



Orca: Flexible Quorums Meet Dynamic Quorums

Kevin Dharmawan
Stony Brook University
kdharmawan@cs.stonybrook.edu

Prajwal Annigeri
Stony Brook University
pannigeri@cs.stonybrook.edu

Mohammad Javad Amiri
Stony Brook University
amiri@cs.stonybrook.edu

ABSTRACT

Large-scale data management systems rely on fault-tolerant protocols to ensure availability and consistency in the presence of node failures. However, standard protocols require majority quorums in every phase, making them impractical in environments with frequent failures, such as IoT networks. This paper presents Orca, a crash-fault-tolerant consensus protocol that integrates Raft-style log replication with Flexible Paxos quorum construction to maintain small replication quorums. Orca exploits the observation that node failures are often independent and temporally separated: by assuming that at most k nodes fail within a window too short for the system to detect the first failure, Orca fixes the replication quorum at $k+1$ nodes. As failures are detected, faulty nodes are temporarily excluded from the voting set and the election quorum is reduced accordingly, allowing the system to continue making progress as failures accumulate. Our evaluation across diverse workloads and fault scenarios demonstrates that Orca tolerates more failures than Raft and FlexiRaft while incurring negligible overhead.

PVLDB Reference Format:

Kevin Dharmawan, Prajwal Annigeri, and Mohammad Javad Amiri. Orca: Flexible Quorums Meet Dynamic Quorums. PVLDB, 19(11): 3565 - 3578, 2026.
doi:10.14778/3836663.3836709

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/SeeMoReLab/orca/tree/artifact>.

1 INTRODUCTION

Large-scale data management systems [4, 6, 8, 14, 15, 26, 40] rely on consensus protocols, such as Paxos [30] and Raft [45], to enable a collection of independent servers to agree on a consistent ordering of transactions. These protocols typically consist of two phases: leader election, where a node is elected to coordinate incoming transactions, and replication, where the leader assigns an order to transactions and replicates them across other nodes. To preserve consistency, both phases require majority quorums, ensuring that any two quorums intersect on at least one common node.

Node failures are common in large-scale deployments, and particularly prevalent in environments such as IoT networks, where hardware constraints, unstable connectivity, and limited maintenance make failures frequent. In such settings, requiring a majority of nodes to participate in every phase can be infeasible, as even a

modest number of failures may render the system unable to make progress.

One approach to handling failures is to rely on reconfiguration protocols: treating a failed node as removed from the system, thereby reducing the quorum size and maintaining availability. However, reconfiguration protocols introduce two significant drawbacks. First, they require a costly transition phase in which typically a majority of both the old and new configurations must participate [44, 45, 58]. Second, some protocols [34, 57] rely on an external service to manage configuration changes, introducing additional resource overhead and a potential single point of failure.

An alternative approach, proposed by Flexible Paxos [24], decouples the quorum requirements of the two phases: using a smaller replication quorum to improve performance while compensating with a larger election quorum to preserve safety. This asymmetry, while reducing replication cost, introduces an intuitive liveness challenge: if the current leader fails and the number of non-faulty nodes is smaller than the election quorum size, which is typically a larger number of nodes compared to the replication quorum, the system cannot make any progress until some faulty nodes recover.

One possible solution is to integrate reconfiguration protocols with flexible quorums: failed nodes are excluded from the system via reconfiguration, reducing the election quorum, while replication proceeds over the smaller replication quorum prescribed by Flexible Paxos. However, this raises a critical question: what is the appropriate replication quorum size? If the replication quorum is too small, the corresponding election quorum, which must be larger to preserve safety, may exceed the number of available nodes when the leader fails, stalling the system until faulty nodes recover. On the other hand, if the replication quorum is set too large, approaching a majority, the performance benefits of flexible quorums are diminished.

Fault-tolerant systems commonly assume that replica failures are independent, that is, a single fault does not trigger the failure of multiple replicas. This independence is typically enforced by either diversifying replica implementations (e.g., n -version programming) [3, 22] or distributing replicas across geographically separated locations (e.g., distinct datacenters) [5, 19, 52, 56]. In practice, this assumption holds well: failures tend to arise from localized causes, such as individual hardware faults, network partitions, or power events confined to a single node or rack, making simultaneous failures across independent replicas unlikely.

This independence implies that failures rarely occur simultaneously; rather, they tend to be separated by a temporal gap. We exploit this gap by assuming that at most k nodes fail within a time window short enough that the system cannot detect the first failure before the k -th occurs. Under this assumption, a replication quorum of size $k+1$ suffices. Furthermore, as failures are detected, failed nodes are temporarily excluded from the voting set, and the election quorum is reduced accordingly, allowing the system to

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836709

continue making progress despite an accumulation of failures over time. More precisely, once a node is marked as faulty by other nodes through consensus, it loses its voting power until recovery is confirmed. Hence, the size of the leader election quorum Q_e is calculated as $n - k - |F_d|$, where F_d represents the set of detected faulty nodes. Intuitively, $n - |F_d|$ represents the operational (voting) nodes, among which $k + 1$ nodes are required for replication quorum; thus, the election quorum consists of $n - k - |F_d|$ nodes.

This paper presents Orca, a crash-fault-tolerant consensus protocol that integrates Raft-style log replication with Flexible Paxos quorum construction and a dynamic failure detection mechanism. By exploiting the temporal separation between independent failures, Orca maintains small replication quorums to obtain the performance benefits of Flexible Paxos, while dynamically shrinking the election quorum as faults are detected, enabling the system to tolerate failures under conditions where standard protocols would stall. Our contributions are as follows:

- We introduce Orca, a crash-fault-tolerant consensus protocol that combines flexible quorums, failure detection, and dynamic quorum adjustment to sustain progress under failure patterns that block existing protocols.
- We formalize a temporal failure assumption, bounding the number of nodes that may fail within a short time window, and show how it can be leveraged to maintain liveness while keeping replication quorums small.
- We implement a prototype of Orca and conduct extensive experiments across diverse workloads and fault scenarios. Our results show that Orca outperforms Raft and FlexiRaft by tolerating a significantly higher number of failures with minimal overhead.

2 BACKGROUND

This section introduces the main building blocks of Orca. We first briefly introduce Paxos [30], Flexible Paxos [24], and Raft [45], as practical crash fault-tolerant consensus protocols that have inspired the design of Orca, and then discuss prior work on failure detection and dynamic reconfiguration mechanisms.

