Nezha: Deployable and High-Performance Consensus Using Synchronized Clocks

Jinkun Geng
Stanford University
gjk1994@stanford.edu

Anirudh Sivaraman
New York University
anirudh@cs.nyu.edu

Balaji Prabhakar
Stanford University
balaji@stanford.edu

Mendel Rosenblum
Stanford University
mendel@stanford.edu

ABSTRACT

This paper presents a high-performance consensus protocol, Nezha, which can be deployed by cloud tenants without support from cloud providers. 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, (optimized) NOPaxos, 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% of 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.

PVLDB Reference Format:
Jinkun Geng, Anirudh Sivaraman, Balaji Prabhakar, and Mendel Rosenblum. Nezha: Deployable and High-Performance Consensus Using Synchronized Clocks . PVLDB, 16(4): 629 - 642, 2022.
doi:10.14778/3574245.3574250

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made 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 generally deployed by cloud tenants with no help from their cloud providers. 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.

Proceedings of the VLDB Endowment, Vol. 16, No. 4 ISSN 2150-8097.
doi:10.14778/3574245.3574250

First, modern databases (e.g., Cosmos DB, TiKV and CockroachDB) aim 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, 31, 39], thereby losing consistency guarantees over the whole data. Second, microsecond-scale applications are pushing the limits of computing [1, 16, 18]. 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 in 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 [22] and Raft [36] 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 [24], Speculative Paxos [40], NetChain [17], NetPaxos [17], 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 Nezha to provide high performance for 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 latency, and a fallback slow path that suffers from a higher latency. Examples include Fast Paxos [21], EPaxos [34], Speculative Paxos [40], NOPaxos [24], etc.

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 packet reordering in public cloud. DOM is a type of multicast that works as follows. The senders' and receivers' clocks are synchronized to each other to produce a *shared* global time. The

[1] Many of these technologies are available to *cloud providers*, but not exposed to cloud tenants. RDMA instances [32] are an exception, but such instances are expensive.

sender attaches a deadline in global time to its message and multicasts [2] 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 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, Nezha still maintains correct by falling back to the slow path. Here, DOM follows Liskov's suggestion of "depending on clock synchronization for performance but not for correctness" [25].

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 the execution of a request from committing the request. Protocols like Multi-Paxos/Raft wait until the request is committed at a quorum of servers before executing the request at 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. 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 proxies fault tolerant.

Evaluation. We compare Nezha to six baselines in the public cloud: Multi-Paxos, Fast Paxos, (optimized) NOPaxos, Raft, Domino and TOQ-EPaxos under closed-loop and open-loop workloads. In closed-loop workloads, commonly used in literature [24, 30, 34, 40], 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 [46], clients submit new requests according to a Poisson process, without waiting for replies for previous requests. 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, and 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.

[2]Unless otherwise specified, "multicast" in this paper refers to application-based multicast, because switch-based multicast is not supported in cloud environment.

(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 use Nezha to replicate two applications (Redis and CloudEx [12]) and show that Nezha can provide fault tolerance with modest degradation: compared with the unreplicated system, Nezha sacrifices 5.9% throughput for Redis; it saturates the capacity of CloudEx and prolongs the order processing latency by 4.7%.

2 MOTIVATION OF NEZHA: REORDERING IN THE PUBLIC CLOUD

Consensus protocols are often used to provide the abstraction of a replicated state machine (RSM) [43], where multiple servers/replicas 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 them 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 has 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 [21, 24, 40, 53]. 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 [21, 40] 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) [19, 42] in R_2's sequence, and the *reordering score* is calculated as:

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

Figure 1: Reordering vs. submission rate on Google Cloud

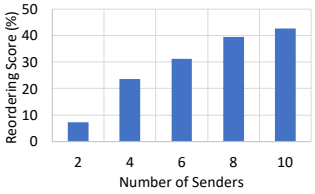


Figure 2: Reordering vs. number of senders on Google Cloud

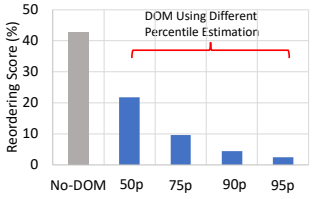


Figure 3: Effectiveness of DOM on reordering on Google Cloud

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.

