Cabinet: Dynamically Weighted Consensus Made Fast

Gengrui Zhang
Concordia University
gengrui.zhang@concordia.ca

Shiquan Zhang
University of Toronto
shiquan.zhang@mail.utoronto.ca

Michail Bachras
University of Toronto
michalis.bachras@mail.utoronto.ca

Yuqiu Zhang
University of Toronto
quincy.zhang@mail.utoronto.ca

Hans-Arno Jacobsen
University of Toronto
jacobsen@eecg.toronto.edu

ABSTRACT

Conventional consensus algorithms, such as Paxos and Raft, encounter inefficiencies when applied to large-scale distributed systems due to the requirement of waiting for replies from a majority of nodes. To address these challenges, we propose Cabinet, a novel consensus algorithm that introduces dynamically weighted consensus, allocating distinct weights to nodes based on any given failure thresholds. Cabinet dynamically adjusts nodes' weights according to their responsiveness, assigning higher weights to faster nodes. The dynamic weight assignment maintains an optimal system performance, especially in large-scale and heterogeneous systems where node responsiveness varies. We evaluate Cabinet against Raft with distributed MongoDB and PostgreSQL databases using YCSB and TPC-C workloads. The evaluation results show that Cabinet outperforms Raft in throughput and latency under increasing system scales, complex networks, and failures in both homogeneous and heterogeneous clusters, offering a promising high-performance consensus solution.

1 INTRODUCTION

Consensus algorithms are crucial for ensuring data consistency across nodes in distributed applications. These applications, typically structured as depicted in Figure 1, consist of clients and services (coordinated nodes). Clients invoke a consensus task by sending requests to services. Then, the consensus algorithm coordinates nodes to agree on the order of the executions of these tasks. After that, nodes update their states and notify the client. With the rapid development of distributed applications, the scale of these
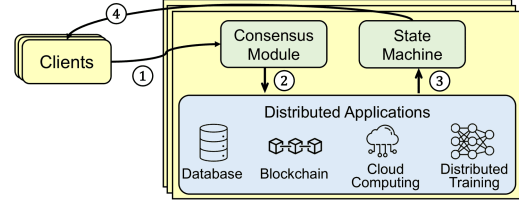
Figure 1: Structure of distributed consensus applications.

systems has witnessed a significant upsurge [3], including permissioned blockchains [4, 18], distributed storage [10, 16, 43], cloud management [19, 20], and distributed training systems [36, 55].

In particular, Paxos [34], Raft [42], and their variants [24, 52, 64, 67] have been widely used to achieve quorum replication, in which consensus follows a *simple-majority rule* and tolerates up to $f = \lfloor \frac{n-1}{2} \rfloor$ benign failures in a system of n nodes. However, as the quorum size grows linearly with the system size, simple-majority becomes inefficient when applied to large-scale systems, particularly those that are heterogeneous [68], such as global-scale databases [15, 16] and blockchains applications [4].

The root cause lies in the fact that the majority rule requires collecting $f + 1$ replies from a total of $n = 2f + 1$ nodes. As the system scales, the number of required replies increases linearly with the system size. For instance, in Google's Spanner's evaluation [15], despite the system scaling from tens to hundreds of nodes, the quorum size keeps growing (e.g., 51 replies in 100 nodes), leading to low performance, especially with high latency, though the probability of half of the nodes failing in the system is exceedingly low.

Furthermore, nodes in modern applications are often heterogeneous with varying configurations [7, 38], with some nodes possessing stronger configurations, such as higher computational capabilities and faster memory access. When the stronger nodes constitute less than a majority, they are compelled to wait for the slower nodes, leading to system-wide performance degradation [47, 49, 56].

To tackle these challenges, we propose Cabinet, a novel consensus algorithm that introduces *weighted consensus* for efficient and fast agreement. Cabinet offers a customizable failure threshold t, where t can be any integer between $1 \le t \le \lfloor \frac{n-1}{2} \rfloor$, along with a tailored weight scheme that assigns each node a distinct weight. Cabinet defines the $t + 1$ nodes with the highest weights as the *cabinet members* and enables *weight quorums*, achieving system-wide agreement as soon as the cabinet members reach an agreement.

In addition, the composition of cabinet members dynamically changes to maintain an optimal configuration. In each round, the

leader reassigns weights to nodes based on their responsiveness. This ensures that faster nodes are assigned higher weights in the subsequent execution. The $t+1$ most responsive nodes in the current execution will become the cabinet members for the next execution.

Cabinet's weighted consensus achieves fast consensus by prioritizing faster nodes with higher weights. For example, in heterogeneous clusters, strong nodes can become cabinet members when they exhibit higher responsiveness, which prevents the system from being delayed by slower nodes. Furthermore, under unstable networks, the weight reassignment can promptly adjust weights by including the most responsive nodes to cabinet members and excluding slow nodes, thereby maintaining an optimal configuration. When node responsiveness fluctuates, the composition of cabinet members adapts accordingly to always include the $t + 1$ most responsive nodes in the current round.

Compared to conventional consensus, Cabinet's weighted consensus enables more flexible fault tolerance. It tolerates a minimum of t nodes (i.e., the nodes of top t weights fail) and a maximum of $n-t-1$ nodes (i.e., the nodes of top $t+1$ weights survive). In heterogeneous clusters, since weak nodes often suffer from limited resources, overutilization, and aging hardware, they are statistically more likely to fail than strong nodes [47]. By assigning lower weights to weak nodes, Cabinet can effectively contribute to a more pragmatic and efficient fault tolerance strategy.

We implemented Cabinet based on Raft and compared their performance using the YCSB [14] and TPC-C [17] workloads on distributedly deployed MongoDB [39] and PostgreSQL [45]. Our evaluation encompassed factors such as heterogeneity, cluster scalability, complex network conditions, and node failures. Cabinet consistently delivered high performance, including higher throughput and lower latency, across all tested cases with remarkable efficiency, robustness, and adaptability, making it an ideal choice for large-scale, heterogeneous distributed systems.

To summarize, Cabinet makes the following contributions:

(1) It defines the properties and fundamentals of weighted consensus and formulates the establishment and calculation of weight schemes that achieve fast agreement with any given failure threshold.
(2) It allows a flexible fault tolerance and yields the option of choosing the failure threshold (t) to applications. It can tolerate a minimum of t failures (when high-weight nodes fail) in the worst case and $n - t - 1$ failures (when high-weight nodes survive) in the best case.
(3) It enables fast agreement with weight quorums and dynamically rearranges node weights based on responsiveness, which empowers the system to maintain optimal performance under network changes and failures.
(4) It proposes a general-purpose benchmark framework for evaluating distributed consensus applications. Compared to conventional consensus, Cabinet achieves higher throughput and lower latency when applied to database applications under both YCSB and TPC-C workloads.

The rest of the paper is organized as follows: §2 presents the background and related work; §3 defines weighted consensus and weight schemes; §4 describes the Cabinet consensus algorithm; and §5 reports the evaluation result.
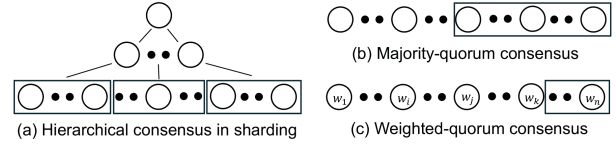

(a) Hierarchical consensus in sharding
(b) Majority-quorum consensus
(c) Weighted-quorum consensus

Figure 2: Illustrations of (a) hierarchical, (b) majority, and (c) weighted consensus mechanisms.

2 BACKGROUND AND RELATED WORK

Consensus algorithms are closely associated with state machine replication (SMR) in distributed systems [48]. They coordinate a group of nodes to agree on a value, even in the presence of failures. Consensus algorithms are majorly categorized into two classes: the one that tolerates *benign failures* (e.g., Paxos [34] and Raft [42]), such as crashes or network delays, and the other one that tolerates *Byzantine failures* [62] (e.g., PBFT [11], Hotstuff [59], and PrestigeBFT [63]), such as arbitrary behavior where nodes may behave maliciously [33, 60]. This paper focuses on the consensus algorithms that tolerate benign failures (i.e., non-Byzantine failures). Paxos and Raft serve as exemplary consensus algorithms and have been widely deployed in numerous applications.

Majority quorums and optimizations. In the Paxos/Raft consensus protocol family [31, 32, 34, 37, 42, 51], majority quorums, adhering to the simple majority rule [1], are the most common quorum replication building blocks. The quorum size typically corresponds to a majority of the total number of nodes; i.e., $\lfloor \frac{n}{2} \rfloor + 1$ in a system of n nodes [23]. Majority quorums use the concept of simple majority to prevent nodes from making conflicting decisions and achieve fault tolerance by tolerating $\lfloor \frac{n-1}{2} \rfloor$ failures.

However, majority quorums become inefficient as distributed applications continue to grow in scale, such as distributed databases [3, 53, 54], blockchains [4, 46, 50], and distributed training [36, 55, 58]. Their consensus threshold (i.e., $\lfloor \frac{n}{2} \rfloor + 1$) grows linearly as the scale of the system increases. However, in modern large-scale computer systems with a significant number of nodes (e.g., $n > 50$ nodes), the likelihood of half the nodes crashing simultaneously is highly improbable [47, 49, 56]. While maintaining fault tolerance remains critical, it is essential to reassess the quorum size and strike a more appropriate balance between fault tolerance and efficiency [9].

Tailoring to specific application scenarios, numerous consensus protocols have proposed optimizations, especially based on Paxos [6, 8, 12, 35, 40, 44]. For example, Vertical Paxos [31] enables reconfiguration that moves "vertically" to change the leader node, while each distinct vertical configuration, managed by different proposers, conducts Paxos instances along a "horizontal" axis. Furthermore, Flexible Paxos [25, 26] introduces simple quorums and grid quorums that increase the size of propose quorums (Q_1) and reduce that of accept quorums (Q_2) where $|Q_1| + |Q_2| > N$ (the number of nodes). With grid quorums, it can reduce the average quorum size to $N_1 + N_2/2$.