2.1 Paxos and Flexible Paxos

Reaching consensus in the presence of failures has been addressed by a large number of fault-tolerant protocols [1, 7, 9, 11, 16, 24, 25, 30–33, 35–38, 42, 43, 45, 46, 55], among which Paxos [30] is the most known. Paxos ensures safety in an asynchronous system with $2f+1$ nodes, tolerating up to f crash failures. Safety is guaranteed through quorum intersection: once a value is chosen, all future leaders must observe it. Paxos proceeds in two phases, leader election and replication, and is designed to decide a single value. Multi-Paxos enhances this by designating a stable leader, which simplifies processing by omitting repeated elections for every transaction: once elected (as a result of executing the first phase), the leader continues until a failure occurs.

Paxos mandates a majority quorum for both leader election and replication to ensure that all quorums intersect. Flexible Paxos [24] suggests that only the election quorum and replication quorums need to intersect (not all replication quorums), allowing for different quorum sizes in each phase. Since leader elections occur

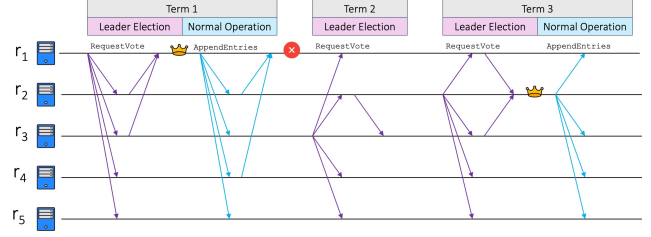


Figure 1: Raft

less frequently than replication, it is advantageous to use a smaller quorum for replication and a larger one for leader election.

2.2 Raft

Raft [45] divides the time into terms, where each term has at most one leader. Each term starts with a leader election phase. If the leader election phase succeeds and a leader is elected, it is followed by a normal operation phase in which log replication occurs. At any given time, a node can be in one of three states: leader, candidate, or follower. Raft requires majority quorums for both leader election and log replication. Once a log entry is committed, Raft's safety property ensures that no other log is committed at the same index, and this log can be safely applied to the state machine.

Figure 1 shows a Raft deployment with 5 nodes. In term 1, node r_1 initiates leader election, is elected as leader, and processes an AppendEntries during the normal operation. The leader (r_1) then fails; thus, term 1 ends. Term 2 begins with r_3 becoming a candidate and initiating an election, but it fails to become the leader, and term 2 ends. In term 3, r_2 transitions to the candidate state, is elected as leader, and the system resumes normal operation.

2.3 Failure Detection

Failure detection is used for managing node failures during consensus protocols. It involves nodes monitoring one another and suspecting failure if responses are not received within a specified timeout. In failure detection mechanisms, completeness ensures that all non-faulty nodes eventually detect the failure of any node, while efficiency indicates rapid and accurate failure detection, minimizing false positives and negatives [23].

Failure detectors were introduced by Chandra and Toueg [10] as a mechanism to solve consensus in asynchronous message-passing systems with crash failures. They differentiate between slow and dead nodes by employing a failure detector module attached to each node, which continuously estimates node failures. However, it has been proven that no deterministic algorithm can ensure both completeness and accuracy in such asynchronous, unreliable networks. Consequently, many distributed applications have adopted failure detector algorithms that guarantee deterministic completeness while achieving probabilistic efficiency [2, 12, 20, 27, 41, 54].

A prevalent failure detection method involves periodic "heartbeat" or "I'm alive" messages sent from nodes to either other nodes or a monitoring system. If a node fails to receive a heartbeat within a predefined timeframe, it suspects failure in the sender [2, 12, 20, 27, 41, 54]. This approach is favored for its assurance of completeness.

Prior studies on failure detection typically focus on the crash-stop model, in which a failed node is assumed not to recover in the near future. Thus, the failure detector’s objective is to permanently suspect the failed nodes. Our work differs from all this prior work in that our failure detection focuses on the crash-recovery model, in which a failed node may recover in the near future.

A complementary line of work avoids failure detection and timeouts altogether. Randomized asynchronous consensus protocols such as QuePaxa [53] replace timeout-triggered view changes with a randomized asynchronous core and short hedging delays, preserving liveness under adverse or adversarial network conditions without relying on a failure detector. Such protocols target robustness to network asynchrony in a fixed, crash-stop membership. Orca instead adds failure and recovery detection to a leader-based, Flexible Paxos-style protocol operating in the crash-recovery model, where failed nodes may recover, and shrinks the election quorum as faults accumulate to sustain progress. The two approaches thus address orthogonal aspects of robustness: tolerance to network asynchrony versus availability under recurring crash-recovery failures.

2.4 Dynamic Reconfiguration

Dynamic reconfiguration refers to the ability to modify the membership of a distributed system by adding or removing nodes while it is operational, enabling it to adapt to changing circumstances without undergoing a complete shutdown. This process requires careful coordination to ensure both safety and efficiency: no data must be lost or reordered during membership updates, all while maintaining system availability and stable performance. Schiper [48] studies the dynamic reconfiguration challenge, framing membership updates as view changes. He proposes a framework for nodes to join and leave the system via reliable communication channels, which guarantee that messages are neither duplicated nor lost. Reliable communication is modeled through atomic broadcast, wherein all correct nodes in the system receive the same set of messages in the same order. Atomic broadcast is equivalent to consensus for a system with failures [28].

Some recent consensus protocols, such as Raft [45], manage dynamic reconfiguration through a joint consensus approach that integrates with the primary replication log. During reconfiguration, nodes from both the old and new configurations must reach an agreement and commit a value collectively via joint consensus. An alternative method involves separating reconfiguration from the main replication log [50], as implemented in MongoDB [39]. This approach enforces an overlap of quorums between the old and new configurations to preserve safety while allowing reconfiguration to function independently of the main replication log. This decoupling reduces latency while maintaining safety. Both the joint consensus mechanism in Raft and the logless reconfiguration strategy in MongoDB are designed to facilitate the introduction of new nodes or the deactivation of existing ones, often for purposes such as software updates. Add/RemoveServer (AR) RPC [44] avoids joint configurations by performing reconfiguration through a sequence of single-node additions or removals, while ReCraft [58] generalizes this approach by supporting multi-node membership changes through explicit quorum resizing. Despite their differences,

these mechanisms fundamentally reconfigure the physical membership of the consensus group. Our research, however, is focused on enhancing the system’s resilience to accommodate a greater number of failures without necessitating the addition or removal of nodes. Instead, we aim to reconfigure the system’s quorum size by marking nodes as either voting or non-voting.