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 (§3). Our consensus protocol, Nezha, depends on DOM to guarantee consistent ordering among replicas, but not set equality. This is intentional and is also why we need further handling (§4–§5) in Nezha to go along with DOM to eventually achieve both requirements.

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 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 DOM 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 their deadlines (i.e., $D(M)$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 deadlines, 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 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 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. Since 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 [25].

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.

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

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 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 [11], the standard deviation for the sending time and receiving time, denoted as σ_S and σ_R. [3] 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 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 OWDs will be piggybacked/sent to the sender to decide the deadlines of subsequent requests.

[3] σ values are calculated based on the method in [10][Appendix A]. $\beta = 3$ in our setting, i.e., a 3σ confidence interval for the sending/receiving time provided by Huygens.

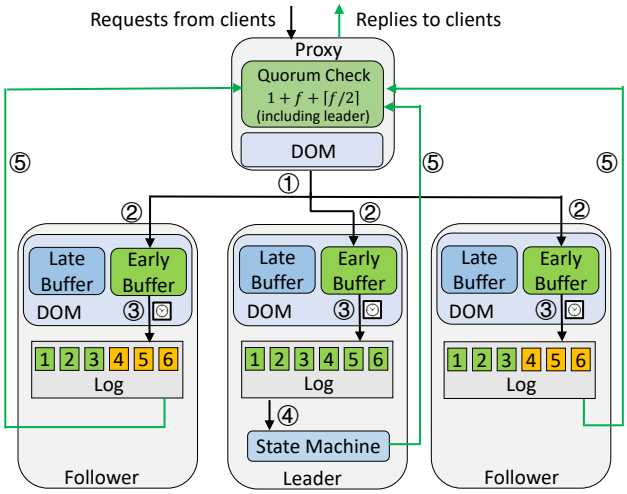


Figure 4: Fast path of Nezha

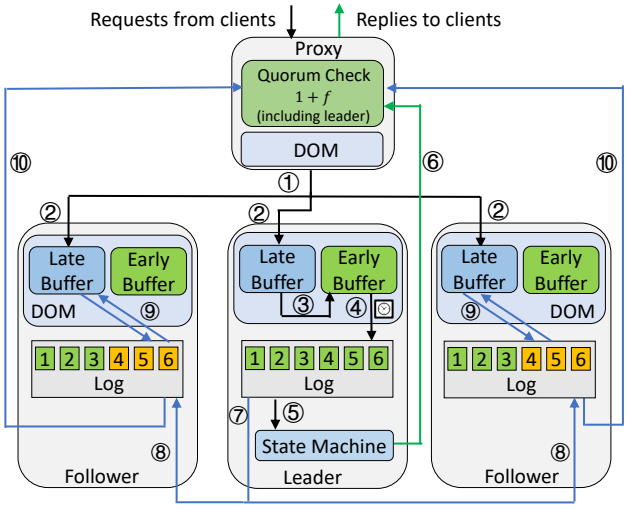


Figure 5: Slow path of Nezha

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 causes lower reordering. However, a higher percentile also causes longer holding delay for messages in DOM, which in turn decreases the latency savings of Nezha (as evaluated in §8.5) .

4 NEZHA OVERVIEW

We use DOM as a building block to develop a consensus protocol, called Nezha, atop DOM. 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 crash simultaneously. Nezha guarantees safety (linearizability) 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" [37]). Nezha's performance is improved by DOM, whose effectiveness depends on accurate clock synchronization and the variance of proxy-to-replica OWDs. But Nezha does not assume the existence of a worst-case clock error bound, because clock synchronization can also fail [25, 27, 28].

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. Besides, only the proxies (DOM senders) and replicas (DOM receivers) need clock synchronization, whereas clients can remain unsynchronized.

Fast path sketch. Figure 4 shows Nezha's 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 at the deadline, and appended to replicas' *logs* ③. The *log* of requests is ordered by request deadline. After that, followers 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 obtains the execution result from the leader's reply, and then forwards 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: without leader-follower communication, a simple quorum cannot persist sufficient information for a new leader to distinguish committed requests from uncommitted ones (details in §5.3).

