Appendices

A Recovery Protocol and Algorithms

We explain how Nezha leverages the diskless crash recovery algorithm [1] from Viewstamped Replication in 3 steps. First, we explain how we adopt the recent concept of crash-vectors [2, 3] to fix the incorrectness in the crash recovery algorithm. Second, we explain how a replica rejoins Nezha following a crash. Third, we describe how the leader election works if the leader crashes.

A.1 Crash Vector

Like Viewstamped Replication, Speculative Paxos and NOPaxos, Nezha also adopts diskless recovery to improve performance. However, in contrast to them, Nezha avoids the effect of *stray messages* [4] (i.e., messages that are sent out but not delivered before replica crash, so that the relaunched replicas forget them) using the *crash-vector* [2, 3]. *crash-vector* is a vector containing $2f+1$ integer counters corresponding to the $2f+1$ replicas. Each replica maintains such a vector, with all counters initialized as 0s.

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

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

(1) There can be *stray messages* during the recovery process. The stray messages cause replicas to elect a leader, whose state falls behind the others, and finally causes permanent loss of committed requests. Such error cases have been analyzed in [3] and we also illustrate it in §J.1 by taking NOPaxos as the example. The *crash-vector* prevents the *stray messages* effect because it enables the replicas to recognize potential *stray messages* by comparing a *crash-vector* received from a replica with the local *crash-vector*. During recovery, the RECOVERING replica first recovers its *crash-vector* by collecting and aggregating the *crash-vectors* from a majority of NORMAL replicas. Then, the replica increments its own counter (i.e. replica i increments the ith counter in the vector) and tags the new *crash-vector* to the messages sent afterwards. Once the update of *crash-vector* is exposed to the other replicas, they can recognize the *stray messages* sent by the replica before crash (i.e., those messages have a smaller value at the ith counter), and avoid processing those messages. Thus, the recovery will not be affected by *stray messages*.

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

A.2 Replica Rejoin

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

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

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

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

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

Step 5: After receiving the RECOVERY-REQ, replicas with NORMAL status update their own *crash-vectors* by aggregating with *cv*, obtained from the request in step 4. Then, these replicas send back a reply including their own *view-id* and

[1] There are many options available to generate the locally unique *nonce* string [1,3]. Nezha uses the universally unique identifier (UUID) (GENERATE-UUID in line 4), which have been widely supported by modern software systems.

Algorithm 1 Replica rejoin

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

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

crash-vector, i.e. <RECOVERY-REP, *view-id*, *crash-vector*> (line 54-63).

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

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

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

A.3 Leader Change

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

Step 1: When a replica fails to receive the heartbeat (i.e., *sync* message) from the leader for a threshold of time, it suspects the leader has failed. Then, it sets its *status* as VIEWCHANGE, increments its *view-id*, and broadcasts a view change request to all replicas including its *crash-vector*, i.e. <VIEW-CHANGE-REQ, *view-id*, *replica-id*, *cv*> (line 6-10) [2]. The replica switches its *status* from NORMAL to VIEWCHANGE, and enters the view change process.

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

Step 3: All replicas under view change send a message <VIEW-CHANGE, *view-id*, *log*, *sync-point*, *last-normal-view*> to the leader of the new view (*replica-id*$=$ *view-id*$\%(2f+1)$) (line 11). Here *last-normal-view* indicates the last view in which the replica's *status* was NORMAL.

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

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

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

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

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

Step 9: After the system is fully recovered, the replicas can continue to process the incoming requests. Recall in §R.3 that the incoming request is allowed to enter the *early-buffer* if its deadline is larger than *the last released request* which is not commutative. To ensure uniform ordering, the eligibility check is still required for the incoming request even if it is the first one arriving at the replica after recovery. The replica considers the entries (requests) in the recovered *log*, which are not commutative to the incoming request, and chooses the one as *the last released request* with the largest deadline among them. The incoming request can enter the *early-buffer* if its deadline than *the last released request*, otherwise, it is put into the *late-buffer*.

Note that the view change protocol chooses the leader in a round-robin way (*view-id*$\%(2f+1)$). Specially, a view change process may not succeed because the new leader also

[2] The view change request will be rebroadcast if the replica times out but is still waiting for the view change process to complete. The same is also true for the view change message described in the next step.

Algorithm 2 Leader change

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

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

fails (as mentioned in [1]). In this case (i.e. after followers have spent a threshold of time without completing the view change), followers will continue to increment their *view-id*s to initiate a further view change, with yet another leader.

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

A.4 Evaluation of Recovery

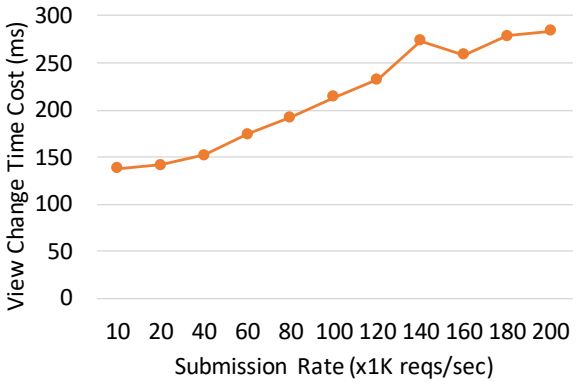


Figure 1: Time cost of view change

We evaluate the failure recovery as shown in Figure 1 and Figure 2. 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 before crash?