Beyond this difference in goals, our approach also differs in how reconfiguration is carried out. Similar to ReCraft, our protocol supports adding or removing multiple nodes within a single reconfiguration entry, avoiding the multi-step transitions required by joint consensus. At the same time, like joint consensus, our design explicitly distinguishes between voting and non-voting replicas, which is not supported in ReCraft. Unlike both joint consensus and ReCraft, however, our approach does not require expanding quorum sizes during reconfiguration. Instead, our design fixes the replication quorum size $|Q_r|$ and adjusts the election quorum $|Q_e|$ to preserve the required overlap, without expanding quorums.

Matchmaker Paxos [57] and Vertical Paxos [34] decouple configuration management from the main replication protocol by introducing an external service that determines the active configuration. Matchmaker Paxos employs a replicated matchmaker service to track configuration history and act as an oracle for the current configuration, reducing the impact of reconfiguration on steady-state replication. Vertical Paxos allows acceptor sets and quorums to change across ballots under the control of a master, enabling progress across configurations even when individual configurations stall. In its primary-backup formulation, progress may require only a single replication node in the active configuration; however, this relies on an external master that must itself be replicated and fault-tolerant to ensure safety and liveness. Orca instead requires a replication quorum of size $|Q_r|$ to make progress, but avoids external configuration services by dynamically managing voting participation entirely within the consensus group.

Cheap Paxos [35] addresses a different problem by targeting deployments with heterogeneous replica capabilities rather than resilience under failures. It introduces asymmetric roles within a Paxos deployment, designating a subset of replicas as active voters whose responses count toward quorums, while the remaining replicas act as auxiliary nodes that passively replicate state but do not participate in voting. Auxiliary replicas may be promoted to active voters upon failures, but voting roles are otherwise fixed by configuration rather than adjusted dynamically. In contrast, Orca adapts voting participation in response to observed failures within a fixed replica set to improve availability.

DPaxos [42] and WPaxos [1] optimize Paxos for geo-distributed environments by reshaping quorums and leadership to reduce wide-area coordination costs. DPaxos decouples leader election from replication by allowing leaders to use small, locality-aware replication quorums while preserving safety by expanding leader-election quorums to intersect with prior replication quorums. WPaxos takes a different approach by distributing leadership across multiple concurrent leaders and partitioning coordination across objects, enabling locality and throughput through parallelism. In both systems, quorum selection and leadership placement are primarily performance optimizations guided by deployment topology. Orca addresses a complementary problem by improving availability under failures through dynamic reconfiguration of voting participation,

rather than optimizing locality or throughput. Moreover, unlike DPaxos and WPaxos, which assume fixed bounds on simultaneous zone/node failures, Orca tolerates additional failures over time as long as no more than k failures occur within a single failure window.

3 SYSTEM MODEL

We consider a network consisting of n nodes, where nodes follow the crash failure model. In a crash failure model, nodes operate at arbitrary speed, may fail by stopping, and may restart; however, they may not collude, lie, or otherwise attempt to subvert the protocol. Network partitions may also occur.

Consensus protocols use the State Machine Replication (SMR) algorithm [29, 49] where the system provides a replicated service whose state is mirrored across different deterministic replicas. At a high level, the goal of an SMR protocol is to assign each client request an order in the global service history and execute it in that order [51]. In an SMR protocol, all replicas execute the same requests in the same order (*safety*) and all correct requests are eventually executed (*liveness*). In an asynchronous system, where replicas can fail, no consensus solutions guarantee both safety and liveness (FLP result) [21]. Orca, similar to most other consensus protocols, assumes the partial synchrony model [17, 18] to circumvent the FLP impossibility. In the partial synchrony model, there exists an unknown global stabilization time (GST), after which all messages between correct replicas are received within some known bound Δ .

Partially synchronous crash fault-tolerant (CFT) protocols, e.g., Paxos [30], guarantee safety using $2f + 1$ nodes to overcome f crash failures. While a CFT protocol can tolerate up to f failures, in practice, nodes do not fail at exactly the same time for two main reasons. First, fault-tolerant systems are architected to ensure that failures among nodes are independent; a failure impacting one node should not trigger failures in other nodes. One approach to achieving this independence is through the use of diverse replica implementations, such as n -version programming [3, 22]. Second, nodes are often distributed across multiple data centers. This geographical distribution not only contributes to the independence of node failures but also enhances system availability during significant outages that may impact entire data centers, such as those resulting from natural disasters.

Using this insight, we assume that the failure of at most k nodes (out of n nodes, $k \leq f$) is dependent, meaning they fail simultaneously or within a short period such that the system cannot detect the failure of the first node before the failure of the k -th node. Importantly, k bounds the number of failures that may occur within such a failure window, rather than the total number of failures the system can tolerate over time. Orca remains safe regardless of the choice of k , as safety is preserved by replication and election quorums' intersection; however, liveness depends on the assumption that at most k nodes fail within any failure-detection window. Intuitively, k captures the maximum number of nodes that may fail simultaneously before earlier failures can be detected and excluded. In practice, k should be chosen conservatively based on the size of failure domains (e.g., rack or data center). A data center scale outage is one example of a correlated failure event that may motivate a choice of k . For instance, consider a network with 9

nodes spread across 5 data centers, where four data centers, each hosting 2 nodes, and the fifth data center hosts 1 node. In this case, if a data center-scale outage happens, at most 2 nodes might fail simultaneously, hence $k = 2$.

In a consensus protocol, a quorum refers to the minimum number of nodes that must agree or participate in a decision or operation for it to be considered valid. Quorums are constructed for deciding on either a value (replication quorum) or the next leader (election quorum). In Paxos [30], Raft [45], and most other CFT protocols, the quorum size is $f + 1$ out of $2f + 1$ (majority). This is to ensure that any two quorums intersect, guaranteeing consistency and preventing conflicting outcomes. Flexible Paxos [24], on the other hand, relaxes this requirement, allowing for potentially disjoint replication quorums within a view (with the same leader), while still ensuring an intersection between all replication quorums within a view and the next leader election quorum. These protocols assume that up to f nodes may fail in the worst case without distinguishing between failures that occur simultaneously and those that occur over time. In contrast, Orca assumes that at most k ($\leq f$) nodes fail within a failure-detection window, explicitly distinguishing between concurrent failures and failures that occur over time.