Slow path sketch. 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 *unique* 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 logs to stay consistent with the leader: as an optimization, followers can retrieve missing requests from their own *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., [21, 24, 40]): it also decouples the request execution (at the leader) and quorum check in the slow path. Such a decoupling design enables a *faster* slow path for the

Algorithm 1 Replica Actions

Member Variables: ▷
eb, ▷ *early-buffer*
lb, ▷ *late-buffer*
synced-log, ▷ the replica's *log* which has been synced with leader
unsynced-log, ▷ the replica's *log* which hasn't been synced with leader
replica-id, *view-id*, *status*, f ▷ Other state variables

1: **upon** RECEIVE $request$ **do**
2: $lastReq \leftarrow$ the last released request from eb.
3: **if** $request.deadline > lastReq.deadline$[4] **then**
4: eb.**insert**($request$) ▷ eb is a priority queue
5: **else**
6: lb.**insert**($request$) ▷ lb is a map
7: **if** *replica-id* = *view-id* % $(2f+1)$ **then** ▷ It is leader
8: $newDdl$ = **max**(CLOCKTIME(), $lastReq.deadline + 1$)
9: $request.deadline = newDdl$ ▷ Modify its deadline
10: eb.**insert**($request$) ▷ Can enter eb with the new deadline
11: **upon** eb.**empty**()=**false and** eb.**top**().$deadline \leq$ CLOCKTIME() **do**
12: $request$ =eb.**top**() ▷ The request to be released from eb
13: eb.**erase**(request)
14: **if** *replica-id* = *view-id* % $(2f+1)$ **then** ▷ Replica is leader
15: $result$ =EXECUTE($request$) ▷ Only leader executes request
▷ Leader directly appends $request$ and $result$ to *synced-log*
16: *synced-log*.**append**({$request$, $result$})
17: *hash*=CALCINCREMENTHASH(*synced-log*)
18: SENDFASTREPLY(*result*, *hash*)
▷ In parallel with sending *fast-reply*, leader conducts broadcast
19: BROADCASTLOGMODIFICATION(*request*)
20: **else**
▷ Follower appends $request$ to *unsynced-log* without execution
21: *unsynced-log*.**append**({$request$, *null*})
22: *synced-hash*=CALCINCREMENTHASH(*synced-log*)
23: *unsynced-hash*=CALCINCREMENTHASH(*unsynced-log*)
▷ Follower's hash is generated by concatenating the two parts.
24: SENDFASTREPLY(*null*, *synced-hash* **XOR** *unsynced-hash*)
▷ Only followers will receive $log\text{-}modification$ messages
25: **upon** RECEIVE $log\text{-}modification$ **do**
▷ Both synced part and unsynced part will be modified, details at §5.4
26: MODIFYSYNCED(*synced-log*, *synced-hash*)
27: MODIFYUNSYNCED(*unsynced-log*, *unsynced-hash*)
28: SENDSLOWREPLY() ▷ Only followers send *slow-reply*

Algorithm 2 Proxy Actions

Member Variables: ▷
f, ▷ the number of replicas is *2f+1*
replySet, ▷ the set of replies received from replicas

