Finding Nemo-Nemo: CFT DAG-based Consensus in the WAN

Rithwik Kerur*
UCSB
rkerur@ucsb.edu

Pasindu Tennage*
Digital Asset and EPFL
pasindu.tennage@gmail.com

Philipp Jovanovic
Mysten Labs and UCL
p.jovanovic@ucl.ac.uk

Dahlia Malkhi
UCSB
dahliamalkhi@ucsb.edu

Alberto Sonnino
Mysten Labs and UCL
alberto@mystenlabs.com

Igor Zablotchi
Mysten Labs
igor@mystenlabs.com

Abstract

This paper introduces Nemo-Nemo, a performant crash-fault tolerant (CFT) consensus protocol for wide-area networks that combines design principles from both the CFT and Byzantine-fault tolerant (BFT) worlds. By structuring command propagation through a causally ordered DAG, Nemo-Nemo allows all consensus replicas to propose commands with a naturally self-regulating communication regime. By exploiting multi-leader architecture, Nemo-Nemo avoids the performance bottleneck inherent to single-leader protocols. By separating command dissemination from consensus logic, Nemo-Nemo handles challenging network conditions even when consensus commits are stalled. Moreover, leader proposals that miss a deadline are never dropped, but deterministically deferred and executed later, preserving throughput under transient network delays. And by enabling Nemo-Nemo to commit on a DAG in just two network hops, it matches the latency of existing CFT systems, while achieving significantly higher throughput. The result is the first DAG-based CFT consensus protocol that exceeds the state-of-the-art in wide-area networks both in terms of performance and robustness.

1 Introduction

Byzantine fault-tolerant (BFT) consensus systems have seen significant advances, driven by a multi-trillion dollar blockchain industry. These advances have yielded powerful techniques, notably DAG-based consensus architectures, that enable high throughput, low latency, and graceful handling of both failures and network instabilities (e.g., [5, 14, 25, 46] and others). Meanwhile, crash fault-tolerant (CFT) consensus remains the workhorse for traditional distributed systems, yet has not benefited equally from these innovations.

This paper bridges that gap. We argue that CFT systems designed to operate in the WAN can and should adopt lessons from modern BFT consensus, while adapting them to exploit the simpler model of crash-only failures. Specifically:

1. **DAG-based architecture:** We can boost performance and alleviate network and failure-induced hiccups by adopting the DAG-based approach, which provides a self-regulating mempool and seamless failure recovery.
2. **Battle-tested implementations:** Rather than building from scratch, we can adapt production-grade BFT codebases to CFT settings.
3. **Unstable network testing:** Even without Byzantine replicas, we must test CFT systems under unstable network conditions, such as shifting connected majorities and leaders, which partial synchrony alone does not capture.

This paper presents Nemo-Nemo, the first DAG-based CFT protocol. Nemo-Nemo applies these principles to achieve the best performance among known CFT protocols in wide-area network settings under both partial synchrony [17] and random asynchronous network [15] models.

Vision and motivation. Modern DAG-based BFT consensus systems offer two central lessons that form the foundation of Nemo-Nemo. The first is the value of a *causally ordered, and self-regulating mempool* A mempool functions as a transport layer for disseminating transaction requests, allowing replicas to inject transactions in parallel. In a DAG-based mempool, transactions are bundled into blocks that maintain causal order. This design achieves three goals: (1) it induces a *self-regulating, round-based* communication pattern where replicas wait for a threshold of blocks before advancing rounds; (2) each block automatically carries its causal history, ensuring consistent DAG views across correct replicas; and (3) progress continues seamlessly despite leader failures, enabling rapid recovery when new leaders are promoted.

In principle, any consensus protocol can sit atop such a DAG mempool, and many BFT protocols leverage this idea: Narwhal-HS [14] integrates HotStuff [62] with a DAG and achieves roughly a 50× performance gain over bare HotStuff, while Autobahn [20] integrates PBFT [12] with a DAG and similarly attains significant throughput improvements.

Beyond a DAG-based mempool alone, the second lesson is that consensus protocols can be embedded directly into

*Equal contribution.

the DAG structure by pre-designating a *skeleton* of leader proposals and treating blocks that reference those proposals as protocol votes [6]. In this view, much of the logic of traditional consensus protocols collapses into simple structural rules on the DAG. For example, simple and direct commit logic can be realized by $f+1$ blocks referencing a leader proposal [31, 47].

Overview of Nemo-Nemo. For the DAG-mempool, a key design choice concerns whether DAG blocks must be explicitly certified for availability by a quorum before insertion [1]. Several BFT DAGs, such as Tusk [14] and Bullshark [46], use certified blocks, motivated by the need for non-equivocation and data availability in adversarial environments. However, pre-certifying each block adds communication rounds on top of those required for consensus, increasing latency. Alternative BFT DAGs, including Cordial Miners [25] and Mysticeti [5], avoid pre-certification to reduce latency.

Prioritizing low latency, Nemo-Nemo uses non-certified blocks. Note that in CFT settings, equivocation is not a concern, hence the benefit of pre-certification is diminished whereas admitting non-certified blocks enables extremely low-latency.

The DAG *direct* commit rule of Nemo-Nemo is straightforward: a skeleton block becomes *directly* committed if referenced by $f+1$ blocks in the next round (see Figure 1). The crux is maintaining safety against failures across replicas, each distinctly evolving the DAG due to network scheduling and omissions. When a skeleton slot is not directly committed, it can become *indirectly* committed by a future skeleton slot using the (causally ordered) DAG structure itself (see Figure 2). Notably, there is no explicit view-change by which a future leader may reinstate the proposed block. Rather, a DAG rule, which is somewhat subtle, is employed to indirectly commit or skip the undecided slot; it is detailed in the body of the paper.

This design enhances state-of-the-art CFT consensus in several non-obvious respects. First, each round can inject new proposals in the same round as voting, without waiting for confirmation that earlier proposals have committed. Second, each round can rotate leaders while preserving the same structure and latency as a stable-leader regime, without requiring an explicit leader-replacement (*view-change*) protocol. Third, each round may include multiple skeleton slots for proposals, enabling greater flexibility and throughput (see Figure 2).

An additional advantage becomes apparent under unstable conditions, such as transient network delays. Consensus protocols face a well-known tradeoff when configuring leader timeouts: timeouts that are too short risk discarding correct leader proposals, while timeouts that are too long can stall the system upon leader failure or hang-up. Nemo-Nemo mitigates this tradeoff by enabling shorter leader timeouts without compromising fairness or throughput. Leader proposals are never dropped; instead, they are deferred, typically to the immediately succeeding round, allowing the system to progress smoothly despite temporary leader disruptions.

Implementing Nemo-Nemo. To validate our vision, we wanted to employ mature code and optimize it for low latency. We started with Mysticeti, a BFT protocol deployed in production by multiple blockchains [13, 22, 48].

Applying Mysticeti to CFT involved both obvious modifications and structural opportunities that fundamentally improve performance, including the following: (i) changing quorum thresholds from $2f+1$ to $f+1$, (ii) eliminating an entire communication round and simplifying commit rules to enable Nemo-Nemo to commit on a DAG in just two network hops, like existing CFT systems, (iii) simplifying authentication by removing signatures and integrity checks, thus streamlining command ingestion without Byzantine peer concerns.

Unstable Network Testing. Finally, in addition to evaluating performance under standard, partial synchrony conditions, we address an existing gap in the evaluation methodology for CFT consensus systems. Existing CFT systems are designed to cope with a minority of stragglers, progressing rather well with a stable, well-connected majority. However, they suffer a performance degradation when network conditions are unstable and the fast majority is constantly "shifting". To address this, in addition to traditional evaluations, we developed a framework to evaluate protocols under a recently proposed *random asynchronous model*, which assumes message delivery delays are random but not adversarially controlled. This model captures realistic network variability—where packets may be delayed or reordered—without assuming malicious nodes. Importantly, this model reveals performance characteristics invisible under standard partial synchrony assumptions, where protocols are typically evaluated with stable, low-latency networks. We detail how we simulate this network model in Section 6

Our evaluations demonstrate that existing CFT protocols behave quite differently under randomized network delays compared to stable networks. Nemo-Nemo, by contrast, maintains robust performance across both settings.This validates our motivating insight: by decoupling block propagation from consensus, Nemo-Nemo remains robust under the random asynchronous model as the DAG continues to grow even in the absence of commits. It contrasts with traditional CFT protocols, which couple these processes, forcing block production to halt until consensus is achieved.

Contributions. We summarize our contributions as follows:

- We present Nemo-Nemo, the first DAG-based CFT consensus protocol. Nemo-Nemo achieves the best performance among known CFT protocols in WAN settings under both partial synchrony and random asynchronous network models.