Inspired by Flexible Paxos, we define flexible replication and election quorums. Given that there are at most k dependent failures, the replication quorum Q_r is sized $k + 1$ to ensure that at least one non-faulty node from each replication quorum will participate in the next election quorum. The size of the leader election quorum Q_e then becomes $n - k - |F_d|$, where F_d is the set of faulty nodes whose failures have already been detected (i.e., excluded from the voting nodes). When a node is detected as faulty, it is no longer part of any quorum until it is detected as recovered, i.e., $Q_r \cap F_d = \emptyset$ and $Q_e \cap F_d = \emptyset$. Intuitively, $n - |F_d|$ are the alive (voting) nodes; out of them, $k + 1$ nodes are required for a replication quorum; hence, the election quorum includes $n - k - |F_d|$ nodes.

We define state function $S : r \mapsto \{A, LF, F, LA\}$ to capture the state of each replica at each point:

- $S(r) = A$ where node r is alive (non-faulty),
- $S(r) = LF$ where node r is locally failed: it appears to be failed in the local view of some nodes, but consensus has not been achieved yet on r 's failure,
- $S(r) = F$ where node r is failed and consensus is achieved on r 's failure, and
- $S(r) = LA$ where node r is locally alive (recovered): it appears to be alive in the local view of some nodes, but consensus has not been achieved yet on r 's recovery.

If $S(r) = F$ or $S(r) = LA$, then $r \in F_d$, thus r is a *non-voting* replica, i.e., its vote does not count towards replication and leader election quorums. However, if $S(r) = A$ or $S(r) = LF$, then $r \notin F_d$, hence, r is a *voting* replica, i.e., its votes count towards replication and leader election quorums.

4 PROCESSING TRANSACTIONS IN ORCA

This section introduces the Orca protocol, starting with normal-case operation where the system operates without any failure. Next, we discuss how the system dynamically reconfigures when a failure or recovery event happens. We briefly discuss the correctness arguments at the end of this section.

4.1 Normal-Case Operation

The normal-case operation of Orca is inspired by Raft [45]. Clients initiate their request by sending a request message containing a transaction to be executed by the replicated state machines to the node they think is the leader, or choose a node randomly if it is their first request. If the receiving node is not the leader, it will redirect the client to the leader. When the leader receives a request message, it will add the request to its own log as a new entry. The leader will run the replication protocol for the entry by sending an AppendEntries RPC to all follower nodes. The AppendEntries contains information about the leader (term, ID, and commit index) along with the previous log entry (index and term) and the log entries to be replicated, which can be empty.

Upon receiving an AppendEntries from the leader, each follower will check the leader's term. If it's the same or higher, the follower will reset its election timer, update its term to follow the leader's if needed, and check the log entry at the same index as the previous log entry. Then, the follower will send back an acknowledgment which contains a flag indicating whether or not the replication is a success – previous log entry match. If the replication is not a success because of a previous log entry mismatch, the leader will send the log entries with a lower index in the next AppendEntries. This mismatch may happen after a change of term, and the nodes have different local logs, and will only happen to uncommitted entries. When the number of nodes that have replicated (based on acknowledgment) a log entry is equal to the replication quorum size, the leader will commit the entry, update its commit index, and send reply messages consisting of the results back to the clients. Recall that the AppendEntries RPC contains the leader's commit index. So, the next time the leader broadcasts an AppendEntries RPC, the followers will see the new commit index and commit the log up to that index. If the client does not receive any reply messages after a predefined amount of time (client timer), it will resend the request message to another node in a round-robin manner.

When there is no new request to the system, the leader will keep the replication protocol going by replicating an empty AppendEntries RPC as a heartbeat message. This mechanism will enable the replicas to track the state of the current leader. If a replica has not received any valid AppendEntries for a predefined time (election timer), it will become a candidate and try to become the leader. We will discuss the leader election mechanism of Orca in Section 4.4.

4.2 Failure Detection

In the normal case operation, the leader initiates the replication protocol by sending an AppendEntries RPC to all followers, containing logs that haven't been replicated by the follower. Whether such a log exists or not, the leader sends an AppendEntries RPC to the follower, waits for an acknowledgment, and retries on the next heartbeat if a reply is not received. The leader keeps track of the number of retries for each follower and increments it for each attempt on sending AppendEntries. When the leader receives an acknowledgment from a follower, it will reset the retry count for that follower.

If a follower exceeds a predefined constant number of max retries τ and that follower is not already detected as failed – current state is not LF or F, the leader assumes that the follower has failed and adds a FailureDetection log entry to its log, changing the follower's

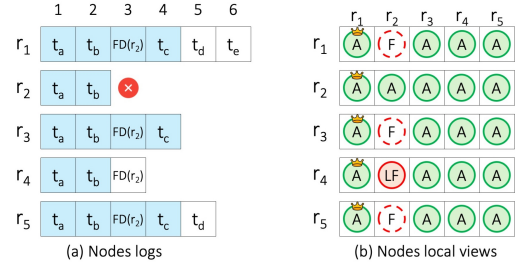


Figure 2: (a) Log entries for each replica; blue blocks are committed while white blocks are replicated but not committed. (b) Membership local view of each replica: the crown symbol represents the leader, green nodes are alive, red nodes are failed, voting nodes have a solid border, while non-voting nodes have a dashed border.

state to LF. FailureDetection is a log entry (similar to a client request) with no content other than the node detected as failed. When a FailureDetection log entry is executed, two things will happen: the node in the FailureDetection entry will become a non-voting node (state change from LF to F) and the election quorum size will be reduced (more details in Section 4.4).

A false failure detection may arise when the leader cannot communicate with a follower, even though other followers can, possibly due to network partitions. Different approaches to handling false failure detections produce different tradeoffs between communication overhead, election stability, and the ability to detect recovery under partitioned networks. More generally, handling false failure detections involves two orthogonal design choices: whether replicas independently verify a leader's failure judgment before accepting a FailureDetection entry (*active* detection), and whether replicas continue to process RequestVote RPCs from nodes currently marked as faulty (*passive* detection). These choices produce different tradeoffs between communication overhead, election stability, and recovery behavior under network partitions.

Orca adopts a non-active and passive approach, leveraging Raft's existing leader election mechanism to resolve false failure detections. Replicas accept FailureDetection entries without independently verifying connectivity, avoiding additional communication, but continue to process RequestVote RPCs from nodes marked as faulty. As a result, followers partitioned from the leader may still initiate elections by incrementing their term and sending RequestVote RPCs, allowing the cluster to converge on a new leader with more stable network connectivity.

Once a server adds the FailureDetection entry to its log, it adopts that configuration for all subsequent decisions. A server consistently relies on the most recent configuration in its log, irrespective of whether the entry has been committed. Note that different nodes might have different local views of the current membership.

