies, Nezha-Proxy maintains a high throughput regardless of the number of replicas. We continue to evaluate the proxy design in §8.6.

8.6 Proxy Evaluation

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

Figure 10c compares the maximum throughput achieved by one client with/without proxies. Given the same CPU resource [4], the throughput of the client without proxies declines distinctly as the number of replicas increases. Such bottlenecks can also occur in the other works with similar offloading design (e.g., Speculative Paxos, NOPaxos, Domino, CURP). By contrast, when equipped with proxies, the client remains a high throughput regardless of the number of replicas.

8.7 Failure Recovery

We evaluate the failure recovery as shown in Figure 11 and Figure 12. Since follower's crash and recovery do not affect the availability of Nezha, we mainly focus on the evaluation of the leader's crash and recovery. We study two aspects: (1) How long does it take for the remaining replicas to complete a view change with the new leader elected? (2) How long does it take to recover the throughput to the same level as before crash?

We maintain 3 replicas and 10 open-loop clients, and vary per-client submission rate from 1K requests/sec to 20K requests/sec, so the total submission rate varies from 10K requests/sec to 200K requests/sec. Under different submission rates, we kill the leader and measure the time cost of view change. As shown in Figure 11, the time cost grows as the submission rate increases, because there is an increasing amount of state (log) transfer to complete the view change. But the time cost of view change is generally low (150 ms-300 ms) because of the acceleration idea (§7.3) integrated in Nezha.

The time cost to recover the same throughout level (Figure 12) 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 relaunch the working threads and reinitialize the contexts; replicas need to handle clients' retried requests, which fail to be responded before crash; followers may need additional state transfer due to lagging too far behind, etc.

Based on the measured trace, we calculate the throughput every 10 ms, and plot the data points in Figure 12. Figure 12 implies that the recovery time is related to the throughput level to recover. A lower throughput level takes a shorter time to recover, and vice versa. It takes approximately 0.7 s, 1.9 s, 4.0 s, to recover to the same throughput level under the submission rate of 20K requests/sec, 100K requests/sec, 200K requests/sec, respectively. As a reference to compare, Figure 3.20 in [51] evaluates the recovery time for an industrial Raft implementation [15], which takes about 6 seconds to recover to 18K requests/sec.

[4]Every client uses one thread for request submission and another for reply handling.

8.8 Nezha vs. Raft

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

We initially use the original Raft implementation [41] (Raft-1 in Figure 13), 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 [13]. Hence, we implement and optimize Raft (Raft-2), by using the Multi-Paxos code from [25] 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 (Figure 13) and open-loop tests [9]. 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.

9 APPLICATION PERFORMANCE

Redis. Redis [45] is a typical in-memory key-value store. We choose YCSB-A [57] as the workload, which operate on 1000 keys with `HMSET` and `HGETALL`. We use 20 closed-loop clients to submit requests, which can saturate the processing capacity of the unreplicated Redis. Figure 14 illustrates the maximum throughout of each protocol under 10 ms 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 [12] is a research fair-access financial exchange system for public cloud. There are three roles involved in CloudEx: matching engine, gateways and market participants. To provide fault tolerance, we replicate the matching engine and co-locate one gateway with one proxy. Market participants are unmodified. Before porting it to Nezha, we improved the performance of CloudEx, compared with the version in [12], by multithreading and replacing ZMQ [60] with raw UDP transmission. We first run the unreplicated CloudEx with its dynamic delay bounds (DDP) strategy disabled [12]. We configure a fixed sequencer delay parameter (d_s) of 200μs. Similar to [12], we launch a cluster including 48 market participants and 16 gateways, with 3 participants attached to one gateway. The matching engine is configured with 1 shard and 100 symbols. We vary the order submission rate of market participants, and find the matching engine is saturated at 43.10K orders/sec, achieving an inbound unfairness ratio of 1.49%.