- We show how, unlike traditional CFT protocols that require explicit view-change mechanisms, Nemo-Nemo integrates leader rotation and multi-proposer rounds directly into the DAG structure, eliminating protocol complexity and latency overhead.

- We exploit CFT's lack of equivocation to admit non-certified blocks into the DAG, achieving commit latencies unattainable in BFT settings. This design choice, along with optimized commit rules and block ingestion, enables Nemo-Nemo to commit in just two network hops, like existing CFT systems, while achieving a significantly higher saturation throughput of at least 2x.
- In addition to the standard partially synchronous model, we also evaluate Nemo-Nemo under the random asynchronous model, which models unstable network conditions. Our results show that existing CFT protocols degrade significantly under this model, while Nemo-Nemo maintains robust performance.

2 System Model and Assumptions

Nemo-Nemo is a peer-to-peer message-passing system which implements atomic broadcast [10], with the following properties: (1) *Validity*: If a correct replica p proposes a value v, then p eventually commits v; (2) *No duplication*: No value is committed more than once; (3) *No creation*: If a replica commits a value v with proposer s, then v was previously proposed by replica s; (4) *Agreement*: If a value v is committed by some correct replica, then v is eventually committed by every correct replica; (5) *Total order*: Let v_1 and v_2 be any two values and suppose p and q are any two correct replicas that commit v_1 and v_2. If p commits v_1 before v_2, then q commits v_1 before v_2. In the random asynchronous model, validity and agreement are satisfied with probability 1 (almost surely).

Fault model. Nemo-Nemo follows the standard CFT assumptions used in prior work [27, 40]. The system consists of $n = 2f + 1$ replicas, of which at most f may crash. A replica is *correct* if it follows the protocol and *crashed* otherwise.

Network model. We analyze and evaluate Nemo-Nemo under two network models: the standard partial synchrony model and the more recent random asynchronous model. Links between correct parties behave reliably, and messages among them eventually arrive.

In the partially synchronous model, we assume the standard Global Stabilization Time (GST) used throughout related work. After GST, every message sent among correct parties arrives within Δ time. Before GST, the protocol operates in a fully asynchronous network where messages eventually arrive but no bound limits their delay.

Although Nemo-Nemo operates in the CFT setting, it is designed to operate in a more unstable environment prone to network instabilities. To model this, we employ the random asynchronous model [15] which assumes that message delivery follows a random schedule. The random asynchronous model strikes a midpoint between the partially synchronous and fully asynchronous settings: it does not rely on synchronous periods to commit, but relaxes the worst-case adversarial scheduling of the classic asynchronous model [7, 34], where an adversary can control the message schedule indefinitely.

3 The Nemo-Nemo DAG: Data Dissemination

Nemo-Nemo operates in two logical layers: (i) a data dissemination layer that propagates client transactions through a structured DAG (this section); and (ii) an ordering layer interpreting the DAG to derive a total order of transactions (see Section 4).

3.1 Block creation

Replicas operate in a sequence of logical *rounds*, and in each round, every replica proposes a unique *block*. During a round, replicas receive transactions from clients and blocks from other replicas. Clients submit transactions to a replica, which includes them in its (next) block. If a transaction does not finalize quickly enough, the client simply resubmits it to a different replica. Each block references blocks from prior rounds, starting with the author's most recent block, and includes *fresh transactions* not yet included in preceding blocks. Specifically, a block contains at least the following elements: (1) the author A of the block; (2) a round number R; (3) a list of transactions; and (4) at least $f + 1$ distinct references to blocks from the previous round $R - 1$, as well as any additional references from earlier rounds, for which the replica has already downloaded the complete causal history. At the core of Nemo-Nemo is a round-based DAG structure where each vertex is a block. Each replica maintains a local view of the DAG, adding a block once it has downloaded its entire causal history. Consequently, all correct replicas eventually converge on the same DAG view.

3.2 Inclusive block proposing

Unlike traditional leader-based consensus protocols, where only leaders propose values, in Nemo-Nemo all replicas, leaders or not, propose values in each round. In each round, a subset of replicas known to everyone is pre-designated as *skeleton* nodes. Skeleton nodes are akin to leaders (possibly multiple ones per view, as in multi-leader protocols). That is, blocks by skeleton nodes drive progress in the protocol: they can become directly committed, and they determine the commit decisions for blocks they causally reference, as described in the ordering layer (see Section 4). A replica disseminates its block only after receiving (in addition to $f + 1$ blocks from the previous round, also) all skeleton blocks from the previous round, or after a timeout expires. This ensures liveness once the system becomes synchronous (see Supplementary Material).

Random Asynchronous Model. This model assumes that the first $f + 1$ blocks received by a replica in a round R constitute a random sample. We discuss the implementation of this mechanism in Section 6 and prove that Nemo-Nemo is live with probability 1 under this model (see Section 5.2).

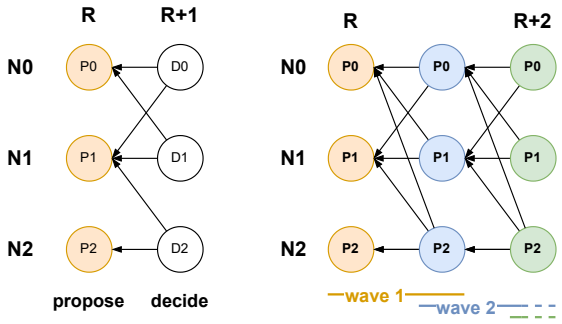

Figure 1: The structure of the Nemo-Nemo DAG. Left: The structure of a wave, consisting of two rounds (Propose and Decide). Right: Waves patterns in the Nemo-Nemo protocol (each round starts a new overlapping wave).

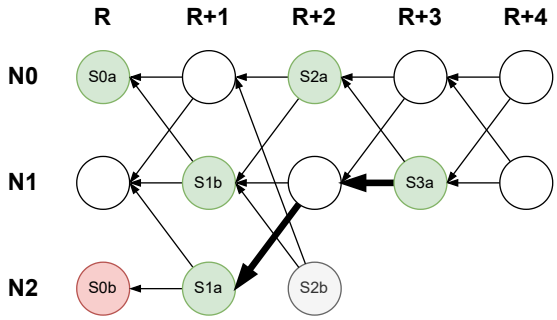

Figure 2: Example execution with three replicas and two skeleton slots per round. Skeleton blocks are classified as `commit` (green), `skip` (red), or `undecided` (grey). Thick arrows signify commits via the indirect decision rule.

4 The Nemo-Nemo Consensus Protocol

By itself, the DAG layer described in Section 3 functions as a scalable data dissemination layer. It grows with the network and ensures that all disseminated transactions are reliably available. However, a consensus layer is required to order transactions. Nemo-Nemo integrates this layer logically into the DAG messages, rather than sending separate consensus messages, and infers decisions directly from the DAG itself.

4.1 Identifying DAG patterns

Nemo-Nemo leverages patterns in the DAG structure to define its decision rules.

Rounds and waves. Figure 1 (left) shows an example of a Nemo-Nemo DAG with three replicas, (N_0, N_1, N_2). Nemo-Nemo defines a *wave* of two rounds for every block. The first round R (Propose) includes the blocks that the wave attempts to commit (P_0, P_1, P_2). The second round $R+1$ (Decide) includes the blocks (D_0, D_1, D_2) implicitly acting as "votes" [27] determining the blocks of the previous round to commit. Figure 1 (right) shows that Nemo-Nemo initiates a new wave every round, with two consecutive waves always overlapping by one round. Put differently, each round starts a new wave. Algorithm 2 (Section 4.2) formally defines a wave.

Skeleton slots. A skeleton slot (or simply "a slot") is a tuple (replica, round) and can be either empty or contain the replica's proposal for the respective round. The slot can assume one of three states: `commit`, `skip`, or `undecided`. All slots are initially set to `undecided` and the goal of the protocol is to classify them as `commit` or `skip`.

The number of skeleton slots instantiated per round can be configured, and for systems with few faults, it can be as high as n so that every block has a chance to commit in two steps. It can also be dynamically adjusted based on the network conditions, following a similar deterministic approach to HammerHead [55]. The protocol is initialized with a deterministic total order among skeleton slots, known to all replicas. Within a single round, this ordering may reflect a fixed (*e.g.,* round robin) or a variable regime per round. Figure 2 illustrates an example of a Nemo-Nemo DAG with three replicas, (N_0, N_1, N_2), two skeleton slots per round, and a potential skeleton slot ordering represented as (S_{0a}, S_{0b}) and (S_{1a}, S_{1b}) for the first and second rounds, respectively.