Figure 2 presents (a) the log and (b) the nodes' local view of the current membership in a deployment with 5 nodes. The blue blocks in the log, e.g., t_1 and t_2 on all nodes, demonstrate committed entries while the white blocks, e.g., t_d on nodes r_1 and r_5 , are replicated but not committed yet. In the local membership view of replicas, the crown symbol represents the leader, green nodes are alive, red nodes are failed, voting nodes have a solid border, while non-voting nodes

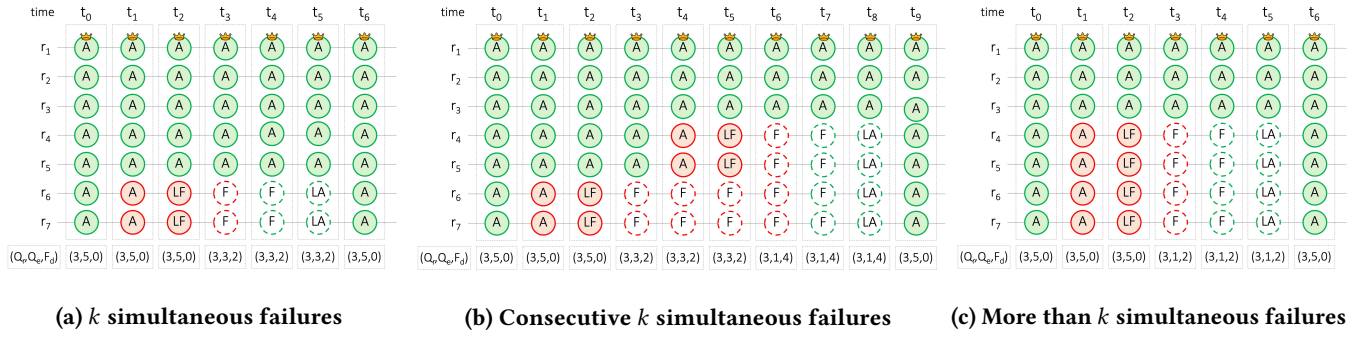


Figure 3: Different failure and recovery detection scenarios

have a dashed border. As can be seen, node r_2 is failed ($S(r_2) = F$) in the local view of nodes r_1 (the leader) and r_3 , but is still locally failed ($S(r_2) = LF$) in the local view of r_4 . We argue that such an inconsistency does not impact the safety of the protocol because only nodes that have the view of the latest membership can become a leader, as we discuss later in Section 4.5.

4.3 Recovery Detection

The recovery detection process is the counterpart of failure detection. Whenever the leader receives an acknowledgment from a follower, it checks whether the sender is a non-voting node whose recovery has not yet been detected ($S(r) = F$). If so, the leader appends an `RecoveryDetection` entry to its log, changing the node's state from `F` to `LA`. Once the `RecoveryDetection` entry is committed, the node becomes a voting replica (`A`) and the election quorum size is increased accordingly.

As an example, consider an Orca deployment with $n = 7$ and replication quorum size $|Q_r| = 3$ ($k = 2$). Figure 3(a) shows the normal failure and recovery process. Nodes r_6 and r_7 fail, are detected and excluded from voting through `FailureDetection` entries, then later both recover and are reintegrated through `RecoveryDetection` entries. During this process, the election quorum shrinks to $|Q_e| = 3$ after failure detection and expands again to $|Q_e| = 5$ after recovery.

Similar to the failure detection, once a server adds the `RecoveryDetection` entry to its log, it adopts that configuration for all subsequent decisions. A server consistently relies on the most recent configuration in its log.

In practice, nodes may fail one after another even if the failures are independent. Figure 3(b) illustrates consecutive failures. After the first pair of failed nodes (r_6 and r_7) is detected and excluded from voting, a second pair (r_4 and r_5) fails. Despite four failed nodes out of seven, Orca continues to make progress because the remaining three active nodes are sufficient to form the replication quorum. Once the failed nodes recover, they are reintegrated through the normal recovery process.

Orca assumes that at most k nodes fail within a failure-detection window, allowing it to use replication quorums of size $k + 1$. Nevertheless, Orca may continue to make progress even when more than k nodes fail simultaneously, provided that the leader remains alive and a replication quorum can still be formed. Figure 3(c) illustrates such a scenario. Four nodes fail simultaneously in a deployment

with $n = 7$ and $|Q_r| = 3$. Because the leader remains operational and can still form a replication quorum, it detects the failures, excludes the failed nodes from voting, and continues processing requests until the failed nodes recover.

4.4 Leader Election

A follower becomes a candidate when it does not receive messages from the leader before its election timer expires. The candidate increments its term, votes for itself, and broadcasts `RequestVote` RPCs. Replicas vote for at most one candidate per term and only grant votes to candidates whose logs are at least as up-to-date as their own. A candidate becomes the leader once it receives votes from an election quorum; otherwise, it starts a new election after its timer expires.

While the leader election process is similar to Raft, Orca's approach differs from Raft in the election quorum size. Whereas Raft uses majority quorum ($\lceil \frac{n+1}{2} \rceil$) as an election quorum size, our approach has the base size (before any failure detection) the same as Flexible Paxos [24] and FlexiRaft [59]: $n - |Q_r| + 1$ in a system with n nodes and replication quorum size $|Q_r|$, to ensure there is at least one node intersection between replication and election quorums. However, in Orca, this number may change after failure detection. We reduce the election quorum size for every failed node ($r \in F_d$) and do not count the votes from that node in the leader election. As a result, the election quorum size becomes $n - |Q_r| - |F_d| + 1$ in a system with n nodes and replication quorum size of $|Q_r|$.

A candidate may have a stale view of the membership and temporarily accept votes from replicas that have already been excluded elsewhere. However, such candidates cannot collect sufficient votes to become the leader, since replicas with a newer membership view will reject the election.

Figure 4(a) illustrates a deployment with $n = 7$ and $|Q_r| = 3$. After two follower failures are detected and excluded from voting, the election quorum shrinks from 5 to 3. Consequently, when the leader and another follower subsequently fail, the remaining voting replicas can still elect a new leader. In contrast, Raft, Flexible Paxos, and FlexiRaft cannot make progress in this scenario because their election quorum requirements remain unchanged.