Sharding on scaling systems. When scaling a system, sharding—partitioning a large group of nodes into hierarchical quorums—is often applied, as exemplified by Google Spanner [16]. Compared to majority-quorum consensus, hierarchical quorum

consensus (HQC) allows consensus decisions to be made simultaneously within groups. Once a consensus decision is reached at the child-level groups, parent groups aggregate these decisions up to the root group (shown in Figure 2). HQC reduces the quorum size needed for each decision-making process, achieving a quorum size of $n^{0.63}$[29], in contrast to the $\lceil \frac{n+1}{2} \rceil$ size required in majority quorums. HQC is also listed as an option for consensus mechanisms in Zookeeper[69].

However, HQC introduces significant latency, as decisions must traverse multiple rounds with respect to the height of the hierarchical tree [16, 57]. In contrast, non-partitioned consensus algorithms, such as Paxos and Raft, typically require only two rounds of message-passing to reach a decision [30, 42]. However, they become inefficient in large-scale distributed replication due to its larger quorum size requirement. Thus, the inefficiency of large-scale replication remains a critical challenge in practice [5].

When nodes are not born equal. In today's distributed applications, heterogeneity is a common feature among nodes [66]. For instance, in distributed databases and blockchain applications, heterogeneous nodes often coexist with varying configurations among different pariciparing entities. Moreover, even within a single entity, heterogeneity prevails due to diverse hardware acquisitions and varying VM setups. Additionally, nodes frequently exhibit divergent responsiveness because of their network conditions, resulting in varying response times, with some nodes being faster at one moment and slower at another. These features heavily impact the overall quality of major quorum consensus, as strong nodes are compelled to wait for weak nodes to reach consensus. Furthermore, weak nodes in heterogeneous clusters statistically experience more failures than strong nodes because of their limited resources, overutilization, and aging hardware [47].

Weighted voting. Using weights to expedite consensus has a long history. For example, back in the 80s, weighted voting differentiated the influence of servers by assigning them weights in the voting process [21, 22, 27]. However, these early approaches reach agreement only for single instances and work only in synchronous networks. In addition, with the rise of quorum replication, investigations into weighted voting and weighted partitions gained momentum [2, 28, 61].

Compared to HQC and majority-quorum consensus [29, 69], weighted consensus addresses the trade-off between quorum size and the number of consensus rounds. Weighted consensus can reduce the quorum size while maintaining a two-round message-passing process to achieve consensus decisions.

What's still missing? Despite decades of optimization in quorum replication, state-of-the-art approaches still struggle to meet the demands of modern computing, characterized by its large-scale and often heterogeneous nature. Notably, these approaches lack the following critical properties:

P1 A universal weight assignment approach for conducting weighted consensus such that quorum replication can be reduced to any quorum size with flexible failure thresholds.

P2 A dynamic weight reassignment approach responsive to changing operating conditions such that the system maintains optimal performance.

P1 enables a valuable tradeoff between fault tolerance and replication efficiency. Particularly, in large-scale systems, assuming a failure threshold t less than the majority (i.e., $t < \lceil \frac{n+1}{2} \rceil$) is practically feasible. In addition, **P2** empowers the system to dynamically prioritize best-performing nodes in weight reassignment, supporting the consensus process to maintain optimal performance.

Motivated by these challenges, Cabinet provides a precise definition with invariants of weighted consensus that allows for customizable fault tolerance. It dynamically reassigns weights by prioritizing more responsive nodes to maintain optimal performance. It offers an efficient yet straightforward implementation of weighted consensus based on Raft [42]. Cabinet provides a practical and effective solution for high-performance consensus algorithms, particularly for large-scale and heterogeneous systems.

3 WEIGHT SCHEME

This section introduces weighted quorum and weight scheme, the most fundamental building blocks to achieve weighted consensus. Compared to conventional consensus (e.g., Paxos and Raft) that tolerates $f = \lfloor \frac{n-1}{2} \rfloor$ failures by the simple majority rule, **weighted consensus offers a new tradeoff frontier to reduce the number of minimum failures** (t where $1 \le t \le f$) **tolerated in exchange for higher performance**. For example, in a system of 10 nodes, weighted consensus may set its failure threshold to $t = 3$, whereas conventional consensus adheres to a fixed failure threshold of $f = 4$.

A **weight scheme (WS)** is a sequence of weight values that designate the weight assigned to each node, while a **consensus threshold (CT)** determines the threshold required to reach an agreement. With a given t, we designate the $t + 1$ nodes with the highest weights as **Cabinet Members**. In the following example, cabinet members are the $t + 1$ nodes that have the $t + 1$ highest weights (w_1 to w_{t+1}).

$$\text{WS} = \underbrace{w_1, w_2, \ldots, w_t, w_{t+1}}_{\text{first } t+1 \text{ values (Cabinet)}}, \underbrace{w_{t+2}, w_{t+3}, \ldots, w_{n-2}, w_{n-1}, w_n}_{\text{remaining } n-t-1 \text{ values}} \quad (1)$$

Weight schemes should uphold the safety and liveness of weighted consensus algorithms.

Safety. The WS and CT should underpin safety; i.e., no two correct nodes decide differently.

Liveness. The WS and CT should underpin liveness; i.e., a correct node eventually decides.

In conventional consensus, where every node weighs one, CT is simply the majority; i.e., $CT = \lfloor \frac{n}{2} \rfloor + 1$. CT cannot be set lower as it may allow more than one group of correct nodes to surpass CT and decide on different values [23, 42].

Similarly, not all WS and CT underpin both safety and liveness of weighted consensus. Figure 3 shows three weight schemes with $t = 2$ and $n = 7$, where n_5, n_6, and n_7 are cabinet members. For instance, in WS_1, weights are assigned based on node IDs with $CT = 8$. However, WS_1 fails to ensure safety because correct nodes may reach different decisions. For example, n_6 and n_7 ($Sum(6, 7) > 8$) may decide on value v, while n_2, n_3, and n_4 ($Sum(2, 3, 4) > 8$) may decide on another value, say w, violating safety.

Weighted voting suggests setting the consensus threshold to half of the total weights [27]. Unfortunately, this alone does not suffice to guarantee both safety and liveness. For instance, in WS_2, weights

Nodes	n_1	n_2	n_3	n_4	n_5	n_6	n_7	CT
WS$_1$	1	2	3	4	**5**	**6**	**7**	8
WS$_2$	1	10	10^2	10^3	**10^4**	**10^5**	**10^6**	55555.5
WS$_3$	2	3	4	6	**8**	**10**	**12**	22.5

Figure 3: Three weight schemes under $t = 2$ and $n = 7$, where n_5, n_6, and n_7 are cabinet members. WS$_1$ violates safety; WS$_2$ violates liveness; WS$_3$ can uphold both safety and liveness.

grow exponentially, yielding $CT = \sum w_i/2 = 55555.5$. However, this scheme sabotages liveness. With $t = 2$, the system should tolerate two failures, but in WS$_2$, even if only n_7 fails, the system halts because the combined weight of the remaining nodes falls below CT, preventing any agreement from being reached.

In Cabinet, we propose two invariants of weight schemes that can uphold both safety and liveness for weighted consensus. We also set the consensus threshold (CT) to half of the total weight.

I1 The sum of the $t + 1$ highest weights (the weights of the cabinet members) is greater than the consensus threshold.

I2 The sum of the t highest weights (the weights of the cabinet members excluding the lowest weight) is less than the consensus threshold.

In other words, a weight scheme should adhere to Eq. 2.

$$\sum_{i=1}^{t} w_i < \text{CT} = \sum_{i=1}^{n} \frac{w_i}{2} < \sum_{i=1}^{t+1} w_i \qquad (2)$$

Invariants **I1** and **I2** serve as the foundation for the correctness of **weighted quorums** in terms of safety and liveness. With **I1**, when the group of cabinet members reaches an agreement, the remaining weights are not sufficient for any other agreement (see Lemma 3.1). With **I2**, when any t nodes fail, the remaining weights are still sufficient to reach an agreement among alive nodes (see Lemma 3.2). Thus, when the cumulative weights surpass the consensus threshold, a system-wide decision is reached. Given that $t + 1$ cabinet members always constitute the smallest-sized weight quorum, the fastest system-wide decision can be made once Cabinet members agree on the decision.

For example, WS$_3$ with $CT = 22.5$ in Figure 3 is an eligible weight scheme that adheres to Eq. 2 (i.e., $Sum(12, 10) < 22.5 < Sum(12, 10, 8)$). Under normal operation, system-wide decisions can be made as fast as the cabinet members (3 nodes) have decided. Non-cabinet members cannot make a conflicting decision because their weights are less than the consensus threshold ($22.5 > Sum(6, 4, 3, 2)$). This weight scheme can tolerate at least 2 failures as the total of weights excluding the highest two is greater than the consensus threshold ($Sum(8, 6, 4, 3, 2) > 22.5$).

Next, we prove that weighted consensus achieves fast agreement where a system-wide decision can be made by the cabinet members agreeing on the decision.

LEMMA 3.1. *The sum of the weights of all non-cabinet members is less than the consensus threshold.*

PROOF. We prove this lemma by contradiction. We denote the sum of the weights of the cabinet members is W_{cab}, and the sum of all weights is W_{all}. Given **I1**, $W_{cab} > W_{all}/2$. We claim that the

sum of the weights of the remaining $n - t - 1$ values is greater than the majority. As a result, $W_{all} - W_{cab} > W_{all}/2$; i.e., $W_{cab} < W_{all}/2$, which violates **I1**, Thus, this lemma holds. □

THEOREM 3.1 (FAST AGREEMENT). *In every execution, if all cabinet members agree on a decision, then all non-faulty nodes agree on the same decision.*

PROOF. We prove this theorem by contradiction. We claim that all cabinet members agree on a decision $\mathcal{D}$, but at least one non-faulty node has decided on a conflicting decision, $\mathcal{D}_\diamond$.

Since $\mathcal{D}$ has been decided by cabinet members, the $t + 1$ highest weights will not be voting for $\mathcal{D}_\diamond$. From Lemma 3.1, the sum of the weights of the remaining $n - t - 1$ nodes (i.e., non-cabinet members) cannot surpass the consensus threshold. Thus, the decision for agreeing on $\mathcal{D}_\diamond$ cannot be made, which contradicts our claim. □

Next, we prove that any weight scheme with a failure threshold of t tolerates at least t failures.