4.2 The Nemo-Nemo decision rule

We present the decision rule of Nemo-Nemo leveraging an example protocol run depicted in Figure 2. In this example, we refer to blocks using the notation $B_{(N_i,R)}$, where N_i is the issuing replica and R is the block's round.

All skeleton slots are initially in the `undecided` state. The replica holds the portion of the DAG depicted in Figure 2 and attempts to classify as many blocks in the skeleton slots as possible as either `commit` or `skip`.

Step 1: Direct decision rule. To start the classification process, the replica processes each slot individually, starting with the highest (S_{3a}), applying the Nemo-Nemo *direct decision rule*. The replica classifies a block B in a slot as `commit` if it observes $f+1$ blocks (*i.e.,* two blocks in our example) from the subsequent round referencing it. Otherwise, the replica leaves the slot as `undecided` (for now). This rule is formally described by the function TryDirectDecide in Algorithm 2.

In Figure 2, the replica targets S_{3a} first. It observes that $B_{(N_0,R+4)}$ and $B_{(N_1,R+4)}$ reference it. Therefore, it classifies S_{3a} as `commit`. Section 7 shows that this scenario is the most common (in the absence of an asynchronous adversary) and results in the lowest latency. Blocks S_{2a}, S_{1b}, and S_{0a} are also classified as `commit` as they each have $f+1$ references from the subsequent round.

Step 2: Indirect decision rule. In the case where the direct decision rule cannot classify a skeleton slot $B_{(N_i,R)}$, the replica uses the Nemo-Nemo *indirect decision rule*. This rule looks at future slots to decide about the current one. First, it finds an *anchor*. This is the earliest non-skipped skeleton slot with a round number $R' > R+1$ that is either still classified as `undecided` or already classified as `commit`. Recall that slots are totally-ordered, so the anchor is uniquely defined regardless of how the DAG evolves at different replicas. If the anchor is

`undecided`, the replica marks the current slot as `undecided`. If the anchor is `commit`, the replica checks if it indirectly references the target slot, that is, it checks whether there is a path between the anchor and the target skeleton slot. If it does, the replica marks the target slot as `commit`. If it does not, the replica marks it as `skip`. This rule is formally described by the function TryIndirectDecide in Algorithm 2. Section 5 shows that the direct and indirect decision rules are consistent: if one replica directly commits a block, no honest replicas will indirectly skip it (and vice versa).

In this example, the replica fails to classify S_{2b}, S_{1a}, and S_{0b} using the direct decision rule and thus searches for their respective anchors. The status of the anchor of S_{2b} is still `undecided`; the replica thus marks S_{2b} as `undecided`. Eventually, the DAG will grow and a block with round $R' > R+3$ will become the anchor for S_{2b}. For the moment however, S_{2b} remains `undecided`. S_{3a} is the anchor for S_{1a}; since S_{3a} has a path to S_{1a} (marked with thick arrows), the replica classifies S_{1a} as `commit`. The anchor of S_{0b} is S_{2a} which does not have a path to S_{0b}. Thus, the replica classifies S_{0b} as `skip`.

Step 3: Skeleton slots sequence. After processing all slots, the replica derives an ordered sequence of the blocks contained in the skeleton slots. It then iterates over this sequence, committing all slots marked as `commit` and skipping all slots marked as `skip`. This process continues until the replica encounters the first `undecided` slot. As shown in Section 5 this commit sequence is safe, and eventually, all slots will be classified as either `commit` or `skip`. In the example shown in Figure 2, the skeleton block output by the replica is [S_{0a}, S_{1a}, S_{1b}, S_{2a}]. Since S_{2b} remains `undecided`, it is not included in the sequence and S_{3a} is also excluded.

Step 4: Commit sequence. Following the approach introduced by DagRider [24], the replica determines a linear ordering of all the blocks within the sub-DAG defined by each skeleton block, including previously skipped skeleton blocks (if they are reachable), by performing a depth-first search. If a previous skeleton slot has already linearized a block, it is not re-linearized. The replica processes skeleton slots sequentially, ensuring that all blocks are included in the final commit sequence in the correct order, according to their causal dependencies. The procedure LinearizeSubDags of Algorithm 3 formally describes this linearization process.

In this example, S_{0a} and S_{0b} do not define any sub-DAG (because the example begins at round R) and are thus directly added to the commit sequence. Next, S_{1a} defines the sub-DAG $\{L_{1a}, B_{(N_1,R)}, L_{0b}\}$, which is linearized as [$B_{(N_1,R)}$, S_{0b}, S_{1a}]. The replica continues this process for each skeleton, linearizing the sub-DAGs defined by S_{1b} as [S_{1b}], since both S_{0a} and $B_{(N_1,R)}$ are already part of the commit sequence, and so forth. The final commit sequence is [S_{0a}, $B_{(N_1,R)}$, S_{1a}, S_{1b}, $B_{(N_0,R+1)}$, S_{2a}].

Algorithm 1 Nemo-Nemo

```
1: leadersPerRound                                   ▷ A number between 1 and 2f+1
2: waveLength = 2

3: procedure TryDecide(r_committed, r_highest)
4:     S ← [ ]                                       ▷ Holds decisions
5:     for r ← r_highest down to r_committed + 1 do
6:         for l ← leadersPerRound − 1 down to 0 do
7:             i ← r mod waveLength
8:             D ← Decider(i, l)
9:             w ← D.WaveNumber(r)
10:            s ← D.TryDirectDecide(w)
11:            if s = ⊥ then s ← D.TryIndirectDecide(w, S)
12:            S ← s ‖ S
13:    return S

14: procedure ExtendCommitSequence(r_committed, r_highest)
15:    S ← TryDecide(r_committed, r_highest)
16:    S_commit ← [ ]                                ▷ Holds committed blocks
17:    for s ∈ S do
18:        if s = ⊥ then break
19:        if s = Commit(b_leader) then S_commit ← S_commit ‖ b_leader
20:    return LinearizeSubDags(S_commit)             ▷ Commit returned blocks
```

5 Correctness

In this section, we prove that Nemo-Nemo guarantees the properties of atomic broadcast.

5.1 Safety proofs

Lemma 1. *If in round R, $f+1$ blocks from distinct replicas vote for a block B, then all blocks at future rounds $R' > R$ will have a path to B.*

Proof. We prove the lemma by induction on R'. The base case is $R' = R+1$. Let B' be a block at round R'. Since B' points to $f+1$ blocks at round R, by quorum intersection, B' must point to at least one of the blocks that vote for B, and thus have a path to B.

For the induction case, assume the lemma holds up to round R' and consider the case of round $R'+1$. Let B' be a block at round $R'+1$. By the induction hypothesis, $f+1$ blocks at round R' have paths to B. Since B' points to $f+1$ blocks from round R', by quorum intersection, B' must point to at least one block that has a path to B. □

Lemma 2. *If a replica directly commits some block in a slot S, then no other replica skips slot S.*

Proof. Assume by contradiction that a replica N directly commits block B in slot S while another replica N' decides to skip S. Let R be the round of S. Since N directly commits B, there exist $f+1$ votes for B at S. Therefore, by Lemma 1, all blocks at rounds $R' > R$, including the anchor of S, have a path to B at S. Thus, N' cannot decide to skip S using the indirect decision rule. We have reached a contradiction. □

Observation 1. *If a slot S is committed at two replicas, then S contains the same block at both replicas.*

Proof. This follows, by construction, from two facts: (1) the sequence of skeleton slots is predetermined and known to all replicas, (2) replicas propose at most one block per slot. □

Algorithm 2 Decider Instance

```
1: waveOffset = i                          ▷ The first parameter of the Decider (i)
2: leaderOffset = l                        ▷ The second parameter of the Decider (l)
3: waveLength = 2

4: procedure WaveNumber(r)
5:     return (r − waveOffset)/waveLength

6: procedure ProposeRound(w)
7:     return (w · waveLength) + waveOffset

8: procedure DecisionRound(w)
9:     return ProposeRound(w)+(waveLength − 1)

10: procedure SupportedLeader(w, b_leader)
11:     B_decision ← GetDecisionBlocks(w)
12:     return |{b' ∈ B_decision : IsLink(b', b_leader)}| ≥ f + 1

13: procedure TryDirectDecide(w)
14:     b_leader ← GetLeaderBlock(w, leaderOffset)
15:     if SupportedLeader(w, b_leader) then return Commit(b_leader)
16:     else return ⊥

17: procedure TryIndirectDecide(w, S)
18:     r_decision ←DecisionRound(w)
19:     s_anchor ← first s ∈ S s.t. r_decision < s.round ∧ s ≠ Skip
20:     if s_anchor = Commit(b_anchor) then
21:         b_leader ← GetLeaderBlock(w, leaderOffset)
22:         if Link(b_anchor, b_leader) then return Commit(b_leader)
23:         else return Skip
24:     return ⊥
```

We say that a slot S is *decided* at a replica N if S is committed or skipped. Otherwise, S is *undecided*.