We then run CloudEx atop the four protocols with the same setting. In Figure 15, only Nezha reaches the throughput (42.93K orders/sec) to nearly saturate the matching engine, and also yields

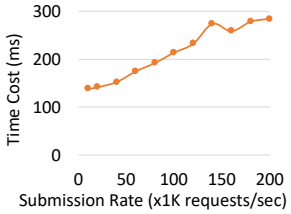
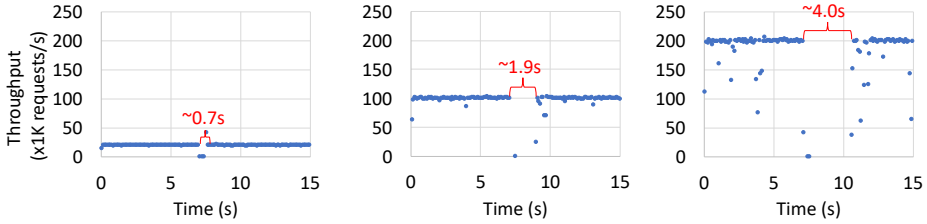


Figure 11: Time cost of view change



(a) 20K requests/sec **(b) 100K requests/sec** **(c) 200K requests/sec**

Figure 12: Time cost to recover to the same throughput level

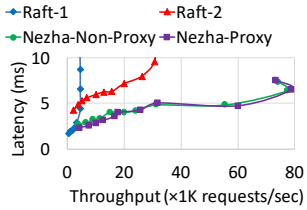


Figure 13: Nezha vs. Raft (closed-loop)

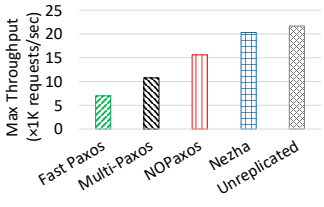


Figure 14: Redis throughput with a 10 ms latency SLO

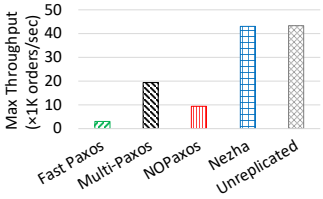


Figure 15: CloudEx throughput

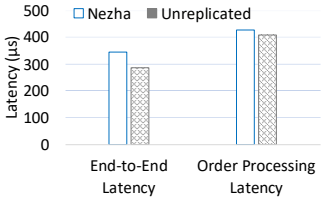


Figure 16: CloudEx latency

a close inbound unfairness ratio of 1.97%. We further compare the end-to-end latency (i.e., from order submission to the order confirmation from the matching engine) and order processing latency (i.e., from order submission to receiving the execution result from the matching engine.) between Nezha and the unreplicated CloudEx. In Figure 16, Nezha prolongs the end-to-end latency by 19.7 % (344 µs vs. 288 µs), but achieves very close order processing latency to the unreplicated version (426 µs vs. 407 µs).

10 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. Mencius [32] 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 [39] can achieve optimal WAN latency in the fast path, but when it comes to the LAN scenarios we focus on, it performs worse than Multi-Paxos [2]. CURP [42] 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 [3], BPaxos [55] and Compartmentalized Paxos [35] address the throughput scaling of consensus protocols with modularity, trading more latency for throughput improvement. The proxy design in Nezha is similar to compartmentalization [35], but Nezha's proxies are stateless. By contrast, [3, 35, 55] 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 closely related to DOM, namely, mostly-ordered multicast (MOM) [44], ordered unreliable multicast (OUM) [26], timestamp-ordered queuing (TOQ) [50] and sequenced broadcast (SB) [48]. 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 [9]. 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 is in the order of seconds.

Clock synchronization applied to consensus protocols. CRaft [51] and CockRoachDB [49] 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 [28, 30, 31]. Domino [59] and TOQ [50] try using clock synchronization to accelerate Fast Paxos and EPaxos respectively. We evaluate and compare them with Nezha in §8.3, and include more detail in [9].

11 CONCLUSION AND FUTURE WORK

We present Nezha, a high-performance consensus protocol, which can be easily deployed in the public cloud. Nezha uses a new multicast primitive called deadline-ordered multicast that leverages high-accuracy clock synchronization. We are considering three lines of future work. First, we intend to replace the Multi-Paxos/Raft backend used by industrial systems (e.g., Kubernetes, Apache Pulsar, etc) so as to boost their performance. Second, although we target LAN scenarios in this paper, applying Nezha in WAN will be an interesting area for future work. Third, we believe DOM can also be applied to other domains. We plan to integrate DOM with concurrency control algorithms (e.g., Two-Phase Locking, Optimistic Concurrency Control) to improve their performance.