LEMMA 3.2. *The sum of the weights of any combination of $n - t$ nodes is greater than the consensus threshold.*

PROOF. We denote the nodes that are the cabinet members as set N_{cab}, where $|N_{cab}| = t + 1$ with a total weight of W_{cab}, and the node that has the lowest weight in N_{cab} as n_{t+1} with weight w_{t+1} (the $(t + 1)$-th highest weight among all nodes). Given **I2**, following a similar way of proof, we know that the sum of weights of all non-cabinet members and n_{t+1} (in a total of $n - t$ nodes) is greater than the consensus threshold. That is,

$$\sum w_i + w_{t+1} > CT, \quad \text{for every } i \text{ where } n_i \notin N_{cab}$$

Since n_{t+1} has the lowest weight in N_{cab}, every node that is not a cabinet member (i.e., $n_i \notin N_{cab}$) has a lower weight than n_{t+1}. Then, the above combination has the lowest weight of any combination of $n - t$ nodes. Therefore, the total weight of any combination of $n - t$ nodes must be greater than the consensus threshold. □

THEOREM 3.2 (FAULT TOLERANCE). *In every execution, weighted consensus tolerates at least t failures.*

PROOF. In weighted consensus, an agreement can be reached when the sum of the weights of the non-faulty nodes is greater than the consensus threshold. Given Lemma 3.2, any t failed nodes will not prevent an agreement from being achieved. Therefore, weighted consensus tolerates at least t failures. □

Flexible fault tolerance. Weighted consensus offers a unique tradeoff between improving performance and tolerating failures. It can tolerate a minimum of t failures (worst-case) and potentially a maximum of $n - t - 1$ failures (best-case). The worst case arises when the top t nodes with the highest weights experience failures (e.g., n_6 and n_7 fail in WS$_3$). In contrast, the best case occurs when all the cabinet members survive (e.g., n_1, n_2, n_3, and n_4 fail in WS$_3$). In heterogeneous clusters, since strong nodes with higher weights are often less likely to fail than weak nodes, applications can reduce the quorum size and operate with fast agreement.

Weight schemes can be implemented in any ways as long as they satisfy **I1** and **I2**. In the following section, we introduce a simple and generalized implementation of weight schemes and present Cabinet in detail.

t	r	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}
1	1.40	**20.7**	**14.8**	10.5	7.5	5.4	3.8	2.7	2.0	1.4	1
2	1.38	**18.2**	**13.2**	**9.5**	6.9	5.0	3.6	2.6	1.9	1.4	1
3	1.19	**4.8**	**4.0**	**3.4**	**2.8**	2.4	2.0	1.7	1.4	1.2	1
4	1.08	**2.0**	**1.9**	**1.7**	**1.6**	**1.5**	1.4	1.3	1.2	1.1	1

Figure 4: Cabinet weight schemes with different customized failure thresholds in a $n = 10$ system. The weights of cabinet members in each scheme are in blue font.

4 THE CABINET CONSENSUS ALGORITHM

We implemented Cabinet based on Raft [42]. Nodes in Raft operate in one of three states: *leader*, *follower*, and *candidate*. In the normal case, there is one leader and the other nodes are followers. The leader issues AppendEntriesRPCs to followers, completing consensus as soon as a majority of the nodes respond to a single round of RPC calls. When the leader fails, followers become candidates and issue RequestVoteRPCs to start leader elections; the candidate that collects a majority of votes will become the new leader.

Cabinet implements weighted consensus on Raft by adding only two parameters to Raft's AppendEntriesRPCs. Raft's communication pattern and validation rules remain intact. In addition, Cabinet focuses on replication and does not change Raft's leader election mechanism. Cabinet preserves Raft's strong leadership concept: the leader, after being elected, computes a weight scheme based on a given failure threshold. It dynamically adjusts the weights of individual nodes based on their responsiveness, maintaining optimal performance throughout the consensus process.

4.1 The Algorithm

Cabinet first initializes a weight scheme according to the given failure threshold t. It uses geometric sequences to construct demanded weight schemes that satisfy Eq. 2. During the consensus process, Cabinet keeps reassigning higher weights to more responsive nodes, maintaining fast agreement and customized fault tolerance.

4.1.1 Cabinet weight schemes. Cabinet implements weight schemes according to a given t where $1 \leq t \leq \lfloor \frac{n-1}{2} \rfloor$ by applying geometric sequences. When its common ratio $1 < r < 2$, a geometric sequence monotonically increases ($1 < r$) with no value greater than the sum of its previous values ($r < 2$). Initially, nodes are assigned weight values in descending order according to their node IDs, with higher IDs receiving lower initial weights.

Weights:	w_1	w_2	$\dots$	w_{n-2}	w_{n-1}	w_n
Sequence:	$a_1 r^{n-1}$	$a_1 r^{n-2}$	$\dots$	$a_1 r^2$	$a_1 r$	a_1

To make the sequence eligible for the given t, the relation of weight values must be in accordance with Eq. 2, where the sum of the t largest weights should be less than the consensus threshold while the sum of the $t + 1$ largest weights should be greater than the consensus threshold (shown in Eq. 3).

$$\sum_{i=n-t}^{n-1} a_1 r^i < CT = \frac{1}{2} \sum_{i=0}^{n-1} a_1 r^i < \sum_{i=n-t-1}^{n-1} a_1 r^i \quad (3)$$

Thus, the relation of r, n, and t must satisfy Eq. 4.

$$r^{n-t-1} < \frac{1}{2}(r^n + 1) < r^{n-t} \quad (4)$$

Since the base value a_1 does not affect the progression of the sequence, for simplicity, we choose $a_1 = 1$. Figure 4 shows examples of geometric sequences being used as weight schemes in a system of $n = 10$ under $t = 1, 2, 3, 4$. By changing the ratio r, the geometric sequence adapts to generate eligible weight schemes that can be used to achieve weighted consensus.

After the initial weight assignment, the weight of each node may change dynamically according to their responsiveness during the consensus process. It is worth noting that the leader redistributes the initial weights among nodes without generating any new weights. So far, Cabinet's weight scheme satisfies Property **P1**, providing a universal approach for quorum replication that underpins safety and liveness under customized failure thresholds. Next, we introduce the complete fast consensus algorithm with weight reassignment that satisfies Property **P2**.

4.1.2 Fast consensus with weight reassignment. Cabinet adopts Raft's AppendEntries RPCs and *adds only two parameters* to achieve the dynamic weight reassignment. The workflow with the parameters is described in Algorithm CABINET-CONSENSUS. Cabinet introduces *weight clock*, denoted by wclock, and *weight value*, denoted by w_i to Raft's original AppendEntries RPCs (Line 2 and 3). The weight clock is simply a logical clock of the round of consensus instances. It is worth noting that Cabinet does not intervene in the original consensus tasks or other parameters defined by the Raft protocol, which upholds Raft's simplicity and correctness.

Initially, the leader calculates a weight scheme according to the given failure threshold t and assigns nodes their initial weights (Line 7); it always assigns itself the highest weight (w_λ). After that, the weighted consensus starts with the leader retrieving each node's weight from the GETWEIGHT function. Then, the leader issues the RPCs to update followers' weights and complete regular Raft's consensus tasks. Once the call completes, the leader adds (*weight*, *node*) to the queue, wQ (Line 22 to 25). Thus, the top x pairs in wQ represent the first x replying followers in the weight clock *wclock*.

The leader keeps accumulating the weights of the replying nodes until they surpass the consensus threshold. The UPDATEWGT function assigns the weights from the highest to the lowest on the weight scheme (*ws*) to nodes based on their orders dequeued from wQ. For example, the first node dequeued from wQ will be the first to execute the UPDATEWGT function and is assigned the second highest weight (the leader always takes the highest). Since wQ is First in, First out, if node n_i has replied faster than node n_j, then n_i will be assigned a higher weight than n_j. Therefore, **in each weight clock, a node's weight value is dynamically adjusted based on its responsiveness.** The leader assigns a higher weight value to a node in wclock $k + 1$ if it received its reply earlier than others in wclock k. Thus, the leader and the top t nodes whose replies are enqueued in wQ in wclock k become the cabinet members in wclock $k + 1$.

When the accumulated weights (including the leader) exceed the consensus threshold, the weighted consensus is reached (Line 18). Then, the leader updates the weights for the remaining nodes (Line 20) whose weights were not previously updated at Line 17. Note that the remaining nodes are not cabinet members. All these remaining nodes are assigned lower weights compared to the nodes that got their weights updated at Line 17.

Algorithm 1 CABINET-CONSENSUS

1: **struct** AppendEntriesRPC:
   . . .   // original parameters in Raft's AppendEntries RPCs
2:   **int** wclock   // added by Cabinet
3:   **float** weight   // added by Cabinet
   ▽ As a leader
4: wclock := 0
5: ws := INITWS(wclock, n, t)   ▷ Generate
   $ws[0] = \{w_1, w_2, ..., w_n\}$
6: **for each** n_i **do**   ▷ Initialize node weights according to ws
7:    UPDATEWGT(wclock + 1, n_i, ws)   ▷ n_i's weight is w_i in ws

8: **procedure** WEIGHTEDCONSENSUS
9:    wclock++   ▷ Increment weight clock
10:   wQ := []   ▷ A queue for receiving replies
11:   **for each** follower n_i **do**
12:      w_i := GETWEIGHT(wclock, n_i)   ▷ Get n_i's weight
13:      **issue** APPENDENTRIESRPC(wclock, w_i, &wQ, . . .)
14:   sum := w_λ ▷ w_λ is the leader's weight
15:   **for** (w_i, n_i) ← wQ **do**
16:      sum+ = w_i
         // Faster nodes are assigned with higher weights.
17:      UPDATEWGT(wclock + 1, n_i, ws)
18:      **if** sum > CT **then**   ▷ Consensus is reached
19:         **break**
      // Assign weights to remaining nodes not in wQ.
20:   **for each** remaining n_i **do**
21:      UPDATEWGT(wclock + 1, n_i, ws)
   // Cabinet adds two new parameters, w_i and wclock to the leader's
   APPENDENTRIESRPCs. Replies are handled in a FIFO queue, wQ.
22: **function** APPENDENTRIESRPCS(wclock, w_i, *wQ, . . .)
23:   RPC.NEWWEIGHT(wclock, w_i)   ▷ See Line 29
24:   RPC.RAFTTASK(. . .)   ▷ Original Raft operations
25:   wQ ← (w_i, n_i)   ▷ First reply, first enqueue
   // The leader assigns weights to followers in each wclock First Come First
   Serve based on the executing order, where ws[0] and ws[n − 1] are the
   highest and the lowest weight, respectively. In each round, initially, k = 0.
26: **function** UPDATEWGT(wclock, n_i, ws)
27:   n_i = ws[wclock][k]
28:   k + +
   ▽ As a follower
   // Store the weight clock and the weight value issued by the leader.
29: **function** NEWWEIGHT(wclock, w_i)
30:   state.wc = wclock
31:   state.w = w_i

More examples. Figure 5 shows examples of weight rearrange-
ment in a Cabinet system of $n = 7$ nodes where $t = 2$. We use the
weight scheme of the example introduced in §3 with a $CT = 22.5$.