Lemma 3. *If a slot S is decided at two replicas N and N', then either both replicas commit S, or both replicas skip S.*

Proof. Assume by contradiction that there exists a slot S such that N and N' decide differently at S. We consider a finite execution prefix and assume *wlog* that S is the highest slot at which N and N' decide differently ($\star$). Further assume *wlog* that N commits S and N' skips S. By Lemma 2, neither N nor N' could have used the direct decision rule for S; they must both have used the indirect rule. Consider now the anchor of S: N and N' must agree on which slot is the anchor of S, since by our assumption ($\star$) above, they make the same decisions for all slots higher than S, including the anchor of S. Let S' be the anchor of S; S' must be committed at both N and N'. Thus, by Observation 1, N and N' commit the same block B' at S'. But then N and N' cannot reach different decisions about slot S using the indirect decision rule, a contradiction. □

We have proven the consistency of replicas' commit sequences: replicas commit (or skip) the same skeleton blocks, in the same order. However, we are not done: we also need to prove that non-skeleton blocks are committed in the same order by replicas. We show this next.

Causal history and commit conditions. Consider a replica N. We call the *causal history* of a block B in N's DAG, the transitive closure of all blocks referenced by B in N's DAG, including B itself. In Nemo-Nemo, a block B is committed by a replica N if (1) there exists a committed skeleton block L in N's DAG such that B is in L's causal history (2) all skeleton slots up to L are decided in N's DAG and (3) B has not been committed as part of a lower slot's causal history. In this case, we say B is *committed at* slot S, or *committed with* block L.

Algorithm 3 Helper Functions

```
1: nodes                                   ▷ The set of nodes

2: procedure GetDecisionBlocks(w)
3:     r_decision ←DecisionRound(w)
4:     return DAG[r_decision]

5: procedure GetLeaderBlock(w, rank)
6:     r_propose ←ProposeRound(w)
7:     leader ← nodes[(r_propose + rank) mod |nodes|]
8:     if ∃b ∈ DAG[r_propose] s.t. b.author = leader then return b
9:     else return ⊥

10: procedure IsLink(b_new, b_old)
11:     return ∃ a sequence of k ∈ ℕ blocks b_1, ..., b_k s.t.
          b_1 = b_old ∧ b_k = b_new ∧ ∀j ∈ [2, k] : b_j ∈ ⋃_{r≥1} DAG[r] ∧ b_{j−1} ∈
      b_j.parents

12: procedure LinearizeSubDags(L)
13:     O ← [ ]                              ▷ Hold output sequence
14:     for b_leader ∈ L do
15:         B ← {b ∈ ⋃_{r≥1} DAG[r, *] s.t. IsLink(b, b_leader) ∧ b ∉ O ∧
          b not already output}
16:         for b ∈ B in any deterministic order do
17:             O ← O || b
18:     return O
```

Lemma 4. *If a block B is committed by two replicas N and N', then B is committed at the same slot S, and B is committed with the same skeleton block L, at both N and N'.*

Proof. Let S be the slot at which B is committed at replica N, and L the corresponding skeleton block in S, also at replica N. Consider now the slot S' at which B is committed at replica N', and L' the corresponding skeleton block. Assume by contradiction that $S' \neq S$. If $S' < S$, then N would have also committed B at slot S', since by Observation 1, they must commit the same skeleton blocks in the same slots, so N could not have committed B again at slot S; a contradiction. Similarly, if $S < S'$, then N' would have already committed B at slot S, since by Observation 1 N and N' must have committed the same block in slot S; contradiction. Thus, it must be that $S = S'$, and by Observation 1, $L = L'$. □

We can now prove the main safety properties of Nemo-Nemo: total order, no duplication, and no creation.

Theorem 1 (Total Order)**.** *Nemo-Nemo satisfies the total order property of Atomic Broadcast.*

Proof. This property follows immediately from Lemma 4 and the fact that replicas order the causal histories of committed blocks using the same deterministic function, and commit blocks in this order. □

Theorem 2 (No duplication)**.** *Nemo-Nemo satisfies the no duplication property of Atomic Broadcast.*

Proof. This is by construction: a block B is committed as part of the causal history of a committed skeleton block only if B has not been committed along with an earlier leader block (see "Causal history & commit conditions" above). So a replica cannot commit the same block twice. □

Theorem 3 (No creation)**.** *Nemo-Nemo satisfies the no creation property of Atomic Broadcast.*

Proof. This follows from two facts: (1) replicas only include in their local DAGs, blocks that have been previously proposed by some replica, and (2) replicas only commit blocks that they have previously included in their DAGs. □

5.2 Liveness proofs

We now turn to liveness. We prove liveness in the random asynchronous model here, and defer the proof of liveness under partial synchrony to the Supplementary Material.

Lemma 5. *If a block B produced by a correct replica N references some block B', then B' will eventually be included in the local DAG of every correct replica.*

Proof. If some replica N' receives B from N, but does not have B' yet, N' will request B' from N; since N is correct and the network links are reliable, N will eventually receive N''s request, send B' to N', and N' will eventually receive B'. The same is recursively true for any blocks from the causal history of B', so N' will eventually receive all blocks from the causal history of B' and thus include B' in its local DAG. □

Lemma 6. *If a correct replica N proposes a block B, then every correct replica will eventually include B in its local DAG.*

Proof. Since network links are reliable, all correct replicas will eventually receive B from N. By Lemma 5, all correct replicas will eventually receive all of B's causal history, and so will include B in their local DAG. □

Lemma 7. *Fix a skeleton slot S. A replica directly commits S with probability at least $1/2$.*

Proof. A block in round $R+1$ will reference a random subset of $f+1$ blocks in round R. The probability that a given block B votes for the skeleton slot S is $p = \frac{f+1}{2f+1} > \frac{1}{2}$. Let $V(S)$ denote the random variable representing the number of replicas in round $R+1$ that vote for S in round R. $V(S)$ follows a Binomial distribution: $V(S) \sim B(n,p)$, where $p = \frac{f+1}{2f+1}$ The expected value of $V(S)$ is: $E[V(S)] = n \cdot p = f+1$.

Since the mean $\mu = f+1$ is an integer, the mean and the median of the distribution are equal [30]. By the definition of the median m, we have $\Pr(V(S) \geq m) \geq \frac{1}{2}$. Thus, the probability of a skeleton slot in round R receiving at least $f+1$ votes in round $R+1$, and thus being directly committed, satisfies: $\Pr(V(S) \geq f+1) \geq 1/2$. □

Lemma 8. *Fix a skeleton slot S. Every correct replica eventually either commits or skips S with probability* 1.

Proof. By Lemma 7, the probability of a correct replica directly committing any skeleton block in a given round is a constant ($\geq 1/2$). Thus, the probability that a sequence of t rounds without any directly committed slot is less than 2^{-t}. This implies that w.h.p., every slot that is not directly decided will eventually have a committed anchor and therefore become decided (*i.e.,* it will be committed or skipped). □

Theorem 4 (Validity)**.** *Nemo-Nemo satisfies the validity property of Atomic Broadcast, with probability* 1.

Proof. Let N be a correct replica and B a block proposed by N. We show that, with probability 1, B is eventually committed by every correct replica. By Lemma 6, B is eventually included in the local DAG of every correct replica. So every correct replica will eventually include a reference to B in at least one of its blocks. Let R be the highest round at which some correct replica includes a reference to B in one of its blocks. By Lemma 7, with probability 1, each skeleton block has a nonzero probability of being directly committed, so eventually some block B' at a round $R' > R$ will be directly committed. Since all replicas have B in their causal histories by round R, B' must therefore have a path to B. Lemma 8 guarantees that all slots before B' are eventually decided, so B' is eventually committed. Thus, B will be committed at all correct replicas at the latest when B' is committed alongside its causal history. □

Theorem 5 (Agreement)**.** *Nemo-Nemo satisfies the agreement property of Atomic Broadcast, with probability* 1.

Proof. Let N be a correct replica and B a block committed by N. We show that, with probability 1, B is eventually committed by every correct replica. Let L be the skeleton block with which B is committed and S the corresponding slot. By Lemma 8, all blocks up to and including S are eventually decided by all correct replicas, with probability 1. By Observation 1, all correct replicas commit L in S. Eventually, all correct replicas commit B. □

6 Implementation

We implemented Nemo-Nemo in Rust, building on the open-source Mysticeti codebase [26], which contains approximately 14,000 lines of code (LOC). Our implementation uses `tokio` [49] for asynchronous networking and raw TCP sockets for replica-to-replica communication.