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**
7: **return**
8: **if** $reply$ **is from new** $view$ **then**
▷ Replicas experienced view change, all previous replies are stale
9: *replySet*.**clear**()
10: *replySet*.**insert**(*reply*)
11: $committedReply$= CHECKCOMMITTED($reply$)
12: **if** $committedReply \neq null$ **then**
13: REPLYTOCLIENT($committedReply$)
14: **function** CHECKCOMMITTED(*reply*)
▷ If the proper quorum is established, return the leader's reply because it contains the exeuction result
15: $quorum = \{msg \in replySet : msg.view\text{-}id = reply.view\text{-}id \ \& \ msg.client\text{-}id = reply.client\text{-}id \ \& \ msg.request\text{-}id = reply.request\text{-}id\}$
16: *leader-id* = *reply.view-id % (2f+1)*
17: **if** *quorum* **not contains** *replica leader-id*'s *fast-reply* **then**
18: **return** *null* ▷ Leader's *fast-reply* must be included
19: *fast-reply-num, slow-reply-num* = 0, 0
20: **for** $r \leftarrow 0$ to $2f$ **do**
21: **if** *quorum* **contains** *replica r*'s *slow-reply* **then**
22: *slow-reply-num++*
▷ *slow-reply* can serve as *fast-reply*, but not the opposite
23: *fast-reply-num++*
24: **else if** *quorum* **contains** *replica r*'s *fast-reply* **and** *replica r*'s *fast-reply.hash= replica leader-id*'s *fast-reply.hash* **then**
25: *fast-reply-num++*
26: **if** *fast-reply-num* $\geq 1 + f + \lceil f/2 \rceil$ **then**
27: **return** *replica leader-id*'s *fast-reply* ▷ Committed in fast path
28: **if** *slow-reply-num* $\geq f$ **then** ▷ We also have leader's *fast-reply*
29: **return** *replica leader-id*'s *fast-reply* ▷ Committed in slow path

proxy to commit requests, which is only one message delay longer than the commit in the fast path. The proxy, through the quorum check, can ensure that the speculative execution result from the leader replica is safe to use.

5 THE NEZHA PROTOCOL

We first describe the state maintained by Nezha, Nezha's message formats and Nezha's fast and slow path. In addition, Algorithms 1 and 2 describe the replica and proxy algorithms with object-oriented and event-driven methods.

5.1 Replica State

Below we summarize the state variables maintained by each replica.

replica-id: Each replica is initially assigned with a unique *replica-id*, ranging from 0 to $2f$.

view-id: Replicas leverage a view-based approach [26]: 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 statuses: NORMAL, VIEWCHANGE and RECOVERING. Replicas are initially launched in NORMAL statuses. They switch to VIEWCHANGE when the leader fails, and switch back to NORMAL after the new leader is elected. A failed replica rejoins the system in RECOVERING status and switch to NORMAL after recovering its state from the other replicas.

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

late-buffer: *late-buffer* is a map indexed by *<client-id, request-id>*. It holds those requests which are not eligible to enter the *early-buffer*, because those requests may later be needed in the slow

[4]When deadlines are equal, the tie is broken by *<client-id,request-id>*.

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. These requests then become the entries in the *log*. The *log* is ordered by requests' deadlines.

sync-point: Followers modify their *log*s to stay consistent with the leader (§5.4). *sync-point* indicates the log position up to which this replica's *log* is consistent (i.e., equal) 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

We explain five types of messages closely related to Nezha. Since Nezha uses a view-based approach for leader change, we omit the description of messages related to leader changes; these messages have been defined in Viewstamped Replication [26].

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:

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

client-id represents the client identifier and *request-id* is assigned by the client to uniquely identify its own request. *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. 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 it has 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*. The proxy can recognize the leader's reply by checking whether its *replica-id* equals *view-id*%$(2f+1)$. *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 replicas involved in a quorum have consistent *log*s.

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 *log*s to stay 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. *deadline* is the request's deadline shown in the leader's *log*, which is either assigned by proxies on the fast path (i.e., ① in Figure 4) 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 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:

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

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

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

$$log\text{-}status=<view\text{-}id, replica\text{-}id, sync\text{-}point>$$

5.3 Fast Path

Nezha relies on DOM to increase the frequency of its fast path (i.e. fast commit ratio, or FCR). As shown earlier (§2), the percentile DOM uses to estimate OWDs is set by the DOM user. A lower percentile sets smaller deadlines, which reduces fast path latency (FPL), but also reduces FCR. Higher percentiles have the opposite problem. We use the 50^{th} percentile in Nezha to strike a balance. This does reduce FCR compared with using a higher percentile; hence, Nezha compensates for this by accelerating its slow path (§5.4) and leveraging commutativity (§7.2).

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.

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. 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 [14]: the client sees *request-1* executed before *request-2* with the old leader but 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 receiving 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 be larger than the last released request's deadline ③. 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 ④, applying it to the state machine ⑤, and sending *fast-reply* ⑥).