(a) When wclock = 1, the cabinet members are n_1, n_2, and n_3 and
    have the $t + 1$ highest weights among all nodes. If the cabinet
    members reply to the leader faster than the other nodes by
    the end of this execution, they remain to be cabinet members
    for the next execution (i.e., wclock = 2).

(b) If n_3, as a cabinet member, fails to reply to the leader faster
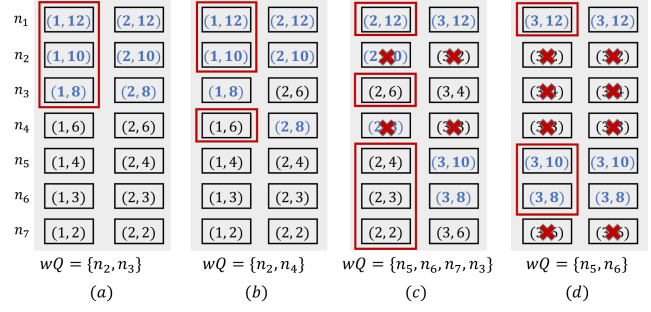   than n_4 in wclock = 1, it will be excluded from the cabinet

Figure 5: Possible weight distribution when $n = 7$ and n_1 is
the leader. The leader dynamically rearranges node weights
(wclock, w_i) based on the responsiveness received in wQ.

members with its weight assigned to n_4 in the next execution
(wclock = 2).

(c) Continuing from **(b)**, if n_2 and n_4 (t cabinet members) crash,
    the leader can still coordinate consensus by collecting replies
    from the other nodes ($Sum(12, 6, 4, 3, 2) > CT$). Based on the
    reply order in wQ, n_5 and n_6 will become the cabinet members
    in the subsequent execution.

(d) Continuing from **(c)**, if all non-cabinet members fail, the con-
    sensus process will not be affected. Although the number of
    failed nodes ($f = 4$) has exceeded the failure threshold ($t = 2$),
    consensus can still be reached by cabinet members. In this case,
    Cabinet achieves a higher fault tolerance than conventional
    consensus approaches, which tolerate up to $f = 3$ failures.

Write and read. In Cabinet, each node is required to store the
consensus result along with the weight assigned to that particular
consensus decision. Unlike conventional consensus where every
node weighs the same, weighted consensus is no longer solely
based on counting the number of nodes; therefore, it becomes
necessary to explicitly indicate the node's weight associated with
the corresponding consensus result.

Once a client request is committed by the system, clients can
read the consensus results. Read operations from clients should
accumulate the stored weight of replies from nodes. Clients confirm
that a proposed request is succeeded when the accumulated weights
surpass the consensus threshold.

Discussion. By adding only two parameters to Raft's original
RPCs, Cabinet efficiently implements weighted consensus with
dynamic weight reassignment. Notably, the processes of weight re-
assignment and replication consensus are atomic, occurring within
the same consensus instance and in the same RPC. This means that
if the consensus to commit a value is successful, the weight update
is applied; otherwise, neither operation succeeds.

With dynamic weight reassignments, Cabinet can seamlessly
adapt to changing operating conditions. Compared to approaches
that predefine voting or non-voting nodes, Cabinet eliminates the
need for this rigid distinction. In a consensus round, nodes included
in the quorum size are Cabinet members, which can be considered
as the "voting" nodes, while the remaining excluded ones can be
considered as "non-voting" nodes. Particularly, in heterogeneous
clusters, where nodes possess varying capabilities, Cabinet's ability

to assign weights to individual nodes allows for a more accurate representation of their influence in the consensus process. Strong nodes in these clusters will consistently maintain higher weights, solidifying their position as influential (cabinet) members.

In addition, Cabinet maintains the understandability of Raft; it preserves Raft's workflow and adheres to the concept of strong leadership, ensuring that the fundamentals of Raft remain intact. Cabinet's implementation of weighted consensus can seamlessly integrate within the existing Raft framework without altering its overall structure and clarity.

4.1.3 Leadership change. Cabinet adopts Raft's leader election mechanism without incorporating weighted consensus. Given the infrequency of leader elections and to maintain simplicity, Cabinet applies weighted consensus exclusively to replication. To comply with the quorum size of weighted consensus, Cabinet adjusts Raft's leader election mechanism by setting the quorum size to $n - t$; i.e., a candidate must collect votes from $n - t$ nodes to become the new leader, akin to Raft's majority quorum requiring $n - f$ votes. This adjustment ensures that an elected leader possesses the most up-to-date log as the minimum quorum size in replication is $t + 1$.

Furthermore, since Cabinet's weight reassignment prioritizes log responsiveness, in heterogeneous systems, the node with the most up-to-date log is one of the cabinet members. Although it can not be guaranteed that cabinet members are all strong nodes under complex networks (e.g., strong nodes may experience high delays and become less responsive), it increases the likelihood of a new leader being a strong node. We discuss and prove the correctness of this modification for Raft's leader election in §4.3.

4.1.4 Reconfiguration of failure thresholds. In addition to Raft's reconfiguration method that changes the cluster size, Cabinet introduces a lightweight reconfiguration for changing the failure threshold. First, the leader generates a new configuration, denoted by C', with a new WS and CT that adhere to Eq. 4. It then starts to coordinate the consensus of changing the system configuration to C'. It first broadcasts C' and waits for consensus to be reached under the new WS. Note that no replication will be conducted during the transition from the old configuration to the new configuration. Once consensus of C' is achieved under the new WS, the leader transitions to operating under the new failure threshold, similar to Raft's reconfiguration procedure. Cabinet's reconfiguration of failure thresholds offers applications an adaptive way to tradeoff between absolute fault tolerance and system performance in evolving operating environments.

4.2 Flexible Fault Tolerance

Cabinet obtains flexible fault tolerance in two distinct dimensions. The first dimension is the customized failure threshold (t), which empowers applications to define their own failure thresholds based on their preferences and user experience. The customization becomes especially valuable when large-scale applications that may not require tolerance for $\lfloor \frac{n-1}{2} \rfloor$ simultaneous failures but prioritize higher performance.

The second dimension lies in its flexible fault tolerance capabilities in any established weight scheme, allowing for the tolerance of at least t and up to $n - t - 1$ physical node failures. In the worst-case
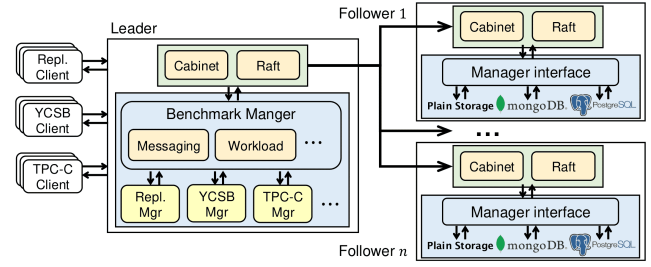

Figure 6: The architecture of Cabinet benchmark framework.

scenario, Cabinet can tolerate t failures, which occurs when the top t nodes with the highest weights experience failures. However, in the best-case scenario where all cabinet members remain operational, the system demonstrates its resilience by tolerating up to $n - t - 1$ physical node failures. This surpasses the fault tolerance limits of conventional consensus approaches, as exemplified in the scenarios presented in Examples **(c)** and **(d)**. The ability of Cabinet to dynamically adjust weights and adapt to changing node conditions empowers it to provide a fault tolerance mechanism that outperforms conventional consensus algorithms in accommodating both extreme and more favorable failure scenarios.

The two-dimensional liquefaction of fault tolerance is especially advantageous in heterogeneous clusters. Strong nodes, assigned with high weights, have a higher likelihood of resilience, while weaker nodes, assigned with lower weights, are more susceptible to failure. Consequently, Cabinet will operate closer to the best-case scenarios, where failures are less likely among cabinet members, allowing it to exceed the expected t tolerance in practice.

4.3 Correctness Argument

Due to space limitations, the correctness argument of Cabinet is presented in the extended version [65], where we show that Cabinet holds validity, safety, and liveness. We also show that Cabinet maintains the correctness of Raft's leader election mechanism.

5 EVALUATION

We implemented Raft and Cabinet in Golang and evaluated their performance on a popular cloud platform [13]. All nodes were equipped with 2.40 GHz Intel Xeon (Skylake) processors, running Ubuntu 18.04.1 LTS. The TCP/IP bandwidth is $\approx$ 400 Megabytes/s, with a raw network latency < 1 ms. Our experiments are conducted in both homogeneous and heterogeneous clusters, where VMs are grouped into distinct *zones* of different configurations.

Heterogeneous clusters: Heterogeneity across zones is defined by variations in the number of vCPUs, RAM, and disk space. For each scale $n = 3, 5, 7, 11, 20, 50, 100$, the five configurations are evenly distributed among the VMs, where "#xc-#$y$gb-#z" represents "#x vCPU, #yGB RAM, and #zGB Disk".