To support crash recovery and ensure data persistence, we implemented a Write-Ahead Log (WAL) tailored to Nemo-Nemo. This provides stronger resilience than existing implementations of Multi-Paxos [50] and QuePaxa [51], which lack mechanisms for recovering after crashes, though this absence arises from implementation choices rather than protocol specifications. Following prior implementations of Multi-Paxos, QuePaxa, and Rabia [41], we support batching at the replica level to amortize communication costs by packing multiple commands into a single message. We collocated front- and back-end within the same Nemo-Nemo binary. Each front-end communicates with its assigned back-end, a design choice common in existing consensus protocol implementations [37].

We also implement a benchmark framework to simulate performance under a random asynchronous network model. The framework forces each consensus instance to gather a randomly selected quorum to commit rather than rely on whichever majority returns first. This shifts performance away from the quickest or geographically closest replicas and instead drives execution through a random message-delivery schedule that removes any bias toward low-latency quorums. We use this framework only in Section 7.7 and rely on standard communication patterns for the rest of our evaluation in Section 7. To the best of our knowledge, this is the first work to introduce an evaluation framework for measuring protocol performance under the random asynchronous model, and we present this evaluation setup as a novel contribution of independent interest. We will open-source our Nemo-Nemo implementation upon acceptance to support reproducibility and enable further research in this area.

7 Evaluation

Our evaluation compares performance, scalability, and resource efficiency to existing CFT consensus protocols. We experiment across a range of execution conditions, including favorable (fault-free, timely communication), crash-fault scenarios, and both normal and unstable network conditions.

7.1 Baselines

We compare the performance of Nemo-Nemo against state-of-the-art CFT protocols: Multi-Paxos [27, 50], EPaxos [35, 36], QuePaxa [51, 52], Rabia [41, 42], and SADL-RACS [53]

We select these baselines for their contrasting designs, ensuring coverage of a broad spectrum of protocol architectures. Multi-Paxos uses a classic leader-based approach that channels all requests through a single leader. EPaxos adopts a multi-leader design in which every replica can propose and commit commands concurrently under dependency constraints. EPaxos produces only a partial order, a weaker abstraction than the total order that Nemo-Nemo enforces. QuePaxa represents the current state of the art in randomized consensus: it matches Multi-Paxos under synchronous conditions and switches to a fallback mode that preserves throughput under asynchrony, which makes it well suited for wide-area deployments. Rabia uses randomized techniques to optimize for low-latency local-area settings; we include it for completeness. SADL-RACS achieves high throughput by separating command dissemination from the consensus critical path. We include QuePaxa, Multi-Paxos, EPaxos, and Nemo-Nemo in all subsequent experiments. Due to page-limit constraints, we report only best-case evaluations for SADL-RACS and Rabia. We observed that the existing EPaxos codebase [35] does not support more than five replicas, an implementation limitation noted by prior work [52, 53]. Therefore, we omit using EPaxos in the scalability experiment in Section 7.4. Nemo-Nemo is designed for crash fault-tolerant systems; however, for completeness, we also compare Nemo-Nemo against

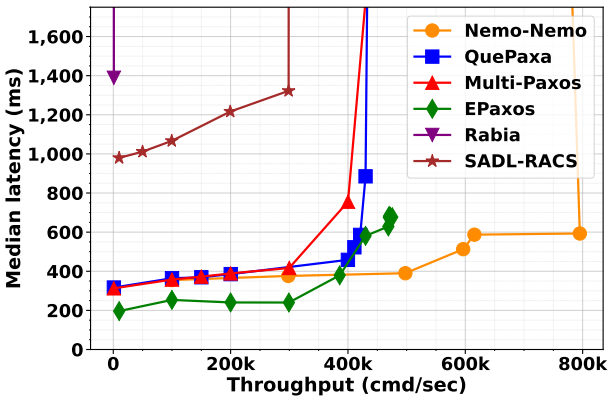

Figure 3: Performance under normal case WAN execution.

two state-of-the-art DAG-based, partially synchronous BFT consensus protocols: Bullshark [46] and Mysticeti [5]. We select these two BFT protocols because they share key architectural features with Nemo-Nemo and because both are widely deployed in production systems [8, 48].

7.2 Experimental setup

We deploy replicas on Amazon EC2 virtual machines [4] of type `c5.4xlarge`. Each machine provides 10 Gbps of bandwidth, 16 virtual CPUs on a 2.5 GHz Intel Xeon Platinum 8124M, 32 GB memory, and runs Ubuntu Linux 22.04 LTS [56]. We select these machines because they provide decent performance, are in the price range of "commodity servers", and are in line with recent related work [5, 23]. We conducted experiments in a wide-area network (WAN) with replicas distributed across multiple AWS regions: Cape Town (`af-south-1`), Hyderabad (`ap-south-2`), Jakarta (`ap-southeast-3`), Osaka (`ap-northeast-3`), and Milan (`eu-south-1`).

We instantiate several geo-distributed benchmark clients collocated with each replica, submitting transactions in an open loop model [45], at a fixed rate. We experimentally increase the submission rate and record the throughput and latency of commits. As a result, all plots illustrate the steady-state latency of all systems under load, as well as the maximal throughput they can provide, after which latency increases rapidly. We verified that the CPU, memory, bandwidth, and storage resources are sufficient to host both the replica and client for all protocols under test.

Following prior studies [42, 52] and production systems [9], we set the command size to 18 bytes. We set the timeouts at 5 sec in each protocol. For each protocol, we measure end-to-end commit latency. Throughput is measured in commands per second (cmd/sec), where one command corresponds to a single 18-byte request. All protocols use batching to amortize network latency.

7.3 Normal case WAN performance

We evaluate the normal-case performance of Nemo-Nemo in a WAN under favorable ("synchronous") network conditions.

Figure 3 shows the throughput and median latency.

Multi-Paxos. We observe in Figure 3 that Nemo-Nemo reaches a saturation throughput of 800k cmd/s at a median latency of 600 ms. In contrast, Multi-Paxos reaches only 400k cmd/s at 750 ms. This corresponds to a 2x throughput improvement and a 20% latency reduction for Nemo-Nemo over Multi-Paxos.

Nemo-Nemo's throughput advantage over Multi-Paxos stems from two key factors. First, Multi-Paxos funnels all client traffic through a single leader, creating a bottleneck, whereas Nemo-Nemo distributes load uniformly across all replicas (see Section 3). Second, Nemo-Nemo amortizes a single message per batch of commands (see Section 4), while Multi-Paxos requires ten messages per batch in a five-replica deployment, consisting of five *propose* messages followed by five *accept* messages.

EPaxos. We observe in Figure 3 that EPaxos delivers over 400k cmd/s at a median latency of 440 ms, compared to Nemo-Nemo which reaches 800k cmd/s at 600 ms. Both EPaxos and Nemo-Nemo distribute load among all participating nodes and therefore outperform leader-based Multi-Paxos in latency.

For a throughput below 300k cmd/s, EPaxos achieves a 130 ms lower median latency than Nemo-Nemo. This latency advantage arises because EPaxos provides only a partial order of commands, which incurs less coordination overhead than Nemo-Nemo, which enforces a total order. In this experiment, we configured EPaxos with a 2% conflict rate [54], meaning that 98% of commands commit in a single round trip, resulting in lower commit latency than Nemo-Nemo at moderate load.

However, beyond 400k cmd/s, Nemo-Nemo achieves roughly 2× the throughput of EPaxos. This gap stems from two factors. (1) The EPaxos implementation suffers from implementation limitations—most notably a single-threaded main execution path—whereas Nemo-Nemo employs a highly parallelized design. (2) Under high arrival rates, EPaxos's conflict-resolution overhead becomes a dominant bottleneck. By avoiding specialized conflict resolution, Nemo-Nemo sustains substantially higher throughput.

EPaxos is well suited for applications requiring only partial order with low conflict rates, such as key-value stores where per-key ordering is sufficient. By contrast, Nemo-Nemo targets applications requiring a total order of all commands, providing higher throughput for workloads that demand strong consistency.

QuePaxa. We observe in Figure 3 that QuePaxa achieves only 400k cmd/s at a median latency of 460 ms. Under normal-case executions, QuePaxa[1] funnels all commands through a single leader replica. As a result, QuePaxa's performance is limited by the leader replica's capacity, whereas Nemo-Nemo achieves higher throughput and lower latency by load balancing across all participating nodes.

[1]Ideally, both QuePaxa and Multi-Paxos should incur the same overhead and achieve comparable performance in normal-case operation. However, the QuePaxa implementation yields higher throughput than Multi-Paxos because it leverages gRPC-based high-speed messaging and multi-threading, whereas the Multi-Paxos implementation employs single-threaded execution.