Leader broadcasts log-modification. In parallel with ⑤-⑥, the leader also broadcasts a *log-modification* message to followers ⑦ after appending each request. Every time a follower receives a *log-modification* ⑧, it checks its log entry at the position *log-id* included in the *log-modification*. (1) If the entry has the same 3-tuple <*client-id*, *request-id*, *deadline*> as that included in the *log-modification*, 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*, 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*. (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*>. 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* 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 *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 §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 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 (§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. Besides, the *log-status* messages only serve for an optional optimization to accelerate recovery. The correctness of Nezha will not be affected even if all *log-status* messages are lost.

5.5 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 [5, 20].

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.

6 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. 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/removing replicas. Both cases are handled by reconfigurable consensus [26, 49], which 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. We explain the details in [9][Appendix A] and only sketch the major steps for the new leader to recover its *log*.

After the new leader is elected, 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 *log* entries which have not been copied to the new leader, the new leader checks each of them as follows: if the entry has larger *deadline* than the one located at *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 from scratch, or from the latest checkpoint if the periodic checkpoint mechanism (§7.3) has been enabled. 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. To maintain correctness, the deposed leader obtains the correct new state from another replica in the fresh view.

We have included the evaluation of failure recovery in [9].

Correctness proof. Our technical report [9][Appendix B] proves that Nezha maintains 3 correctness properties for consensus: durability, consistency and linearizability. We modeled the whole protocol in TLA+ [9] and checked the 3 properties in TLA+ Toolbox.

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 their deadlines 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, 38]. 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*.

7.3 Periodic Checkpoints

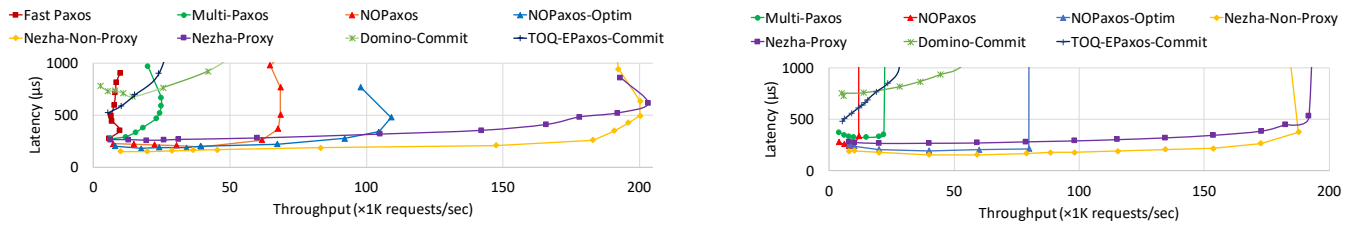
To (1) accelerate the recovery process after leader failure and (2) enable a deposed leader to quickly catch up with the fresh state, we can integrate a periodic checkpoint mechanism into Nezha. Since Nezha only allows the leader to execute requests during normal processing, it can lead to inefficiency during leader change, either caused by leader's failure or network 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 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 to stable storage 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 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 §6), it can start from its latest checkpoint state, and only retrieve from another replica (in the fresh view) requests beyond its *commit-point*.

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? (§8.2)



(a) Closed-loop workload **(b) Open-loop workload**

Figure 6: Latency vs.throughput

(2) How does Nezha compare to the recent protocols which also use clock synchronization (i.e., Domino and TOQ-EPaxos)? (§8.3)

(3) How effective are the proxies in saving clients' CPU cost, especially when there is a large number of replicas? (§8.6)

(4) How does Nezha compare to Raft when both are equipped with log persistence to stable storage? (§8.8)

(5) Does Nezha provide sufficient performance for replicated applications? (§9)

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 the NOPaxos sequencer, and `n1-standard-32` VMs for Nezha proxies. All VMs are in a single cloud zone (except §8.7). Huygens is installed on all VMs and has an average 99th 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 [23] with necessary modification: (1) we change switch multicast into multiple unicasts because switch multicast is unavailable in cloud. (2) we use a software sequencer (refer to §4.2.3 in [24]) with multi-threading for NOPaxos because tenant-programmable switches are not yet available in cloud. We also add two recently proposed protocols that leverage synchronized clocks for comparison, i.e., Domino [53] and TOQ-EPaxos [46].