Homogeneous clusters: All VMs have the same configuration of Z3. The uniform configuration remains consistent across the evaluated cluster scale, ensuring that all VMs within the cluster have identical configurations.
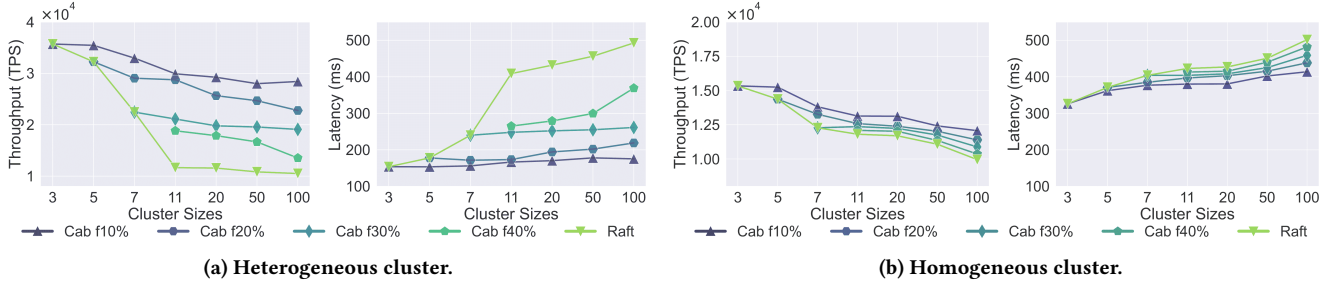
(a) Heterogeneous cluster.

(b) Homogeneous cluster.

Figure 7: Performance of YCSB+MongoDB under increasing cluster sizes with $b = 5k$, $d = 0$ under Workload A.

(a) Heterogeneous cluster.
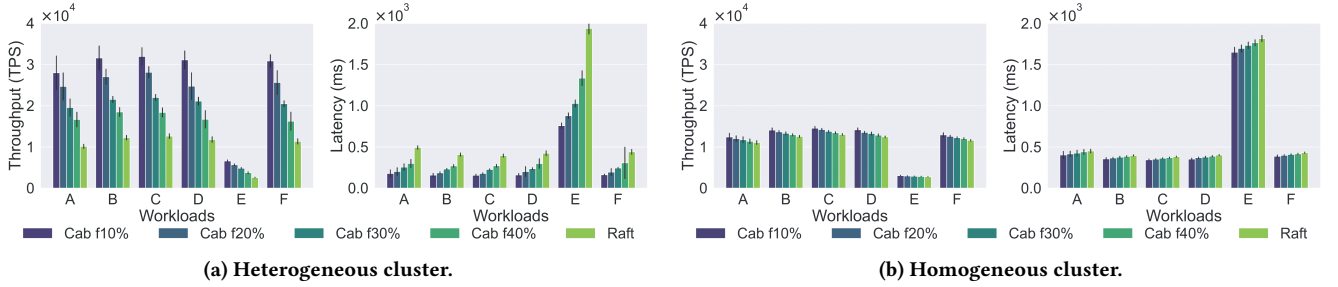
(b) Homogeneous cluster.

Figure 8: Performance of YCSB+MongoDB under $n = 50$, $b = 5k$, $d = 0$ under all YCSB workloads.

Zone	Heterogeneity	3	5	7	11	20	50	100
Z1	1c-7.5gb-56	1	1	2	2	4	10	20
Z2	2c-15gb-92		1	1	2	4	10	20
Z3	4c-15gb-164	1	1	2	2	4	10	20
Z4	8c-30gb-308		1	1	2	4	10	20
Z5	16c-60gb-596	1	1	2	3	4	10	20

We introduce the following notations to report the results:
"n" represents the cluster size; i.e., the number of VMs.
"d" represents the implemented network delays (see §5.3).
"b" represents the batch size.
"cab fx%" represents that Cabinet's $t = x$% of n; e.g., "cab $f10$%" under $n = 50$ means Cabinet's $t = 5$.

The rest of this section is organized as follows. It introduces the proposed benchmark framework for distributed consensus applications (§5.1). It reports the performance of Cabinet with varying failure thresholds compared against Raft in both heterogeneous and homogeneous clusters (§5.2), complex network conditions of differently distributed and dynamically changing network delays (§5.3), and node failures under various crash scenarios (§5.4).

5.1 Cabinet Benchmark Framework

We propose a benchmark framework tailored for distributed consensus applications, with a special focus on leader-based consensus algorithms. Our framework introduces a set of versatile and adaptable interfaces that seamlessly integrate existing benchmarks. In the context of distributed database applications, it has integrated the widely adopted YCSB [14] and TPC-C [17] workloads.

The architecture of our framework is depicted in Figure 6, providing a comprehensive overview of its key components and their interactions. The leader node takes charge of orchestrating and

managing different integrated benchmarks. Each integrated benchmark is equipped with its dedicated manager. These managers act as control centers, enabling users to fine-tune and customize evaluation parameters for specific requirements, such as batch sizes, workload sizes, ratios of workload types, and combinations of transactions. In addition, the leader batches workload data proposed from clients and issues RPCs piggybacking the consensus metadata and the batched workload data to followers. Followers encompass distributed applications that execute the transmitted workload data.

The proposed framework reports throughput and latency for the consensus process and is also able to report other criteria contained in the integrated benchmark. By leveraging our proposed framework, we can conduct evaluations under various workloads, either by creating custom workloads or by utilizing existing workloads provided by the integrated benchmarks. This flexibility empowers users to simulate real-world scenarios and accurately evaluate the performance of consensus algorithms in different contexts.

In this evaluation, we deployed the consensus module with Cabinet and Raft [42], where Cabinet has four failure thresholds of $t = 10\%n, 20\%n, 30\%n, 40\%n$ for each cluster size of $n = 10, 20, 50, 100$. The applied databases on followers are MongoDB [39], which is a popular document-oriented NoSQL database, and PostgreSQL [45], which is a widely used relational database management system. In addition, our evaluation paired MongoDB with YCSB [14] and PostgreSQL with TPC-C [17] benchmarks.

YCSB+MongoDB: YCSB is often used to compare the relative performance of NoSQL database management systems. It consists of six standard workloads (from **A** to **F**), each with a specific distribution of four operation types: READ, UPDATE, SCAN, and INSERT (details presented in [65]). In the YCSB+MongoDB evaluation, we
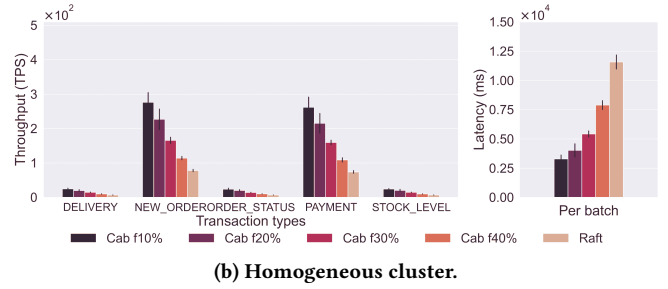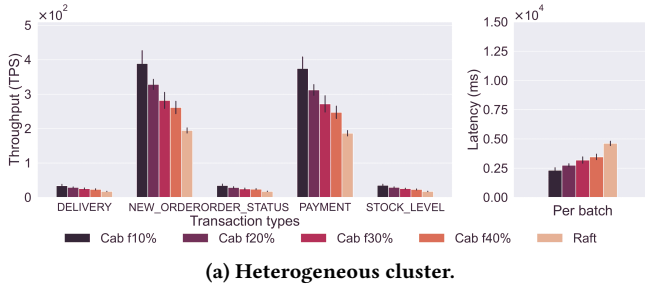
(a) Heterogeneous cluster.

(b) Homogeneous cluster.

Figure 9: Performance of TPC-C+PostgreSQL under $n = 50$, $b = 2k$, $d = 0$ under all TPC-C workloads.

(a) Heterogeneous cluster (left $n = 11$, right $n = 50$).

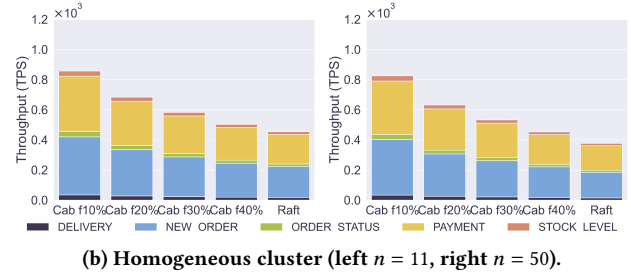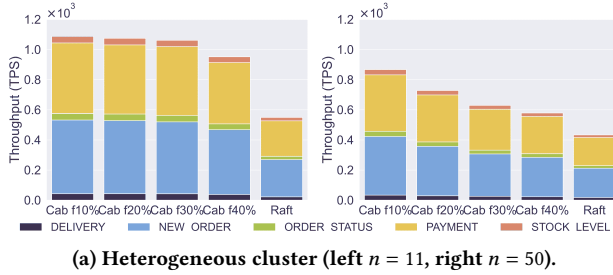(b) Homogeneous cluster (left $n = 11$, right $n = 50$).

Figure 10: Performance of TPC-C+PostgreSQL under two cluster sizes ($n = 11$, 50) with $b = 2k$, $d = 0$ under all workloads.

conducted 10 runs of a specific workload, with each run involving the leader processing a total of 500,000 corresponding operations, where the batch size was set to $b = 5,000(5k)$. As a result, each run concluded after completing 100 rounds of processing.

TPC-C+PostgreSQL: TPC-C [17] is one of the most widely used benchmarks for evaluating the performance and scalability of transactional databases for online transaction processing. It simulates a complete order-entry system, including a predefined mix of transactions performed by concurrent users. In the evaluation of `TPC-C+PostgreSQL`, a total of 50,000 transactions are processed by the leader in batches of size $b = 2,000$ (2k). The transaction mix within each batch follows the same predefined ratio. Each follower instance of PostgreSQL is configured with 10 warehouses and 16 users. We conducted 10 independent runs for each experiment.

5.2 Performance under Scaling Clusters

We evaluated the performance of YCSB and TPC-C across different scales, where $n = 3, 5, 7, 11, 20, 50$, and 100, using various workloads. Due to space limitations, we present the results of only Workload A in Figure 7. With $n = 3$, Cabinet and Raft have the quorum size of 2 with near identical performance. With $n = 5$, Cabinet can choose $t = 1$ or $t = 2$, where $t = 2$ is the same as Raft. When $n \geq 5$, Cabinet starts to show the advantage of weighted consensus by reducing the quorum size.