If more than k nodes fail simultaneously, two outcomes are possible. If the leader survives and a replication quorum can still be formed, Orca may continue to make progress, as shown previously

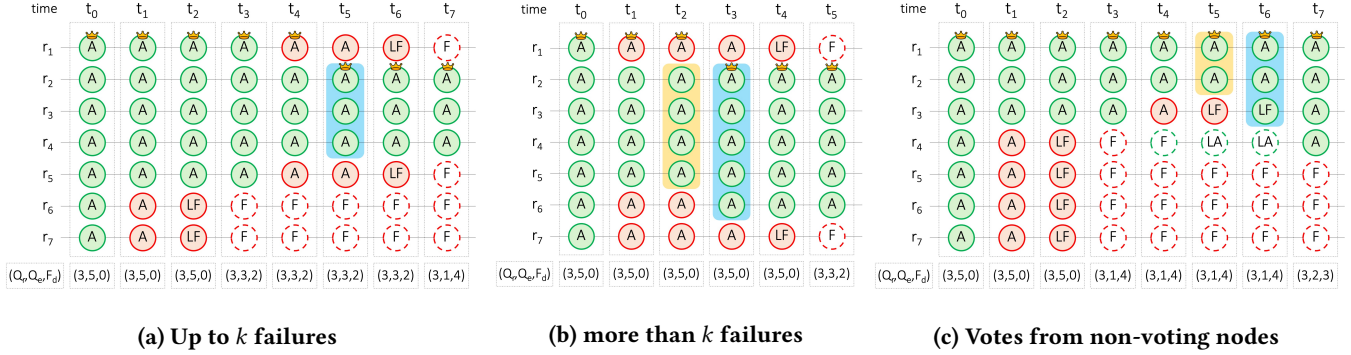


Figure 4: Different leader failure scenarios

in Figure 3(c). Otherwise, progress is impossible because the system cannot form an election quorum. Figure 4(b) illustrates such a case. With $n = 7$ and $|Q_r| = 3$, the leader and two followers fail simultaneously. Since the failures cannot be detected without an active leader, the election quorum remains unchanged at 5, while only four replicas remain alive. Consequently, no new leader can be elected until one of the failed replicas recovers.

This highlights the significance of selecting the appropriate value for k ; if no more than k simultaneous failures are allowed, such a scenario cannot occur. Even in the event of a network partition, if we have chosen k wisely, the larger partition will either include the leader or be capable of electing a new one. It's important to note that both Flexible Paxos and FlexiRaft will also be unable to make progress in this situation.

In certain executions, Orca may be unable to make progress even when Flexible Paxos and FlexiRaft can. This occurs because recovered replicas remain in state LA until their recovery is acknowledged and therefore do not participate in quorums. As a result, progress depends on the number of voting replicas rather than the total number of alive replicas. Figure 4(c) illustrates such a scenario. After four failures are detected and excluded, one replica recovers but remains in state LA. Before it regains voting rights, another voting replica fails. Although three replicas are alive, matching the replication quorum size, only two are voting. Consequently, Orca cannot make progress until another voting replica recovers. In contrast, Flexible Paxos and FlexiRaft can continue operating in this situation.

4.5 Correctness Arguments

In this section, we briefly discuss the safety and liveness of Orca. We mainly focus on agreement and termination properties.

LEMMA 4.1. If node r commits request m on log entry n , no other node commits m' ($m \neq m'$) on log entry n .

In Orca, the replication quorum Q_r has a constant size of $k + 1$, while the election quorum Q_e is sized $n - k - |F_d|$, where $|F_d|$ represents the number of detected failures at any given time. Achieving safety within the same term (under the same leader) is straightforward, as the leader is responsible for assigning log entries, and the failure or recovery of backup nodes does not affect the size of the replication quorum. However, the challenge arises when the

current leader fails, necessitating that the newly elected leader be aware of what has been replicated by the replication quorums in the most recent term, which requires intersection between the replication and election quorums. This is complicated because failure or recovery of nodes alters the size of the election quorum, and since nodes might have different views of the membership (voting and non-voting nodes), the number of votes needed for leader election (election quorum size) might be different across different nodes.

We divide the system's lifetime into a sequence of configurations. A system moves from configuration i to $i + 1$ whenever a FailureDetection or RecoveryDetection entry is replicated by a replication quorum, which subsequently results in an update to the set of detected failures, F_d . Hence, each configuration i can be represented with its set of detected failures F_d^i , e.g., $F_d^0 = \emptyset$ denotes the first configuration of the system where no failure has been detected yet.

Intuitively, nodes within F_d^i are non-voting nodes in configuration i . As a result, they cannot be part of any replication or election quorum in configuration i : $Q_r^i \cap F_d^i = \emptyset$ and $Q_e^i \cap F_d^i = \emptyset$. We also know that the election quorum size in i is $|Q_e^i| = n - k - |F_d^i| = n - |F_d^i| - |Q_r^i| + 1$. This can be rewritten as $|Q_e^i| + |F_d^i| + |Q_r^i| = n + 1$. Since $Q_e^i \cup F_d^i \cup Q_r^i = n$ and nodes in F_d^i are not part of Q_e^i or Q_r^i , we conclude that within configuration i , the election quorum Q_e^i and replication quorum Q_r^i have to intersect on at least one node.

To transition from a configuration F_d^i to another configuration F_d^{i+1} , a FailureDetection or RecoveryDetection entry needs to be replicated by at least a replication quorum Q_r (where the replication quorum size $|Q_r|$ is the same across all configurations). Hence, Q_r^i nodes from configuration i participate in (and hence are aware of) the transition to configuration $i + 1$. The lagging nodes in configuration i (nodes that are not aware of the new configuration) can elect a leader if they can collect $|Q_e^i| = n - |F_d^i| - |Q_r^i| + 1$ votes. However, out of n nodes, $|F_d^i|$ have been detected as faulty in configuration i and can not vote, and nodes in Q_r^i are aware of the transition to configuration $i + 1$ and will not vote. Hence, there will be at most $n - |F_d^i| - |Q_r^i|$ votes from lagging nodes in configuration i toward leader election, which is smaller than the election quorum size $|Q_e^i|$. Thus, nodes from an old configuration can not elect a new leader if a newer configuration exists.

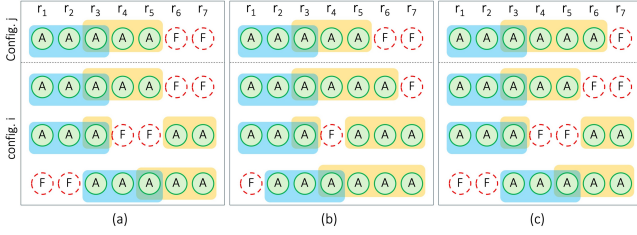


Figure 5: Replication (blue) and election (yellow) quorums intersect when (a) all nodes have the same election quorum size, (b) outdated nodes have a larger election quorum size, and (c) outdated nodes have a smaller election quorum size

Figure 5 illustrates three possible scenarios for two configurations i and j where $i \leq j$: (a) nodes in both configurations have the same election quorum size: $|Q_e^i| = |Q_e^j|$, (b) outdated nodes have a larger election quorum size: $|Q_e^i| > |Q_e^j|$, and (c) outdated nodes have a smaller election quorum size: $|Q_e^i| < |Q_e^j|$. In each scenario, a current configuration j and three possible past configurations i have been shown.