REFERENCES

[1] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. 2020. Microsecond Consensus for Microsecond Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 599–616. https://www.usenix.org/conference/osdi20/presentation/aguilera

[2] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. 2019. Dissecting the Performance of Strongly-Consistent Replication Protocols. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1696–1710. https://doi.org/10.1145/3299869.3319893

[3] Martin Biely, Zarko Milosevic, Nuno Santos, and André Schiper. 2012. S-Paxos: Offloading the Leader for High Throughput State Machine Replication. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*. 111–120. https://doi.org/10.1109/SRDS.2012.66

[4] 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

[5] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. 2020. Overload Control for µs-scale RPCs with Breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 299–314. https://www.usenix.org/conference/osdi20/presentation/cho

[6] 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* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/2517349.2522712

[7] Ben Darnell. [n.d.]. Scaling Raft. https://www.cockroachlabs.com/blog/scaling-raft.

[8] etcd. [n.d.]. Benchmarking etcd v3. https://etcd.io/docs/v3.5/benchmarks/etcd-3-demo-benchmarks/.

[9] Jinkun Geng, Anirudh Sivaraman, Balaji Prabhakar, and Mendel Rosenblum. 2022. Nezha: Deployable and High-Performance Consensus Using Synchronized Clocks [Technical Report]. https://gitlab.com/steamgjk/nezhav2/-/blob/main/docs/Nezha-technical-report.pdf

[10] Yilong Geng. 2018. *Self-Programming Networks: Architecture and Algorithms*. Ph.D. Dissertation. Stanford University. https://www.proquest.com/dissertations-theses/self-programming-networks-architecture-algorithms/docview/2438700930/se-2?accountid=14026

[11] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. 2018. Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, USA) *(NSDI'18)*. USENIX Association, Berkeley, CA, USA, 81–94.

[12] Ahmad Ghalayini, Jinkun Geng, Vighnesh Sachidananda, Vinay Sriram, Yilong Geng, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. 2021. CloudEx: A Fair-Access Financial Exchange in the Cloud. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. 8. https://doi.org/10.1145/3458336.3465278

[13] Google. [n.d.]. Storage options. https://cloud.google.com/compute/docs/disks.

[14] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. *SIGMOD Rec.* 23, 2 (may 1994), 243–252. https://doi.org/10.1145/191843.191886

[15] HashiCorp. [n.d.]. HashiCorp Raft. https://github.com/hashicorp/raft. Accessed: 2022-03-31.

[16] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

[17] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable Message Latency in the Cloud. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 435–448. https://doi.org/10.1145/2829988.2787479

[18] Theo Jepsen, Stephen Ibanez, Gregory Valiant, and Nick McKeown. 2022. From Sand to Flour: The Next Leap in Granular Computing with NanoSort. *arXiv preprint arXiv:2204.12615* (2022).

[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*.

[20] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for microsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[21] Jan Kończak, Paweł T. Wojciechowski, Nuno Santos, Tomasz Żurkowski, and André Schiper. 2021. Recovery Algorithms for Paxos-Based State Machine Replication. *IEEE Transactions on Dependable and Secure Computing* 18, 2 (2021), 623–640. https://doi.org/10.1109/TDSC.2019.2926723

[22] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19 (October 2006), 79–103. https://www.microsoft.com/en-us/research/publication/fast-paxos/

[23] Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

[24] Collin Lee and John Ousterhout. 2019. Granular Computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 149–154. https://doi.org/10.1145/3317550.3321447

[25] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. [n.d.]. NOPaxos Code Repository. https://github.com/UWSysLab/NOPaxos.

[26] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA.

[27] Yilong Li, Seo Jin Park, and John Ousterhout. 2021. MilliSort and MilliQuery: Large-Scale Data-Intensive Computing in Milliseconds. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 593–611. https://www.usenix.org/conference/nsdi21/presentation/li-yilong

[28] Barbara Liskov. 1991. Practical Uses of Synchronized Clocks in Distributed Systems. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing* (Montreal, Quebec, Canada) *(PODC '91)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/112600.112601

[29] Barbara Liskov and James Cowling. 2012. Viewstamped replication revisited. (2012).

[30] Jennifer Lundelius and Nancy Lynch. 1984. A New Fault-Tolerant Algorithm for Clock Synchronization. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, British Columbia, Canada) *(PODC '84)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/800222.806738

[31] Jennifer Lundelius and Nancy Lynch. 1984. An upper and lower bound for clock synchronization. *Information and Control* 62, 2 (1984), 190–204. https://doi.org/10.1016/S0019-9958(84)80033-9

[32] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 369–384.