Under a larger scale, compared to Raft, Cabinet witnesses a more pronounced advantage in the heterogeneous setting. It is worth noting that the performance loss when scaling up the system is minimal for both Cabinet and Raft. This is because both Cabinet and Raft achieve agreement in just one round of remote procedure calls, making them linearly scalable. We report the results of all YCSB workloads under $n = 50$ in detail in Figure 8. Cabinet consistently
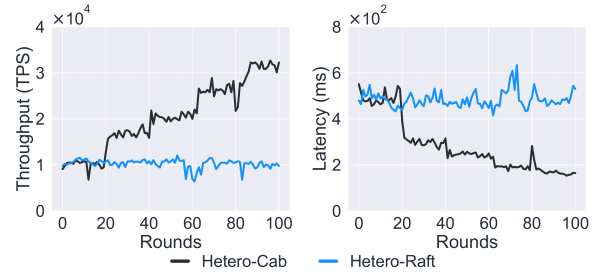

Figure 11: Throughput and latency in heterogeneous clusters of YCSB+MongoDB with $d = 0$, $n = 50$, $b = 5k$ under Workload A, where t=25 (initial), 20, 15, 10, and 5 for every 20 rounds.

outperforms Raft across all workloads in terms of both throughput and latency. This advantage is particularly evident in heterogeneous settings. When compared to Raft, Cabinet with `t=f10%=5` achieves approximately 3× higher throughput and 3× lower latency across all workloads in heterogeneous clusters. For instance, in Workload A and F, Cab f10% achieved throughputs of 27,999 and 30,864 TPS, while Raft achieved throughputs of 10,136 and 10,335 TPS, respectively. Furthermore, similar performance improvements are observed in the TPC-C workloads, as depicted in Figure 9. By employing a smaller failure threshold, Cabinet progressively enhances the overall performance for all types of transactions.

In addition, the heterogeneous cluster demonstrates superior performance compared to the homogeneous cluster under both YCSB and TPC-C, with the advantage being more pronounced in YCSB. TPC-C workloads includes certain transactions that heavily rely on locks, making it challenging to execute them in parallel. Since the heterogeneity of the deployed clusters primarily lies in
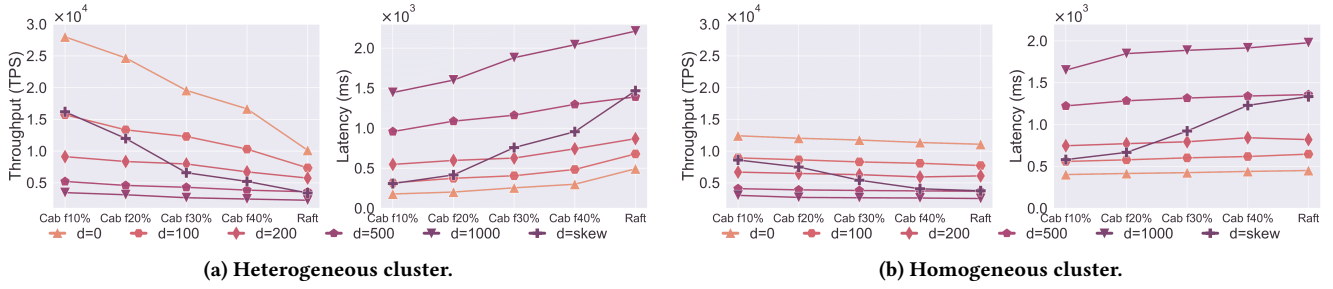
(a) Heterogeneous cluster.

(b) Homogeneous cluster.

Figure 12: Performance of YCSB+MongoDB under $d = $ D1, D2 with $n = 50$, $b = 5k$ under Workload A.
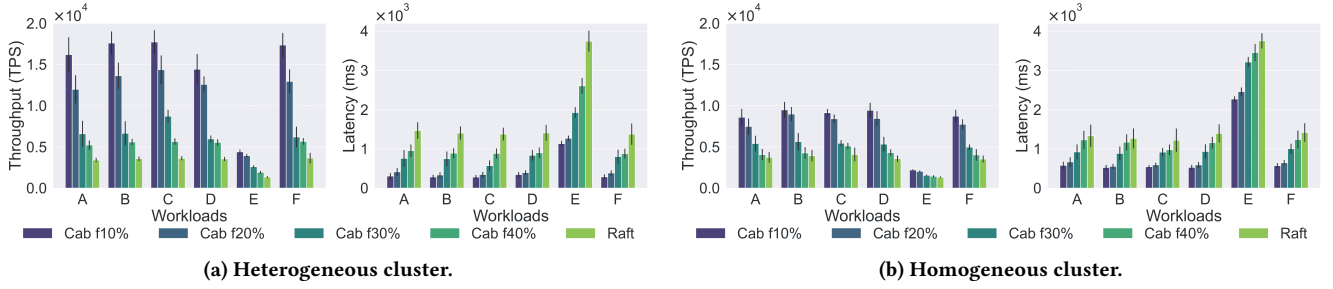

(a) Heterogeneous cluster.

(b) Homogeneous cluster.

Figure 13: Performance of YCSB+MongoDB under skew latency ($d = $ D2) with $n = 50$, $b = 5k$ under all YCSB workloads.

D1 : Uniformly distributed delays
D2 : Skew delays
D3 : Dynamically changing delays
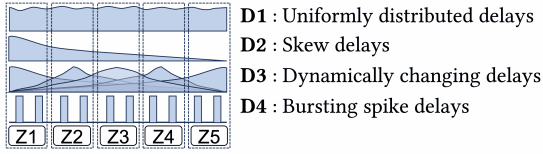**D4** : Bursting spike delays

Figure 14: Emulated complex network conditions.

the number of virtual CPUs (vCPUs) rather than hardware differences, the advantage of the heterogeneous cluster is diminished in the context of TPC-C. Nevertheless, the heterogeneity brings in a remarkable 2.3× higher throughput in YCSB (Figure 8a vs. 8b) and a 1.4× higher throughput in TPC-C (Figure 9a vs. 9b).

In the TPC-C+PostgreSQL evaluation, similar to the evaluation in YSCB, both algorithms did not encounter a notable performance drop when cluster sizes scale from 10 to 100. Figure 10 shows the breakdown of the different types of transactions. Cabinet sustained the performance gain in both heterogeneous and homogeneous clusters. The performance gain obtained is proportional to the mix of the transaction types.

Dynamic failure thresholds. Cabinet can dynamically adjust its failure threshold using its lightweight reconfiguration mechanism (§4.1.4). We conducted performance evaluations using YCSB under Workload A with MongoDB, as illustrated in Figure 11. Initially, $t = 24$ (consistent with Raft), and subsequently decreased four times ($t = 20, 15, 10, 5$) every 20 rounds. The results indicate an increase in throughput as t decreases, accompanied by a continual reduction in latency. This evaluation highlights the persistent performance gains achievable by selecting lower values for t.

5.3 Performance under Complex Networks

To assess the performance of Cabinet and Raft in complex network scenarios, we introduced additional network delays using netem [41]. These delays were categorized as follows:

D1 Uniformly distributed delays. The emulated delays are evenly implemented on all nodes, comprising four sets of increasing delays with variances: $d = 100 \pm 20$ ms, 200 ± 40 ms, 500 ± 100 ms, 1000 ± 200 ms.

D2 Skew delays. The delays were implemented in a skewed and uneven manner across the nodes. The delay values declined from $1,000 \pm 200$ ms to 100 ± 20 ms progressing across the nodes (illustrated in Figure 14).

D3 Dynamically changing delays. The delays imposed in **D2** undergo periodic changes. The intervals of these changes were carefully configured to ensure that each zone experienced the full range of emulated delay values, encompassing both the highest and lowest delays.

D4 Bursting delays. The delays in each zone repeatedly spiked for a brief period before dissipating. This behavior emulates heavily unstable networks.

Compared to normal cases without network delays (as discussed in §5.2), Cabinet demonstrated even greater superiority over Raft, with significantly improved throughput and latency. Figure 12 illustrates the performance of Cabinet and Raft under network delays of $d = $ **D1** and **D2** in clusters of $n = 50$.

Under increased network delays, both Cabinet and Raft experienced performance degradation. Nonetheless, compared to Raft, Cabinet demonstrated strong resilience, with a lower increase in latency in both homogeneous and heterogeneous clusters. In addition,
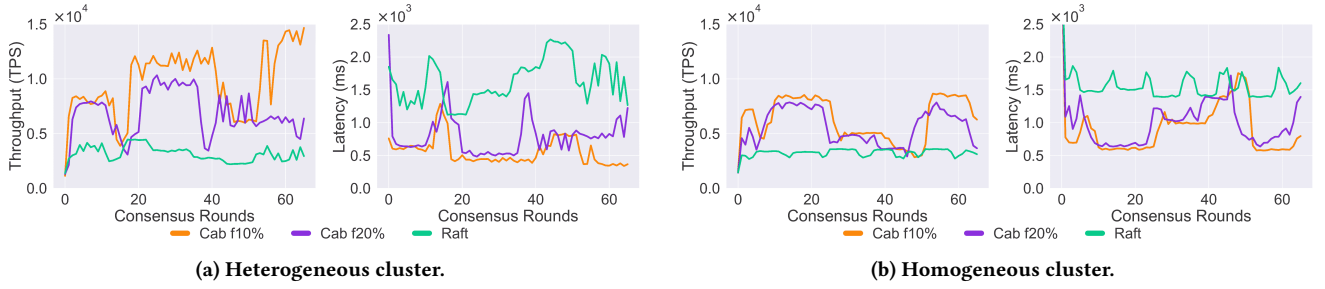
(a) Heterogeneous cluster.

(b) Homogeneous cluster.

Figure 15: Real-time performance of YCSB+MongoDB under d = D3, n = 50, b = 5k under Workload A.