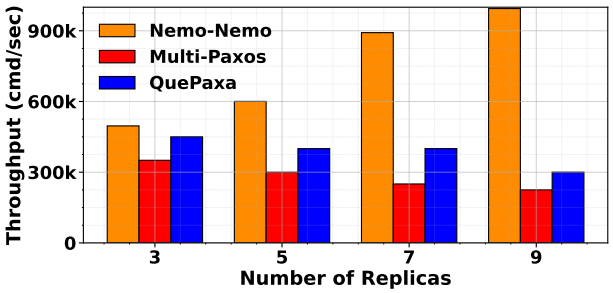

Figure 4: Scalability with replication factor. Saturation throughput under 500 ms median latency.

SADL-RACS. SADL-RACS uses a mempool to disseminate client commands off the consensus critical path. In our deployment, it achieves a saturation throughput of 300k cmd/s with a median latency of 1.35 s. In comparison, Nemo-Nemo reaches 800k cmd/s and 600 ms, corresponding to a 166% higher throughput and 125% lower latency.

The performance advantage of Nemo-Nemo arises from its avoidance of extra dissemination round trips. SADL-RACS requires two additional hops to propagate commands, which significantly increases latency. By embedding command dissemination into the DAG and balancing load across replicas, Nemo-Nemo sustains higher throughput and lower latency.

Rabia. In our WAN deployment, Rabia delivers only 1k cmd/s with latency above 1.25 s, whereas Nemo-Nemo achieves 800k cmd/s at 600 ms. Its lower WAN performance stems from its design assumptions and optimizations for low-latency, high-bandwidth LANs. As network diameter and latency variability increase, Rabia's design assumptions no longer hold, leading to severely reduced throughput and higher latency.

Tail latency. None of the evaluated systems is optimized for tail latency, and in practice we observe comparable percentile 99 behavior across all protocols. Under a 1 s percentile 99 bound, Nemo-Nemo sustains 400k cmd/s, while EPaxos reaches 375k cmd/s, and both QuePaxa and Multi-Paxos reach roughly 300k cmd/s. These results are consistent with prior findings reported in related work [52].

Summary. Across our experiments, Nemo-Nemo consistently outperforms existing CFT protocols in WAN deployments, achieving higher throughput and lower latency. Its advantages arise from uniform load distribution, minimal message delays, and avoidance of specialized assumptions (such as latency bounds) or conflict-resolution bottlenecks. These observations validate our vision: Nemo-Nemo is well suited for applications that require total ordering, high throughput, and low latency, in cloud and wide-area environments.

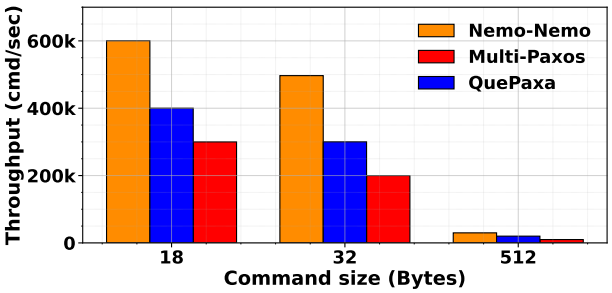

Figure 5: Scalability with command size. Saturation throughput under 500 ms median latency and 5 replicas.

7.4 Scalability with number of replicas

We evaluate how Nemo-Nemo scales as the number of replicas increases. We deploy ensembles of 3, 5, 7, and 9 replicas across geographically distributed AWS regions and compare against pipelined Multi-Paxos and QuePaxa[2]. Figure 4 shows the saturation throughput under 500 ms median latency.

Leader-based performance degradation. Multi-Paxos and QuePaxa exhibit throughput degradation of 35% and 33%, respectively, when scaling from 3 to 9 replicas. The performance degradation stems from leader-centric designs: as the replication factor grows, the leader must handle more messages due to larger quorum sizes, creating a resource bottleneck.

Multi-leader scalability. In contrast, Nemo-Nemo's throughput doubles from 500k to 1M cmd/sec over the same range, demonstrating superior scalability. Nemo-Nemo exhibits opposite behavior due to its multi-leader DAG-based architecture. With only 3 replicas, Nemo-Nemo underutilizes available network and CPU resources. Adding more replicas improves resource multiplexing across leaders, increasing throughput. Furthermore, more replicas increase the number of blocks per round and the causal history size of leader blocks without additional network hops, enabling higher parallelism.

7.5 Scalability with command size

We evaluate the impact of command size on Nemo-Nemo's performance. We experiment with three command sizes: 18 B, 32 B, and 512 B, as used in recent SMR work [3, 53]. We deploy Nemo-Nemo, pipelined Multi-Paxos, and pipelined QuePaxa in a WAN setting with 5 replicas. Figure 5 depicts the saturation throughput under 500 ms median latency.[3]

As command size increases from 18 B to 512 B, Multi-Paxos degrades from 300k to 10k cmd/s, while QuePaxa drops from 400k to 20k cmd/s. In contrast, Nemo-Nemo maintains 50–66% higher throughput at 18 B and 32 B, and still retains a 5% advantage at 512 B, degrading from 600k to 30k cmd/s.

This performance gap stems from fundamental architectural differences. In Multi-Paxos and QuePaxa, the single leader must handle all client commands and broadcast them to followers, making the leader's network bandwidth the bottleneck as command size grows. Nemo-Nemo's multi-leader DAG-based design distributes bandwidth usage evenly across all replicas, avoiding the leader bottleneck entirely.

7.6 Performance under crash failures

We evaluate how each system handles crash failures by deploying five replicas in a WAN setup under a constant load of 40k cmd/sec. At 25 seconds, we crash the leader replica in Multi-Paxos and QuePaxa, and a random replica in Nemo-Nemo and EPaxos. Figure 6 shows the throughput over time.

Leader-based protocols suffer downtime. Multi-Paxos throughput drops to zero immediately upon leader crash and remains at zero during the leader election (view change). Once a new leader is elected, the backlog of accumulated commands is processed, causing a throughput spike before returning to steady state. QuePaxa exhibits similar behavior despite employing multiple leaders via a hedging schedule: by default, only the first leader proposes in the crash-free case, and the second leader begins proposing only *after the configured hedging timeout*. This design still creates a period of zero throughput during leader transition, followed by a spike when the new leader processes the backlog.

Multi-leader protocols avoid downtime. Both EPaxos and Nemo-Nemo exhibit no visible downtime under replica crashes. As multi-leader protocols, the failure of one replica does not affect the ability of the remaining replicas to make progress. Nemo-Nemo avoids explicit view changes and promptly discards blocks from crashed replicas, preventing head-of-line blocking in the commit sequence. This ensures that even if a crashed replica was previously a leader, its uncommitted blocks do not stall the system's progress.

7.7 Performance under random asynchrony

We evaluate Nemo-Nemo's performance when experiencing random message delays, leveraging the benchmarking framework presented in Section 6. To amplify the effects of network randomization, we deploy 5 replicas in Cape Town (af-south-1), London (eu-west-2), Ireland (eu-west-1), Milan (eu-south-1), and Hyderabad (ap-south-2). This setup includes three replicas in Europe (forming a fast majority) and two geographically distant replicas, creating significant latency variance. Figure 7 compares throughput and median latency for both normal conditions and random asynchrony.

Single leader-based protocols suffer under asynchrony. Multi-Paxos degrades from 400k to 300k cmd/sec while QuePaxa drops from 500k to 400k cmd/sec, under 500ms latency bound. This degradation stems from their reliance on single leader-based dissemination: all commands must flow through a single leader to its quorum. Under random asynchronous network conditions, the leader or its quorum members experience slow message delivery, creating a bottleneck that degrades overall

[2]Due to an implementation error, EPaxos only supports up to 5 replicas and is excluded from this experiment.

[3]The EPaxos implementation does not allow configurable command sizes and was omitted from this experiment.

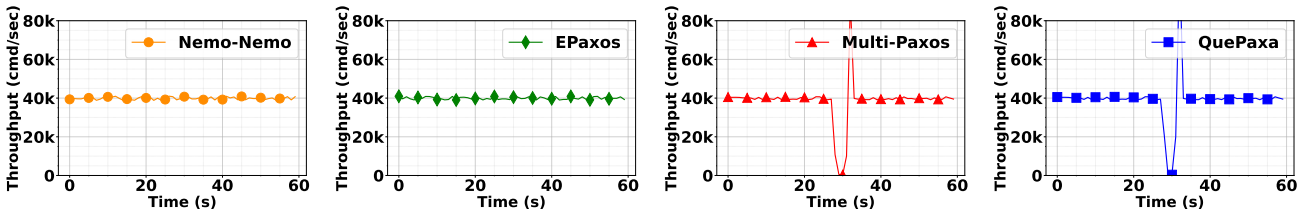

Figure 6: Throughput under crash failures with 5 replicas. At 25 seconds, we crash the leader in Multi-Paxos and QuePaxa, and a random replica in Nemo-Nemo and EPaxos. Constant arrival rate of 40k cmd/sec.