Our goal is to show that lagging nodes in an outdated configuration i can not construct an election quorum that does not intersect with the replication quorum of the current configuration j . This is easy to show as out of n nodes, in configuration i , $|F_d^i|$ nodes have already failed and do not vote. Moreover, $|Q_r^i|$ nodes also have a more complete log (as they vote for the transition to configuration $i + 1$), and do not vote for any outdated nodes in configuration i . Hence, at most $n - |F_d^i| - |Q_r^i|$ votes from nodes in configuration i can be collected, which is smaller than the election quorum size $|Q_e^i|$. This is evident in all three cases: the election quorum Q_e^i intersects with the replication quorum Q_r^i in configuration i . However, nodes within Q_r^i will not vote for any of the other nodes remaining in configuration i to assume the leader position, as the nodes in Q_r^i have a more complete log (i.e., they have been involved in changing the configuration to $i + 1$).

Observation. The safety proof relies on preserving the intersection between election and replication quorums across configurations. While Orca dynamically adjusts election quorums using a fixed replication quorum, the same proof applies symmetrically if replication quorums are adjusted based on a fixed election quorum, provided the intersection invariant is maintained.

LEMMA 4.2. A correct request m eventually completes.

Orca provides liveness *only* during periods of synchrony, due to the FLP result [21]. We utilize mechanisms used in Raft (and other partially synchronous consensus protocols) to increase the chance of liveness. In particular, Orca (1) requires the time it takes for a server to exchange messages (broadcast time) to be much smaller than the leader election timer, so that leaders can reliably send the heartbeat messages required to keep followers from starting elections (2) ensures every client request will eventually be received by the leader (either directly from a client or a follower informs the client about the current leader when the client retries) and if the quorum requirement is fulfilled, the leader will send a reply back to the client, and (3) utilizes random election timers to reduce the chance of conflicts when leader election happens.

Orca operates under the assumption that a maximum of k nodes can fail simultaneously. When this assumption is violated and more than k nodes fail in parallel, Orca can still make progress if the simultaneous failures do not include the leader and there are enough operational (voting) nodes to meet the replication quorum requirement. Otherwise, Orca will be unable to make progress. This condition is not unique to Orca; other consensus protocols, such as Raft (when k equals f) and FlexiRaft [59], share a similar requirement. Notably, Orca can still progress even if the total number of failures reaches $n - k - 1$, which is a more relaxed requirement compared to FlexiRaft or Raft, which can only tolerate k or $\lfloor \frac{n}{2} \rfloor$ failures, respectively.

Orca achieves this by modifying its configuration to designate failed nodes as non-voting members, thus reducing the election quorum. In the event of a leader failure, if the most recent configuration is i , then the nodes in F_d^i are considered non-voting, and Orca requires at least $|Q_e^i|$ voting nodes to remain operational. The value $|Q_e^i|$ is determined by the formula $n - |F_d^i| - k$. This requirement is still more relaxed than that of other consensus protocols; for instance, FlexiRaft mandates $n - k$, which is greater than or equal to $|Q_e^i|$, and the same applies to Raft, where k equals f .

5 EXPERIMENTAL EVALUATION

The primary goal of our evaluation is to examine the resilience and efficiency of Orca. Resilience is quantified by the number of failures Orca can tolerate under various conditions, while efficiency is assessed based on the overhead involved in the proposed mechanisms for failure detection and recovery. We benchmark our protocol against Raft [45] and FlexiRaft [59] (sometimes we refer to as FRaft). We selected Raft since it serves as the foundational protocol for Orca, and FlexiRaft was chosen due to its smaller replication quorum size (similar to Flexible Paxos), which allows us to analyze the overhead associated with Orca. The Orca implementation is built upon Raft, which we implemented in Go, integrating our failure detection, recovery mechanisms, and dynamic quorum adjustments. We implemented FlexiRaft using our Raft implementation.

We also incorporate batching techniques in our implementation. As discussed in Section 4.1, when the leader receives a request from a client, it appends the request to its own log and replicates it using the AppendEntries RPC once its heartbeat timer expires. However, if a high volume of requests is received, the system may end up spending a considerable amount of time just waiting for the leader's heartbeat timer to expire. To optimize this process, we introduce a batch timer that operates independently and is roughly set to 0.1 of the heartbeat timer. Additionally, we introduce two parameters: LowBatchSize and HighBatchSize. Upon receiving a request from a client, the leader will initiate the batch timer instead of waiting for the heartbeat timer to expire. The leader will then flush the batch either when the batch timer elapses or when the number of received requests exceeds LowBatchSize before the batch timer runs out. This approach significantly reduces latency, with the trade-off being a minimal amount of "useless" heartbeat signals during normal operation, where we use the batch flush, but the heartbeat is still running. In practice, some followers may process requests more slowly than others. Therefore, HighBatchSize is defined as the maximum number of log entries per AppendEntries RPC, and

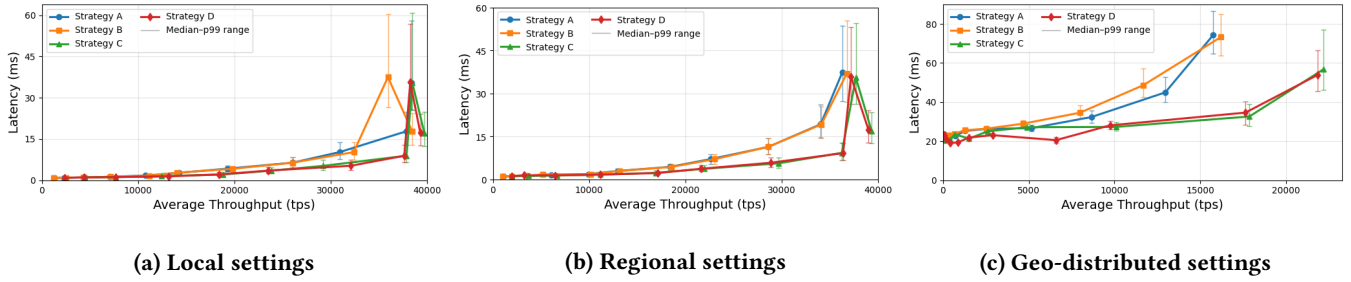


Figure 6: Latency-throughput graph with increasing concurrent clients. Lines show p90 latency; error bars span median to p99.

it should be at least equal to (ideally greater than) LowBatchSize to allow the slower followers to catch up.