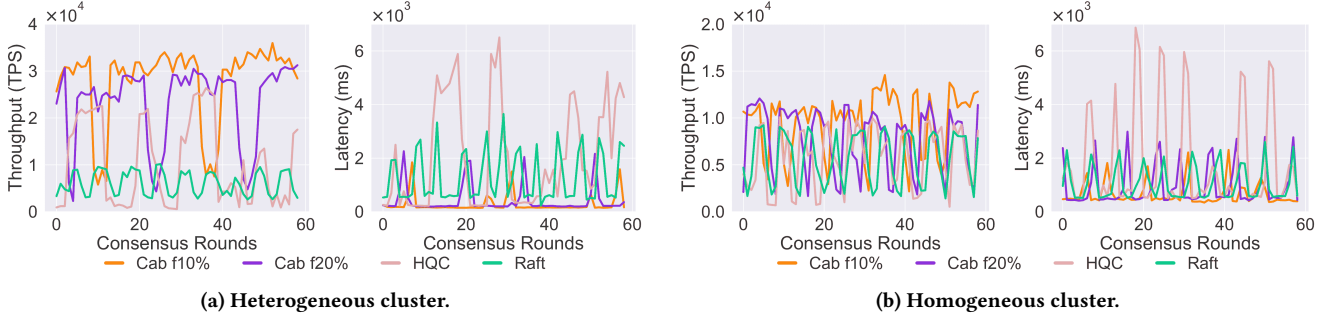(a) Heterogeneous cluster.

(b) Homogeneous cluster.

Figure 16: Performance of YCSB+MongoDB under n = 11, d = D4, b = 5k under Workload A; HQC divides cluster into 3-3-5.

under **D2**, the performance of `Cab f10%` remained comparable to that under **D1** = 100ms, whereas Raft's performance was degraded to the level observed under **D1** = 500ms. These results indicate Cabinet's performance gains in diverse network conditions.

Figure 13 shows the performance of all YCSB workloads under a skew delay of **D2**, further highlighting Cabinet's superiority in heterogeneity. For example, Cabinet achieved approximately 6× higher throughput (e.g., 18,899 TPS of `Cab f10%` vs. 3045 TPS of Raft in Workload A) and 5.5× lower latency (e.g., 264ms of `Cab f10%` vs. 1477ms of Raft in Workload C).

In addition to its high performance under static delays, Cabinet can effectively adapt to changing network delays, ensuring the system remains at optimal performance. Figure 15 shows the real-time performance of Cabinet and Raft under varying network delays of **D3**. Cabinet's throughput drops and latency increases (around Round 20 and 50) when strong nodes with high weights experience higher delays. Nevertheless, Cabinet promptly responds by reassigning high weights to currently faster nodes and regains high performance.

Bursting delays. In addition to dynamically changing delays, we evaluated the system's performance under bursting delay conditions (**D4**), where delay spikes of 1000±100ms occur intermittently. A 5-second period of bursting delays follows a 10-second no-delay period (2:1 ratio). Figure 16 presents the real-time throughput and latency performance in both heterogeneous and homogeneous clusters. For comparison, we included hierarchical quorum consensus (HQC) as a baseline. HQC partitions the system into three groups with sizes $|G_1|$ = 3, $|G_2|$ = 3, and $|G_3|$ = 5. Consensus is first
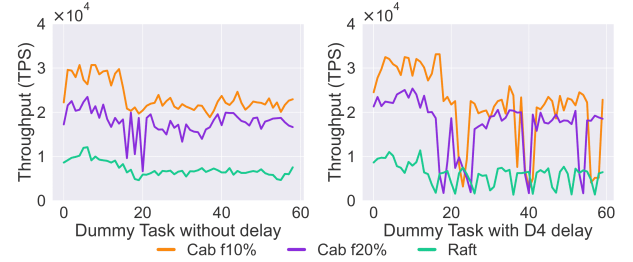

Figure 17: Real-time performance of YCSB under n = 11, b = 5k under Workload A; dummy tasks start at Round 20.

reached within each group, followed by global consensus among the group leaders.

Under spike delays, the performance of all evaluated consensus algorithms degrades, but heterogeneous clusters demonstrate greater robustness. Cabinet effectively mitigates the impact of spike delays by reassigning higher weights to more responsive nodes. As analyzed in §2, HQC incurs significantly higher latency due to its multi-round message-passing, which amplifies delays under heavy delay conditions. For example, in Figure 16a, in Round 18, HQC exhibits latency (6505ms) up to 4.3× that of `Cabinet f10%` (1502ms) and 3× that of Raft (2154ms) when hit by a delay spike.

Resource contention. We also evaluated scenarios where nodes concurrently handle other tasks, a common setup in modern deployments where physical machines run multiple containerized applications sharing CPU resources. Specifically, we introduced a CPU-heavy dummy process that repeatedly computes hash values

(a) Crash failures, starting at Round 20.

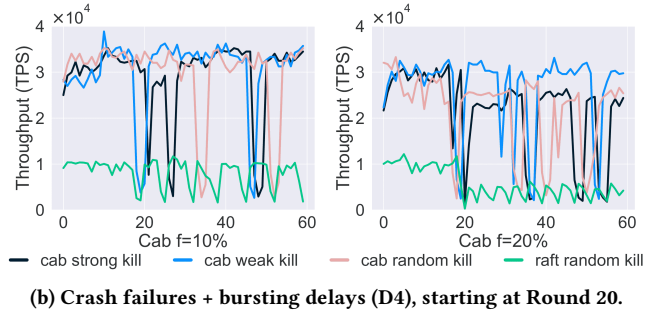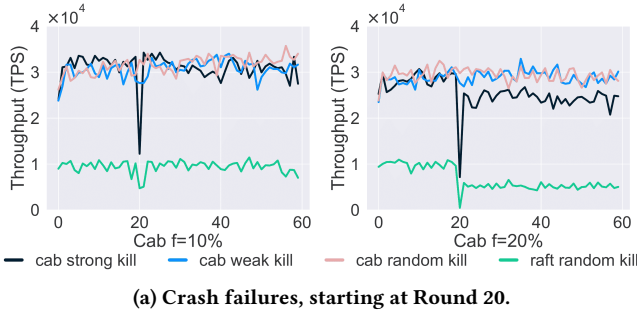(b) Crash failures + bursting delays (D4), starting at Round 20.

Figure 18: Performance of crash + spike (D4) under YCSB+MongoDB, where $n = 11$, $b = 5k$ under Workload A.

and utilizes multiple threads equal to the number of the node's vCPUs, creating significant CPU loads.

Figure 17 reports the performance of YCSB+MongoDB in an $n = 11$ cluster, where the dummy task starts around round 20. The left figure shows results when only the dummy task is executed, while the right figure shows results when the dummy task is executed alongside bursting delays, with the first delay spike coinciding with the task's start. Without bursting delays, all three algorithms experienced a performance dip. When bursting delays were introduced, performance exhibited additional fluctuations. Notably, Cabinet quickly adjusted node weights and recovered from impacted rounds. The presence of the dummy task combined with spike delays did not change the performance rankings among the evaluated algorithms, highlighting Cabinet's robustness in handling compounded disruptions.

Although throughput inevitably drops when strong nodes experience high network delays, Cabinet can quickly adapt and return to high throughput by reassigning weights in the next round. This adaptability is a crucial feature of Cabinet, allowing it to deliver consistently high performance in real-world deployments where networks are unpredictable.

5.4 Performance under Failures

We evaluated the performance of Cabinet and Raft in the presence of failures and network delays in heterogeneous clusters of $n = 11$ nodes. Figure 18 shows the real-time throughput of Cabinet with f10%, f20%, and Raft under YCSB+MongoDB (Workload A) under three crash scenarios: strong kills, weak kills, and random kills. These scenarios allow us to assess how weighted consensus performs under targeted failures, where strong and weak nodes are specifically impacted, as well as random failures. Note that since Raft does not apply weights, all crashes were random kills.

Strong kills crash x nodes with the top x highest weight.
Weak kills crash x nodes with the bottom x lowest weight.
Random kills crash x nodes randomly regardless of weights.

Figure 18a reports the evaluation results of crash failures under different strategies. All crashes occurred at Round 20. For example, in f=20% under strong kills, we crashed the nodes with the top 2 weights at Round 20. Before Round 20, all Cabinet operated stably at high throughput, with strong nodes assigned higher weights. At Round 20, strong kills crashed nodes with high weights, resulting in

plummeted throughput in the next round (f=10%). However, higher weights were promptly reassigned to non-faulty nodes, leading to a throughput recovery in the next round. The recovered throughput became lower than the throughput before the crash, as t strong nodes were lost. The recovered throughput dropped more substantially (f=10% vs. f=20%) under a higher t as more strong nodes were killed. Despite the drop in throughput after recovery, Cabinet still outperformed Raft.

In contrast, under weak kills, where nodes with low weights were crashed, performance was unaffected. The throughput remained unchanged compared to normal operation before Round 20, as the leader collected replies from the same set of stronger nodes. Additionally, the throughput under random kills in Cabinet stayed between the strong kills and weak kills scenarios since it involved killing both strong and weak nodes.

We further evaluated Cabinet and Raft under a combined failure scenario: crash failures and delay spikes **D4**). At Round 20, crashes occurred as per the above strategies, accompanied by a sudden burst of delays across all nodes. Figure 18b highlights Cabinet's resilience in this scenario. Despite the compounded failures, Cabinet rapidly reassigned weights to responsive nodes and recovered throughput within a few rounds. In contrast, Raft exhibited slower recovery and lower post-failure throughput.

To conclude, Cabinet demonstrated strong robustness and adaptability under diverse network delays and failure conditions. These results highlight that Cabinet's efficiency makes it particularly suitable for large-scale, heterogeneous systems.

6 CONCLUSIONS

Consensus algorithms relying on majority quorums encounter inefficiencies in large-scale, especially heterogeneous, systems, as strong nodes are forced to wait for weaker nodes to reach consensus. This paper proposes Cabinet, which enables dynamic weighted consensus. Consensus is no longer based on a majority of physical nodes but on a majority of weights, allowing stronger nodes to have a more significant influence in the agreement process. Furthermore, Cabinet dynamically reassigns high weights to more responsive nodes, ensuring the system stays at optimal performance. In our evaluation, Cabinet consistently outperforms Raft under various scenarios, including increasing system scales, static, skew, and dynamic network delays, and different proportions of crash failures.