We maintain 3 replicas and 10 open-loop clients, and vary the client submission rate from 1K reqs/sec to 20K reqs/sec, so the total submission rate varies from 10K reqs/sec to 200K reqs/sec. Under different submission rates, we kill the leader and measure the time cost of view change, as shown in Figure 1. To mitigate the noise effect, we also run each case for 5 times and average them as the reported value. We can see from Figure 1, the time cost of view change grows as the submission rate increases, because there is an increasing amount of state (log) transfer to complete the view change. In general, the view change takes about 150 ms-300 ms.

The time cost to recover the same throughout level is larger than the time cost of view change, because there are other necessary works to do after the replicas enter the new view. For example, replicas need to relaunch the working threads and reinitialize the contexts; replicas need to notify proxies and further the clients to continue submitting requests; 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.

In Figure 2, we plot three recovery cases with different throughput levels. Based on the measured trace, we calculate the throughput every 10 ms, and plot the data points in Figure 2. Figure 2 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. Figure 2 shows three different throughput levels: it takes approximately 0.7 s, 1.9 s, 4.0 s, to recover to the same throughput level under the load of 20K reqs/sec, 100K reqs/sec, 200K reqs/sec, respectively. As a reference to compare, Figure 3.20 in [5] evaluates the recovery time for an industrial Raft implementation [6], which takes about 6 seconds to recover to 18K reqs/sec.

A.5 Reconfiguration

Just like Speculative Paxos and NOPaxos, Nezha can also use the standard reconfiguration protocol from Viewstamped Replication [1] (with its incorrectness fixed by *crash-vector* [2, 3]) to change the membership of the replica group, such as replacing the failed replicas with the new ones that have a new disk, increasing/decreasing the number of replicas in the system, etc. However, Nezha is free from reconfiguring the network, whereas Speculative Paxos and NOPaxos require to modify their network (e.g., updating the forwarding rules of the Openflow controller, initializaing a new session number at the sequencer, etc.) for every reconfiguration, which adds non-trivial complexity in a real deployment.

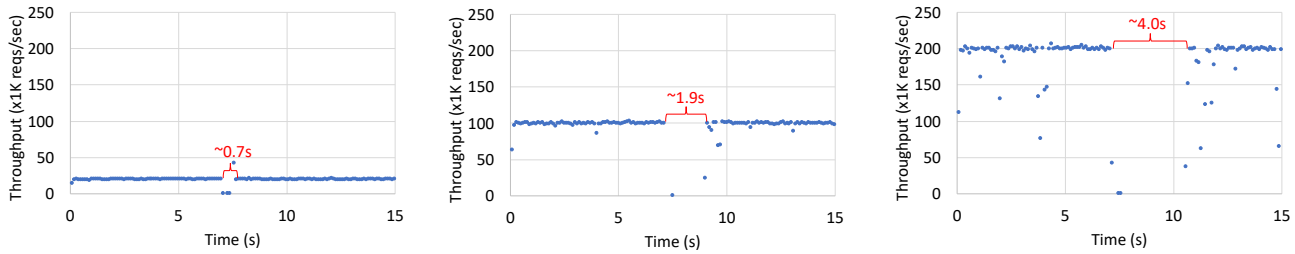
References

[1] Barbara Liskov and James Cowling. Viewstamped replication revisited. 2012.

[2] Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres. Recovering Shared Objects Without Stable Storage. In *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91, pages 36:1–36:16, 2017.

[3] Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres. Recovering Shared Objects Without Stable Storage [Extended Version]. Technical report, University of Washington, 2017.

[4] Jan Kończak, Paweł T. Wojciechowski, Nuno Santos, Tomasz Żurkowski, and André Schiper. Recovery algorithms for paxos-based state machine replication. *IEEE*



(a) Total submission rate of 20K reqs/sec (b) Total submission rate of 100K reqs/sec (c) Total submission rate of 200K reqs/sec

Figure 2: Recover to the same throughput level under different load

Transactions on Dependable and Secure Computing, 18(2):623–640, 2021.

[5] Feiran Wang. *Building High-performance Distributed Systems with Synchronized Clocks*. PhD thesis, Stanford University, 2019.

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

[7] Sarah Tollman, Seo Jin Park, and John Ousterhout. Epaxos revisited. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 613–632. USENIX Association, April 2021.

[8] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 329–341, Lombard, IL, April 2013. USENIX Association.

[9] Jeffrey C. Mogul and Ramana Rao Kompella. Inferring the network latency requirements of cloud tenants. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[10] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable message latency in the cloud. *SIGCOMM Comput. Commun. Rev.*, 45(4):435–448, August 2015.

[11] Infrastructure security in Amazon EC2. https://docs.aws.amazon.com/AWSEC2/latest/WindowsGuide/infrastructure-security.html.

[12] Network Bandwidth. https://cloud.google.com/compute/docs/network-bandwidth.

[13] Network Isolation Options for Machines in Windows Azure Virtual Networks. https://azure.microsoft.com/en-us/blog/network-isolation-options-for-machines-in-windows-azure-virtual-networks/.

[14] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 43–57, USA, 2015. USENIX Association.

[15] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. Derecho: Fast state machine replication for cloud services. *ACM Trans. Comput. Syst.*, 36(2), apr 2019.

[16] Jha Sagar, Rosa Lorenzo, and Ken Birman. Spindle: Techniques for optimizing atomic multicast on rdma. *arXiv:2110.00886v1*, 2021.

[17] Derecho Discussion Issue 237. https://github.com/Derecho-Project/derecho/discussions/237.

[18] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[19] ramfs. https://wiki.debian.org/ramfs.

[20] libfabric Programmer Manual. https://ofiwg.github.io/libfabric/v1.11.1/man/fi_tcp.7.html.

[21] Xinan Yan, Linguan Yang, and Bernard Wong. 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*, CoNEXT '20, page 351–363, New York, NY, USA, 2020. Association for Computing Machinery.