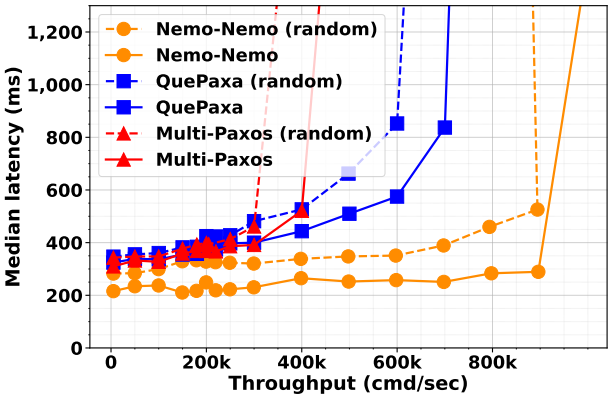

Figure 7: Performance under normal conditions and random asynchrony with 5 replicas.

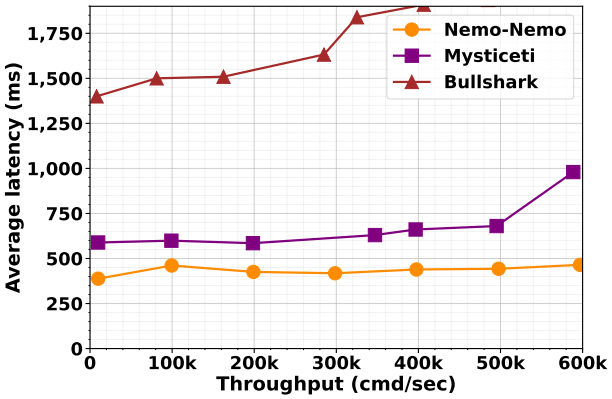

Figure 8: Comparison against state-of-the-art BFT DAG-based protocols with 10 replicas and 32 B commands.

system performance.

DAG-based dissemination provides robustness. In stark contrast, Nemo-Nemo maintains stable throughput at 900k cmd/sec, showing no throughput degradation despite network randomization. This results in Nemo-Nemo outperforming Multi-Paxos by 3× and QuePaxa by 2.25×, in throughput, under random asynchrony. Nemo-Nemo's robustness arises from its DAG-based architecture: all replicas disseminate commands concurrently, creating multiple parallel paths to commitment. Nemo-Nemo adapts to network conditions without performance degradation.

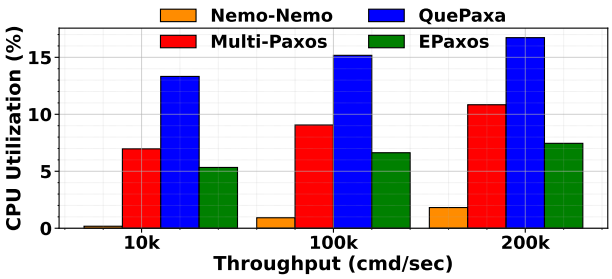

Figure 9: Average CPU utilization across all 5 replicas.

7.8 Comparison against BFT protocols

We compare Nemo-Nemo against state-of-the-art BFT DAG-based consensus protocols to understand how our CFT-optimized design compares to systems designed for BFT. We deploy 10 replicas with 32 B commands and compare against two popular BFT DAG protocols: Bullshark and Mysticeti. Bullshark is a partially synchronous protocol using Narwhal as its DAG layer. Mysticeti is an uncertified DAG protocol achieving an optimal three-round commitment. Since both Bullshark and Mysticeti report average latency, we depict the average latency in this evaluation. Figure 8 shows the latency-throughput trade-off.

Throughput. Nemo-Nemo and Mysticeti achieve at least 600k cmd/sec throughput, as both employ lightweight DAG-based architectures for parallel block processing. Bullshark attains only 450k cmd/sec (25% lower) due to its heavier Narwhal-based certified DAG design.

Latency advantages. More importantly, Nemo-Nemo achieves substantially lower latency than both BFT protocols: 1200 ms lower than Bullshark and 180 ms lower latency than Mysticeti. This latency advantage stems from two key differences. First, Nemo-Nemo's typical commit path requires only 2 message delays, fewer than both Mysticeti (3 message delays) and Bullshark (6 message delays). Second, Nemo-Nemo targets crash faults and avoids cryptographic operations entirely, while BFT protocols must authenticate every message with signatures and hashes to defend against Byzantine behavior. The combination of fewer message delays and zero cryptographic overhead enables Nemo-Nemo to achieve significantly lower latency than BFT alternatives.

7.9 Resource utilization

We measure the CPU utilization of Nemo-Nemo and follow Matte et al. [33], who use CPU utilization as a proxy for resource consumption. We record per-replica CPU utilization once per second with Go's "gopsutil" tool [16], aggregate the readings across replicas, and report the mean per-replica utilization in Figure 9.

Figure 9 shows that Nemo-Nemo uses only 1% CPU at 100k cmd/sec, compared to 9%, 15%, and 7% for Multi-Paxos, QuePaxa, and EPaxos.[4] The same pattern holds at 10k and 200k cmd/sec. Multi-Paxos, QuePaxa, and EPaxos consume more CPU because they process more messages: Nemo-Nemo broadcasts one message per block, while existing protocols require at least two propose–vote rounds to commit a batch. This extra communication directly increases their CPU cost, whereas Nemo-Nemo avoids it.

8 Related Work

Leader-based protocols. Most deployed consensus protocols rely on a leader to order requests and achieve one-round-trip normal-case commit latency [27, 39, 40]. While simple and efficient in favorable conditions, leader-based protocols suffer from two fundamental limitations: all client requests must funnel through a single leader, limiting throughput scalability, and performance degrades significantly under asynchronous or poor network conditions when the leader or its closest quorum experiences delays. As shown in Figures 3 to 5 and 7, Nemo-Nemo outperforms leader-based protocols in both throughput and latency by distributing load evenly across replicas, providing robustness under network randomization.

Multi-leader protocols. To address the leader bottleneck, several protocols enable multiple concurrent leaders. Mencius [32] statically partitions the replicated log across replicas, but its main drawback is that SMR progress depends on the slowest replica. In contrast, Nemo-Nemo progresses at the speed of the majority. Generalized Paxos [28] and EPaxos [36] support multi-leaders by exploiting request dependencies and allowing partial order of commands. Under low load, EPaxos achieves lower latency than Nemo-Nemo, but under high load, the dependency-checking overhead becomes a bottleneck, degrading performance. Fast Paxos [29] and Multi-coordinated Paxos [11] also permit multiple leaders but deliver suboptimal performance in practice [36].

Parallel dissemination. SADL-RACS [53] takes a different approach by decoupling command dissemination from the consensus critical path, following a design similar to Narwhal [14]. Unlike Nemo-Nemo, SADL-RACS does not build a DAG; instead, it uses the SADL overlay to maintain a per-replica chain of blocks, which are committed when the next RACS block is committed. This separation requires at least 5 network messages to commit a batch of commands, resulting in high latency overhead (see in Figure 3). In contrast, Nemo-Nemo embeds command dissemination directly into the DAG, avoiding extra message hops and achieving lower latency and higher throughput.

DAG-based protocols. DAG-based consensus protocols are recent advancements in Byzantine fault-tolerance (BFT), but have not been explored in the context of crash fault-tolerant (CFT) systems. Existing BFT DAG-based protocols can be classified into two categories: certified DAGs [14, 46] and uncertified DAGs [5, 23, 25]. Nemo-Nemo is the first CFT protocol to adopt a DAG-based architecture, specifically using an uncertified DAG design.

DAG-Rider [24], Tusk [14], Bullshark [46], and Dumbo-NG [19] are representative certified DAG protocols that use consistent broadcast to explicitly certify each DAG block [44]. Fides [60] is a TEE-assisted BFT DAG protocol that also uses certified blocks, achieving certification in two rounds without signatures by leveraging TEE guarantees. Explicit certification ensures that equivocating blocks cannot exist, simplifying the commit rule. However, certification adds at least 3 message delays per DAG round and increases latency, as seen in Bullshark's results in Figure 8. It also raises bandwidth and CPU costs, since replicas must disseminate, receive, and verify cryptographic certificates. To avoid these drawbacks, Nemo-Nemo employs an uncertified DAG architecture.