[33] Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres. 2017. Recovering Shared Objects Without Stable Storage. In *31st International Symposium on Distributed Computing (DISC 2017)*, Vol. 91. 36:1–36:16. https://doi.org/10.4230/LIPIcs.DISC.2017.36

[34] Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres. 2017. *Recovering Shared Objects Without Stable Storage [Extended Version]*. Technical Report. University of Washington. https://doi.org/UW-CSE-17-08-01

[35] Whittaker Michael, Ailijiang Ailidani, Charapko Aleksey, Demirbas Murat, Giridharan Neil, Hellerstein Joseph, Howard Heidi, Stoica Ion, and Szekeres. Adriana. 2021. Scaling Replicated State Machines with Compartmentalization. *Proc. VLDB Endow.* (2021), 12.

[36] Microsoft. [n.d.]. Global data distribution with Azure Cosmos DB-under the hood. https://docs.microsoft.com/en-us/azure/cosmos-db/global-dist-under-the-hood.

[37] Microsoft. [n.d.]. NIC series. https://docs.microsoft.com/en-us/azure/virtual-machines/nc-series.

[38] Jeffrey C. Mogul and Ramana Rao Kompella. 2015. Inferring the Network Latency Requirements of Cloud Tenants. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland. https://www.usenix.org/conference/hotos15/workshop-program/presentation/mogul

[39] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 358–372. https://doi.org/10.1145/2517349.2517350

[40] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 517–532. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/mu

[41] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[42] Seo Jin Park and John Ousterhout. 2019. Exploiting Commutativity for Practical Fast Replication *(NSDI'19)*. USENIX Association, USA, 47–64.

[43] PingCap. [n.d.]. TiKV-Data Sharding. https://tikv.org/deep-dive/scalability/data-sharding.

[44] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI'15)*. USENIX Association, USA, 43–57.

[45] Redis Enterprise. [n.d.]. Redis. https://redis.io.

[46] Tollman Sarah. [n.d.]. TOQ-based EPaxos Repository. https://github.com/PlatformLab/epaxos.

[47] Fred B. Schneider. 1993. *Replication Management Using the State-Machine Approach.* ACM Press/Addison-Wesley Publishing Co., USA, 169–197.

[48] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. 2022. State Machine Replication Scalability Made Simple. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 17–33. https://doi.org/10.1145/3492321.3519579

[49] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

[50] Sarah Tollman, Seo Jin Park, and John Ousterhout. 2021. EPaxos Revisited. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 613–632. https://www.usenix.org/conference/nsdi21/presentation/tollman

[51] Feiran Wang. 2019. *Building High-performance Distributed Systems with Synchronized Clocks.* Ph.D. Dissertation. Stanford University. https://www.proquest.com/dissertations-theses/building-high-performance-distributed-systems/docview/2467863602/se-2?accountid=14026

[52] Zhaoguo Wang, Changgeng Zhao, Shuai Mu, Haibo Chen, and Jinyang Li. 2019. On the Parallels between Paxos and Raft, and How to Port Optimizations. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. Association for Computing Machinery, New York, NY, USA, 445–454. https://doi.org/10.1145/3293611.3331595

[53] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *SIGOPS Oper. Syst. Rev.* 35, 5 (oct 2001), 230–243. https://doi.org/10.1145/502059.502057

[54] Michael Whittaker, Neil Giridharan, Adriana Szekeres, Joseph M Hellerstein, Heidi Howard, Faisal Nawab, and Ion Stoica. 2020. Matchmaker paxos: A reconfigurable consensus protocol [technical report]. *arXiv preprint arXiv:2007.09468* (2020).

[55] Michael Whittaker, Neil Giridharan, Adriana Szekeres, Joseph M Hellerstein, and Ion Stoica. 2020. Bipartisan paxos: A modular state machine replication protocol. *arXiv preprint arXiv:2003.00331* (2020).

[56] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. 2013. Bobtail: Avoiding Long Tails in the Cloud. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 329–341. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/xu_yunjing

[57] Yahoo! [n.d.]. YCSB Workload. https://github.com/brianfrankcooper/YCSB/tree/master/workloads.

[58] Xinan Yan. [n.d.]. Domino Repository. https://github.com/xnyan/domino.

[59] Xinan Yan, Linguan Yang, and Bernard Wong. 2020. Domino: Using Network Measurements to Reduce State Machine Replication Latency in WANs. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies* (Barcelona, Spain) *(CoNEXT '20)*. Association for Computing Machinery, New York, NY, USA, 351–363. https://doi.org/10.1145/3386367.3431291

[60] ZeroMQ community. [n.d.]. ZeroMQ. https://zeromq.org/. Accessed: 2021-02-02.