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 [15, 33, 51]. We run each experiment 5 times and average values before plotting.

Evaluation method. We follow the method of NOPaxos [24] and run a *null application* with no execution logic. Traditional evaluation [24, 30, 34, 35, 40, 48] uses 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 [46] 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. 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 [13] 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

We plot two versions of Nezha in Figure 6. 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 [24]. Fast Paxos suffers the most and reaches only 4.0K requests/second at 425 µs in open-loop test (not shown in Figure 6b). When clients send at a higher rate, the frequent request reordering forces 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 sequence 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 6) by using separate threads for the two tasks. NOPaxos-Optim outperforms all the other baselines because it offloads request serialization to the sequencer and quorum check (fast path) to clients. But it still loses significant throughput in the open-loop test compared with the closed-loop test. This is because open-loop tests create more bursts of requests, and cause packet reordering/drop more easily. 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 execution 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 §8.4.

8.3 Comparison with TOQ-EPaxos

We should have plotted the execution latencies of Domino and TOQ-EPaxos in Figure 6. However, their execution latencies are far larger than Nezha and the other baselines and are not suitable to be put in the same figure, so we plot their commit latencies and still find that Nezha performs better. We believe this is partially related to their different implementations (e.g., Go vs. C++). Below, we compare Nezha with TOQ-EPaxos from a design perspective. [9] includes the comparison between Nezha and Domino design.

(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 [46]). 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 (see §8.7).

(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 [2], even implemented under the same framework, EPaxos is less performant than Multi-Paxos in LAN. By contrast, Nezha is a generally high-performance protocol that yields good performance in both LAN (Figure 6) and WAN (Figure 10) settings.

8.4 Ablation Study

During the ablation study, 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 7. No-DOM removes the DOM primitive from Nezha. No-QC-Offloading relies on the leader to do the quorum check, and it still relies on DOM for consistent ordering (the proxies still perform request multicast). No-Commutativity disables Nezha's commutativity optimization. We run all protocols under the same setting as Figure 6b. Figure 7 shows that, removing any of the three components degrades the performance (i.e., throughput and/or latency).

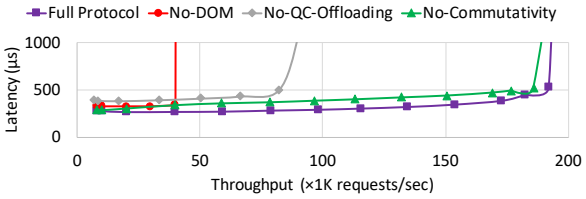


Figure 7: Ablation study of Nezha

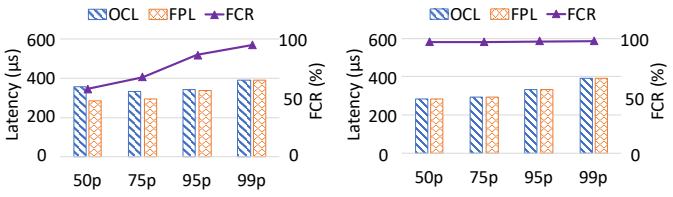
(1) The No-DOM variant makes the fast path meaningless: requests are no longer ordered by their deadlines without DOM, so consistent ordering is not guaranteed and set equality (i.e. reply messages with consistent hash) no longer indicates the state consistency among replicas. In this case, the No-DOM 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 %.

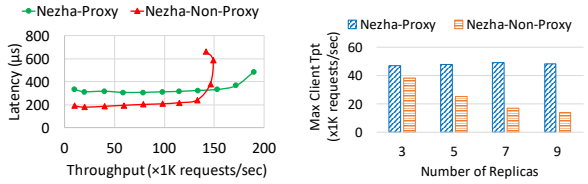
8.5 DOM's Trade-Off at Different Percentiles

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)



(a) Without Commutativity **(b) With Commutativity**

Figure 8: Trade-off of using different percentiles in DOM



(a) Latency vs. throughput **(b) Max. client throughput**

Figure 9: Proxy Evaluation

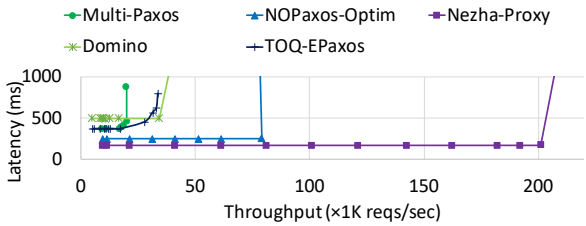


Figure 10: Latency vs. throughput in WAN

FPL, but both FCR and FPL affect the the overall commit latency (OCL) *of all requests*.