Cordial Miners [25], Mysticeti [5], Mahi-Mahi [23], and BlueBottle [58] operate over uncertified DAGs, where each block is disseminated using best-effort to all replicas. Nemo-Nemo also builds on an uncertified DAG, but differs fundamentally from these BFT protocols in two key ways. First, by targeting crash faults instead of Byzantine faults, Nemo-Nemo's protocol structure requires one less DAG-round to commit, reducing message delays from 3 to 2. Second, Nemo-Nemo avoids expensive cryptographic operations such as signature generation and verification, and hash computation and verification. The combination of fewer message delays and zero cryptographic overhead enables Nemo-Nemo to achieve significantly lower latency than Mysticeti (see in Figure 8).

Orthogonal goals. Nemo-Nemo focuses on high-performance, simple consensus, but does not address several other important goals: *e.g.,* scalability via partitioning commands and state [2, 18, 36, 43], shrinking quorum sizes [2, 21], exploiting WAN locality [2, 38], storage optimization with erasure coding [57, 59], or reducing replica load by outsourcing work [61]. We plan to explore how techniques from these complementary efforts can integrate with Nemo-Nemo in future work.

[4] QuePaxa adopts a gRPC-based multi-threaded design, while Multi-Paxos uses a single-threaded one, which increases QuePaxa's CPU usage.

References

[1] Ittai Abraham, Neil Giridharan, and Kartik Nayak. What's dag got to do with it? https://decentralizedthoughts.github.io/2025-08-08-DAGs/, August 2025.

[2] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. WPaxos: Wide area network flexible consensus. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):211–223, 2019.

[3] Mohammadreza Alimadadi, Hieu Mai, Shenghsun Cho, Michael Ferdman, Peter Milder, and Shuai Mu. Waverunner: An elegant approach to hardware acceleration of state machine replication. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 357–374, 2023.

[4] Amazon. AWS instance types. https://aws.amazon.com/ec2/instance-types/, 2023.

[5] Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. Mysticeti: Low-latency dag consensus with fast commit path. *arXiv preprint arXiv:2310.14821*, 2023.

[6] Leemon Baird and Atul Luykx. The Hashgraph Protocol: Efficient Asynchronous BFT for High-Throughput Distributed Ledgers. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, 2020.

[7] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 27–30. ACM, August 1983.

[8] Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, and et al. Sui lutris: A blockchain combining broadcast and consensus. In *CCS*, 2024.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference USENIX ATC 13*, pages 49–60, June 2013.

[10] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Science & Business Media, 2011.

[11] Lásaro Jonas Camargos, Rodrigo Malta Schmidt, and Fernando Pedone. Multicoordinated paxos. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 316–317, 2007.

[12] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[13] CoinEx. What Is IKA? IKA: Exploring the Fastest MPC Network on Sui Blockchain, 2025. CoinEx Academy.

[14] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *ACM EuroSys*, 2022.

[15] George Danezis, Jovan Komatovic, Lefteris Kokoris-Kogias, Alberto Sonnino, and Igor Zablotchi. Byzantine consensus in the random asynchronous model. *arXiv preprint arXiv:2502.09116*, 2025.

[16] DataDog. gopsutil. https://github.com/DataDog/gopsutil, 2025.

[17] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[18] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for Planet-Scale systems. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*, April 2020.

[19] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency. In *ACM CCS*, 2022.

[20] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. Autobahn: Seamless high speed bft. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 1–23, 2024.

[21] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible Paxos: Quorum intersection revisited. In *Proceedings of the 20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, December 2016.

[22] IOTA Stiftung. Consensus on IOTA. https://docs.iota.org/about-iota/iota-architecture/consensus, 2025. IOTA Documentation.

[23] Philipp Jovanovic, Lefteris Kokoris Kogias, Bryan Kumara, Alberto Sonnino, Pasindu Tennage, and Igor Zablotchi. Mahi-mahi: Low-latency asynchronous bft

dag-based consensus. *45th IEEE International Conference on Distributed Computing Systems*, 2025.

[24] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All You Need is DAG. In *ACM PODC*, 2021.

[25] Idit Keidar, Oded Naor, Ouri Poupko, and Ehud Shapiro. Cordial Miners: Fast and Efficient Consensus for Every Eventuality. In *DISC*, 2023.

[26] Mysten Labs. Mysticeti: Low-latency dag consensus with fast commit path. https://github.com/asonnino/mysticeti, 2024.

[27] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4*, 32:51–58, December 2001.

[28] Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

[29] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[30] Nick Lord. Binomial averages when the mean is an integer. *The Mathematical Gazette 94, 331-332*, 2010.

[31] Dahlia Malkhi and Pawel Szalachowski. Maximal extractable value (mev) protection on a dag. In *Tokenomics*, 2022.

[32] Yanhua Mao, Flavio Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for WANs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, December 2008.

[33] Venkata Swaroop Matte, Aleksey Charapko, and Abutalib Aghayev. Scalable but wasteful: Current state of replication in the cloud. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, pages 42–49, July 2021.

[34] Michael S. Paterson Michael J. Fischer, Nancy A. Lynch. Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 1985.

[35] Iulian Moraru, David G Andersen, and Michael Kaminsky. EPaxos go-lang. https://github.com/efficient/epaxos/, 2013.

[36] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372, November 2013.

[37] Iulian Moraru, David G Andersen, Michael Kaminsky, and Pasindu Tennage. EPaxos go-lang – modified for QuePaxa experiments. https://github.com/dedis/quepaxa-ePaxos-open-loop, September 2023.

[38] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. DPaxos: Managing data closer to users for low-latency and mobile applications. In *ACM SIGMOD/PODS Conference on Management of Data*, June 2018.

[39] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 8–17, January 1988.

[40] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference ATC14*, pages 305–319, June 2014.

[41] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. Rabia. https://github.com/haochenpan/rabia, 2021. Rabia implementation in the Go language (GitHub repository).

[42] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. Rabia: Simplifying state-machine replication through randomization. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 472–487, October 2021.

[43] Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Making fast consensus generally faster. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2016.

[44] Mayank Raikwar, Nikita Polyanskii, and Sebastian Müller. SoK: DAG-based Consensus Protocols. In *IEEE ICBC*, 2024.

[45] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: A cautionary tale. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 06)*. USENIX, May 2006.

[46] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT Protocols Made Practical. In *ACM CCS*, 2022.

[47] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: the partially synchronous version. arXiv preprint arXiv:2209.05633, 2022.

[48] The Sui team. Sui. https://github.com/mystenLabs/sui, 2024.

[49] The Tokio Team. Tokio. https://tokio.rs, 2024.

[50] Pasindu Tennage. Paxos and Raft, September 2023. GitHub repository https://github.com/dedis/paxos-and-raft.

[51] Pasindu Tennage. QuePaxa, September 2023. GitHub repository https://github.com/dedis/quepaxa.

[52] Pasindu Tennage, Cristina Basescu, Lefteris Kokoris-Kogias, Ewa Syta, Philipp Jovanovic, Vero Estrada-Galinanes, and Bryan Ford. Quepaxa: Escaping the tyranny of timeouts in consensus. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 281–297, 2023.

[53] Pasindu Tennage, Antoine Desjardins, and Lefteris Kokoris-Kogias. Racs-sadl: Robust and understandable randomized consensus in the cloud. In *2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*, pages 362–373, 2025.

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

[55] Giorgos Tsimos, Anastasios Kichidis, Alberto Sonnino, and Lefteris Kokoris-Kogias. Hammerhead: Leader reputation for dynamic scheduling. In *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*, pages 1377–1387, 2024.

[56] Ubuntu. Ubuntu Linux. https://releases.ubuntu.com/focal/, 2023.

[57] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha. Near-optimal latency versus cost tradeoffs in geo-distributed storage. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, February 2020.

[58] Preston Vander Vos, Alberto Sonnino, Giorgos Tsimos, Philipp Jovanovic, and Lefteris Kokoris-Kogias. BlueBottle: Fast and Robust Blockchains through Subsystem Specialization. *ArXiv preprint arXiv:2511.15361*, 2025.

[59] Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shangming Cai, Zihan Xu, and Dongsheng Wang. CRaft: An Erasure-coding-supported version of Raft for reducing storage cost and network cost. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, February 2020.

[60] Shaokang Xie, Dakai Kang, Hanzheng Lyu, Jianyu Niu, and Mohammad Sadoghi. Fides: Scalable censorship-resistant dag consensus via trusted components. *ArXiv preprint arXiv:2501.01062*, 2025.

[61] Zichen Xu, Christopher Stewart, and Jiacheng Huang. Elastic, geo-distributed RAFT. In *Proceedings of the International Symposium on Quality of Service*. Association for Computing Machinery, 2019.

[62] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *ACM PODC*, 2019.