REFERENCES

[1] Divyakant Agrawal and Amr El Abbadi. 1992. The generalized tree quorum protocol: An efficient approach for managing replicated data. *ACM Transactions on Database Systems (TODS)* 17, 4 (1992), 689–717.

[2] Yair Amir and Avishai Wool. 1998. Optimal availability quorum systems: Theory and practice. *Inform. Process. Lett.* 65, 5 (1998), 223–228.

[3] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2020. Modern large-scale data management systems after 40 years of consensus. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1794–1797.

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 30.

[5] Dynamo at 15: What Worked and What's Next in Majority Quorum Databases. [n.d.]. Scott Andreas. http://hpts.ws/papers/2022/Dynamo_at_15-HPTS2022. pdf.

[6] Catalonia-Spain Barcelona. 2008. Mencius: building efficient replicated state machines for WANs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*.

[7] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2019. *The datacenter as a computer: Designing warehouse-scale machines*. Springer Nature.

[8] William J Bolosky, Dexter Bradshaw, Randolph B Haagens, Norbert P Kusters, and Peng Li. 2011. Paxos replicated state machines as the basis of a {high-performance} data store. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*.

[9] Eric Brewer. 2012. CAP twelve years later: How the" rules" have changed. *Computer* 45, 2 (2012), 23–29.

[10] Apache Cassandra. 2022. Apache Cassandra 4.1: Rock Solid, Cloud-Native, Strongly Consistent and Highly Scalable. https://cassandra.apache.org/_/blog/Cassandra-4.1-is-here.html.

[11] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OsDI*, Vol. 99. 173–186.

[12] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2021. Pigpaxos: Devouring the communication bottlenecks in distributed consensus. In *Proceedings of the 2021 International Conference on Management of Data*. 235–247.

[13] Compute Canada Cloud. 2021. Arbutus cloud. https://docs.computecanada.ca/wiki/Cloud_resources

[14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[15] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 261–264. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett

[16] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[17] Transaction Processing Performance Council. 1992. TPC Benchmark C: an on-line transaction processing (OLTP) benchmark. https://www.tpc.org/tpcc/.

[18] Diem. 2020. The Diem blockchain. https://developers.diem.com.

[19] Docker. 2015. Raft consensus in swarm mode. https://docs.docker.com/engine/swarm/raft/.

[20] etcd. 2021. A distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io/.

[21] Hector Garcia-Molina and Daniel Barbara. 1985. How to assign votes in a distributed system. *Journal of the ACM (JACM)* 32, 4 (1985), 841–860.

[22] David K Gifford. 1979. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*. 150–162.

[23] Maurice Herlihy. 1986. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems (TOCS)* 4, 1 (1986), 32–53.

[24] Heidi Howard, Aleksey Charapko, and Richard Mortier. 2021. Fast flexible paxos: Relaxing quorum intersection for fast paxos. In *Proceedings of the 22nd International Conference on Distributed Computing and Networking*. 186–190.

[25] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2017. Flexible Paxos: Quorum Intersection Revisited. In *20th International Conference on Principles of Distributed Systems*.

[26] Heidi Howard and Richard Mortier. 2022. Relaxed Paxos: Quorum Intersection Revisited (Again). In *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*. 16–23.

[27] Sushil Jajodia and David Mutchler. 1987. Dynamic voting. *ACM SIGMOD Record* 16, 3 (1987), 227–238.

[28] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Gustavo Alonso, and Bettina Kemme. 2003. Are quorums an alternative for data replication? *ACM Transactions on Database Systems (TODS)* 28, 3 (2003), 257–294.

[29] Akhil Kumar. 1991. Algorithm for Managing Replicated Data. *IEEE transactions on Computers* 40, 9 (1991).

[30] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.

[31] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2009. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*. 312–313.

[32] Leslie Lamport and Mike Massa. 2004. Cheap paxos. In *International Conference on Dependable Systems and Networks, 2004*. IEEE, 307–314.

[33] LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE. 1982. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems* 4, 3 (1982), 382–401.

[34] Lamport Leslie. 1998. The part-time parliament. *ACM Trans. on Computer Systems* 16 (1998), 133–169.

[35] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. 2016. Just say {NO} to paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 467–483.

[36] Shen Li. 2023. PYTORCH DISTRIBUTED. https://pytorch.org/tutorials/beginner/dist_overview.html.

[37] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. 2010. Ring Paxos: A high-throughput atomic broadcast protocol. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE, 527–536.

[38] Jason Mars and Lingjia Tang. 2013. Whare-map: heterogeneity in" homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 619–630.

[39] MongoDB. [n.d.]. https://www.mongodb.com/.

[40] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2012. Egalitarian paxos. In *ACM Symposium on Operating Systems Principles*.

[41] Netem. [n.d.]. https://man7.org/linux/man-pages/man8/tc-netem.8.html.

[42] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 305–319.

[43] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. 2015. The RAMCloud storage system. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 1–55.

[44] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. 2015. Designing distributed systems using approximate synchrony in data center networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 43–57.

[45] PostgreSQL. [n.d.]. https://www.postgresql.org/.

[46] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, Julien Maffre, Thomas Moscibroda, Kartik Nayak, Olya Ohrimenko, Felix Schuster, Roy Schwartz, Alex Shamis, Olga Vrousgou, and Christoph M. Wintersteiger. 2019. *CCF: A Framework for Building Confidential Verifiable Replicated Services*. Technical Report MSR-TR-2019-16. Microsoft.

[47] Ramendra K Sahoo, Mark S Squillante, Anand Sivasubramaniam, and Yanyong Zhang. 2004. Failure data analysis of a large-scale heterogeneous server environment. In *International Conference on Dependable Systems and Networks, 2004*. IEEE, 772–781.

[48] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319.

[49] Bianca Schroeder and Garth A Gibson. 2009. A large-scale study of failures in high-performance computing systems. *IEEE transactions on Dependable and Secure Computing* 7, 4 (2009), 337–350.

[50] Alex Shamis, Peter Pietzuch, Burcu Canakci, Miguel Castro, Cédric Fournet, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, Antoine Delignat-Lavaud, Matthew Kerner, et al. 2022. IA-CCF: Individual Accountability for Permissioned Ledgers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 467–491.

[51] Rong Shi and Yang Wang. 2016. Cheap and available state machine replication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 265–279.

[52] Benedict Elliott Smith. 2022. CEP-14: Paxos Improvements. https://cwiki.apache.org/confluence/display/CASSANDRA/CEP-14%3A+Paxos+Improvements.

[53] NetFlix TechBlog. 2011. Benchmarking Cassandra Scalability on AWS — Over a million writes per second. https://netflixtechblog.com/benchmarking-cassandra-scalability-on-aws-over-a-million-writes-per-second-39f45f066c9e.

[54] NetFlix TechBlog. 2014. Revisiting 1 Million Writes per second. https://netflixtechblog.com/revisiting-1-million-writes-per-second-c191a84864cc.

[55] TensorFlow. 2023. Distributed training with TensorFlow. https://www.tensorflow.org/guide/distributed_training.

[56] Devesh Tiwari, Saurabh Gupta, James Rogers, Don Maxwell, Paolo Rech, Sudharshan Vazhkudai, Daniel Oliveira, Dave Londo, Nathan DeBardeleben, Philippe Navaux, et al. 2015. Understanding GPU errors on large-scale HPC systems and the implications for system design and operation. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 331–342.

[57] Ash van der Spuy. 2020. Google Cloud Spanner Technical Overview. https://medium.com/google-cloud/google-cloud-spanner-technical-overview-e3e37d81ea60.

[58] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S Rellermeyer. 2020. A survey on distributed machine learning. *Acm computing surveys (csur)* 53, 2 (2020), 1–33.

[59] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 347–356.

[60] Gengrui Zhang and Hans-Arno Jacobsen. 2021. Prosecutor: An efficient BFT consensus algorithm with behavior-aware penalization against Byzantine attacks. In *Proceedings of the 22nd International Middleware Conference*. 52–63.

[61] Gengrui Zhang and Hans-Arno Jacobsen. 2022. Escape to precaution against leader failures. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 625–635.

[62] Gengrui Zhang, Fei Pan, Yunhao Mao, Sofia Tijanic, Michael Dang'Ana, Shashank Motepalli, Shiquan Zhang, and Hans-Arno Jacobsen. 2024. Reaching consensus in the byzantine empire: A comprehensive review of bft consensus algorithms. *Comput. Surveys* 56, 5 (2024), 1–41.

[63] Gengrui Zhang, Fei Pan, Sofia Tijanic, and Hans-Arno Jacobsen. 2024. Prestigebft: Revolutionizing view changes in bft consensus algorithms with reputation mechanisms. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 1930–1943.

[64] Gengrui Zhang and Chengzhong Xu. 2019. An efficient consensus protocol for real-time permissioned blockchains under non-byzantine conditions. In *Green, Pervasive, and Cloud Computing: 13th International Conference, GPC 2018, Hangzhou, China, May 11-13, 2018, Revised Selected Papers 13*. Springer, 298–311.

[65] Gengrui Zhang, Shiquan Zhang, Michail Bachras, Yuqiu Zhang, and Hans-Arno Jacobsen. 2025. Cabinet: Dynamically Weighted Consensus Made Fast. https://arxiv.org/abs/2503.08914.

[66] Qian Zhang, Jingyao Li, Hongyao Zhao, Quanqing Xu, Wei Lu, Jinliang Xiao, Fusheng Han, Chuanhui Yang, and Xiaoyong Du. 2023. Efficient distributed transaction processing in heterogeneous networks. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1372–1385.

[67] Tony Zhang, Blake Eggleston, and Scott Andreas. [n.d.]. CEP-15: Fast General Purpose Transactions. ([n. d.]).

[68] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. 2018. Sdpaxos: Building efficient semi-decentralized geo-replicated state machines. In *Proceedings of the ACM Symposium on Cloud Computing*. 68–81.

[69] Zookeeper. [n.d.]. Zookeeper Hierarchical Quorums. https://zookeeper.apache.org/doc/r3.4.2/zookeeperHierarchicalQuorums.html.