To measure this, in Figure 8 we run Nezha with/without commutativity optimization in the open-loop test (20K requests/second).

(1) Without commutativity optimization (Figure 8a), as we use larger percentiles (from 50p to 75p), the improvement of FCR 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 8b), Nezha already reaches a high FCR using 50p. Therefore, using larger percentiles brings little FCR improvement, but only increases FPL, and in turn OCL. Therefore, we choose 50p in DOM and have verified its robustness under different workloads (details in [9]).

8.6 Proxy Evaluation

Figure 9a compares the two versions of Nezha with 10 open-loop clients and 9 replicas. Nezha-Proxy employs 5 proxies. As clients increases their submission rates. 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 than Nezha-Proxy as the throughput grows, because Nezha-Non-Proxy makes the clients CPU-intensive.

Figure 9b compares the maximum throughput achieved by one client with/without proxies. Given the same CPU resource, [5] the throughput of the client without proxies declines distinctly as the number of replicas increases. Such bottlenecks can also occur in other protocols with similar offloading design (e.g., [24, 38, 40, 53]). By contrast, when equipped with proxies, the client retains a high throughput regardless of the number of replicas.

8.7 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 6), Nezha outperforms all the baselines even more in WAN. As shown in Figure 10, 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 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 of [9].

8.8 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 compare with two versions of Raft in Figure 11: Raft-1 is the open-sourced implementation from [36]. Raft-2 is our own implementation by using Multi-Paxos code from [23] as a starting point. Our evaluation shows that Nezha outperforms Raft in both closed-loop (Figure 11) and open-loop tests [9]. Besides, 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

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 [41] is a typical in-memory key-value store. We choose YCSB-A [52] as the workload, which operates on 1000 keys with `HMSET` and `HGETALL`. We use 20 closed-loop clients to saturate the processing capacity of the unreplicated Redis. Figure 12 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 [12] is a fair-access financial exchange system for the public cloud, which includes three roles: matching engine, gateways and market participants. We replicate the

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

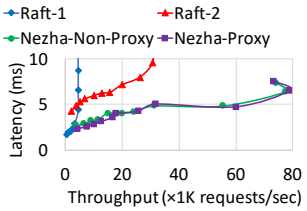


Figure 11: Nezha vs. Raft (closed-loop)

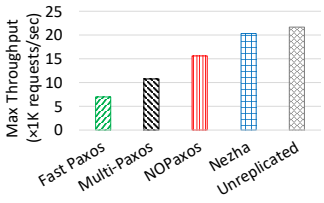


Figure 12: Redis throughput with a 10 ms latency SLO

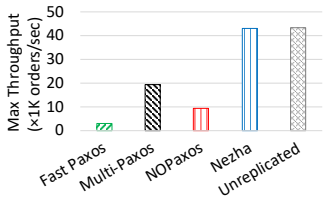


Figure 13: CloudEx throughput

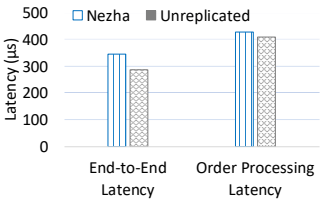


Figure 14: CloudEx latency

matching engine and co-locate one gateway with one proxy. Market participants are unmodified. Compared with the version in [12], we have improved the performance of unreplicated CloudEx with multi-threading and more efficient communication. On the matching engine, we configure 1 shard with 100 symbols, and use a fixed sequencer delay parameter (d_s) of 200μs. Similar to [12], we launch 48 participants and 16 gateways, with 3 participants attached to one gateway. 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 13, 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 14, 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, such as Multi-Paxos, Raft and Viewstamped Replication, make no distinction between fast and slow paths, whereas Nezha uses an optimistic approach to improve latency in the common case. Mencius [29] exploits a multi-leader design to mitigate the single-leader bottleneck. However, it introduces extra coordination cost among multiple leaders, and the crash of any leaders temporarily stops progress. By contrast, Nezha reduces the leader's bottleneck using proxies and followers' crash does not affect progress. EPaxos [34] can achieve optimal WAN latency by colocating clients and some replica in the same zone, but its WAN latency benefit is nullified when replicas are separated from clients. Besides, EPaxos performs worse than Multi-Paxos in LAN [2]. CURP [38] can complete commutative requests in 1 RTT, but cost up to 3 RTTs even if all witnesses process non-commutative requests in the same order. SPaxos [3], BPaxos [50] and Compartmentalized Paxos [30] address the scaling of consensus protocols with modularity, trading more latency for better throughput. The proxy design of Nezha shares some similarity to compartmentalization [30], but Nezha's proxies are stateless whereas [3, 30, 50] use stateful components that complicates the recovery process. We include a comparison summary in [9][Table 1].

Network primitives to improve consensus. There are four other primitives in the literature closely related to DOM, namely, mostly-ordered multicast (MOM) [40], ordered unreliable multicast (OUM) [24], timestamp-ordered queuing (TOQ) [46] and sequenced broadcast (SB) [44]. 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 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 [47] and CockRoachDB [45] use clock synchronization to improve the throughput of Raft. However, they base their correctness on the assumption of a known worst-case clock error bound, which is not practical for clock synchronization [25, 27, 28]. Domino [53] and TOQ [46] try using clock synchronization to accelerate Fast Paxos and EPaxos respectively. We evaluate and compare them with Nezha in §8.3, and include more details 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. Our evaluation has shown Nezha can significantly outperform the typical baselines in public cloud. Our future works include (1) replacing the Multi-Paxos/Raft backend used by industrial systems (e.g., Kubernetes) with Nezha to boost their performance; (2) using DOM to improve the performance of the existing concurrency control algorithms (e.g., Two-Phase Locking) or invent new concurrency control protocols.

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] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. 2020. Microsecond 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

[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 *Proceedings of the 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] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2021. PigPaxos: Devouring the Communication Bottlenecks in Distributed Consensus. In *Proceedings of the 2021 International Conference on Management of Data*. Association for Computing Machinery, New York, NY, USA, 235–247. https://doi.org/10.1145/3448016.3452834

[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://arxiv.org/abs/2206.03285

[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] 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

[14] 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

[15] 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

[16] 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).

[17] 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 *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*.

[18] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for microsecond-scale Tail Latency. In *Proceedings of the 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

[19] Wayne Kevin. [n.d.]. Longest Increasing Subsequence. https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/LongestIncreasingSubsequence.pdf.

[20] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

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

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

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

[24] 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.

[25] 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

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

[27] 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

[28] 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

[29] 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.

[30] 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.

[31] 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.

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

[33] Jeffrey C. Mogul and Ramana Rao Kompella. 2015. Inferring the Network Latency Requirements of Cloud Tenants. In *Proceedings of the 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

[34] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the 24th 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

[35] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Proceedings of the 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

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

[37] John Ousterhout and Diego Ongaro. [n.d.]. Implementing Replicated Logs with Paxos. https://ongardie.net/static/raft/userstudy/paxos.pdf.

[38] Seo Jin Park and John Ousterhout. 2019. Exploiting Commutativity for Practical Fast Replication. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI'19)* (Boston, MA, USA). USENIX Association, USA, 47–64.

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

[40] 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.

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

[42] C. Schensted. 1961. Longest Increasing and Decreasing Subsequences. *Canadian Journal of Mathematics* 13 (1961), 179–191. https://doi.org/10.4153/CJM-1961-015-3

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

[44] 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

[45] 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

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

[47] 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

[48] 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

[49] 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).

[50] 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).

[51] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. 2013. Bobtail: Avoiding Long Tails in the Cloud. In *Proceedings of the 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

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

[53] 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