In our evaluation, we investigate how the performance of Orca and its competitors is influenced by various parameters, including:

- Transactions batch size (Section 5.2),
- Geo-distribution of nodes (Section 5.2),
- Failure of the leader and follower nodes (Section 5.3),
- Number of tolerated failures, k (Section 5.4), and
- Network size, n (Section 5.5), and
- Comparison with reconfiguration protocols (Section 5.6).

5.1 Experimental Setup

The experiments were conducted on the Amazon EC2 platform, utilizing m5.xlarge instances that feature 4 vCPUs and 16GB of RAM, powered by Intel Xeon Platinum 8000 series processors operating at 3.1 GHz, with a bandwidth capacity of 10 Gbps. Each virtual machine (VM) hosts either a single node (replica) or the client, with the number of instances varying across different experiments.

We deployed three distinct distributed configurations on the AWS EC2 instances: (1) *Local*, where all nodes are situated within a single availability zone, resulting in roughly 0.2 to 0.7 ms communication latency, (2) *Regional*, where nodes are distributed across multiple availability zones within the same region, experiencing around double the local latency, typically ranging from 0.5 to 1.2 ms, and (3) *Geo-distributed*, where nodes are located in 5 different regions across Europe, resulting in latency ranging from 5 to 10 ms.

For our evaluations, we employed the YCSB benchmark [13], implemented in Go [47], with a focus on a write-only workload to assess replication performance. Our default experimental configuration consists of 7 nodes within a single availability zone, with a parameter $k = 2$, a LowBatchSize of 256, a HighBatchSize of 512 and 512 concurrent clients. We vary timers (i.e., election timer, heartbeat interval, batch flush timeout, and time between failures) based on the deployment: for the local deployment, the timers are 150 – 300 ms, 50 ms, 4 ms, 5 seconds, respectively. For regional deployment, the timers are double the local to match the difference in latency, and for geo-distributed deployment, the timers are roughly quadruple the regional deployment to make it close to the difference in latency but also not too long. We restrict the number of outstanding requests on each node to 100 for any other node, and we set the retry count for failure detection to 3 failed heartbeats.

For reporting throughput and latency, we have the client make a report for every second of the system running, a feature from [47].

Each experimental trial runs for 30 to 240 seconds, depending on the time between failures, and the maximum number of tolerated failures. Each run also includes a 30 second warm-up period. The results reported are the average outcomes from three runs.

5.2 Varying Batch Sizes in Distributed Settings

We first analyze how varying batch sizes affect the performance of our protocol across different networks and workloads. By tuning the batch size and the number of concurrent clients, we aim to determine the optimal configuration that maximizes performance.

We run the experiment with 7 nodes in local (Figure 6(a)), regional (Figure 6(b)), and geo-distributed (Figure 6(c)) settings, and increase the number of concurrent clients from 1 to 2^{10} . We further implement four different strategies to choose the batch size:

- Strategy A: low batch size and high batch size are the same and equal to the number of clients,
- Strategy B: low batch size is equal to the number of clients, while high batch size is double the number of clients,
- Strategy C: low batch size is half of the number of clients, while high batch size is equal to the number of clients, and
- Strategy D: low batch size is half of the number of clients, while high batch size is double the number of clients

Results. In the local deployment, as demonstrated in Figure 6(a), Orca achieves its peak throughput with 512 clients and batching strategy C by processing 39.7k transactions per second. Increasing the number of clients beyond this point provides little benefit, indicating that the system has reached saturation. Strategies C and D perform consistently better than strategies A and B because they only need to wait for half of the client requests before flushing a batch. The latency of Orca increases almost linearly with the number of concurrent clients in all four strategies, as larger batches require more processing and communication time.

In the regional deployment, where nodes are distributed across multiple availability zones within the same region, as shown in Figure 6(b), both throughput and latency are comparable to those seen in the local deployment. Orca reaches its maximum throughput with 512 clients using strategy C, closely followed by strategy D. This similarity arises because the additional network latency is largely offset by the latency involved in processing larger batches.

Compared to the local and regional, geo-distribution deployments significantly reduce throughput and increase latency due to the cost of wide-area replication. With 512 clients, throughput is

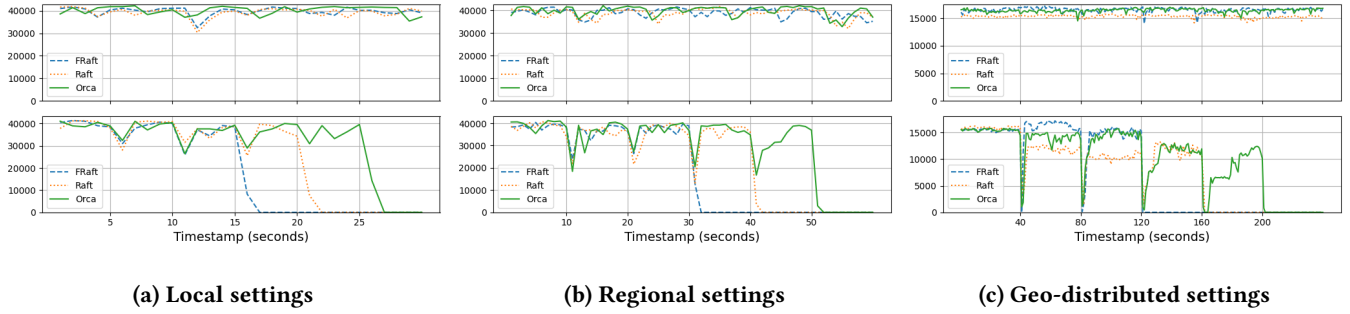


Figure 7: Throughput of different protocols with no failure (top) and leader failure (bottom)

~55% lower, and latency is ~121.6% higher than in the local deployment. The performance difference between batching strategies A/B and C/D is also more pronounced in the geo-distributed setting. With 512 clients, strategies C and D achieve 15% higher throughput and 32% lower latency. This is expected because the latency gap between fast and slow replicas is larger across regions, making it more beneficial to flush batches without waiting for all client requests. Unlike the local and regional deployments, latency remains nearly constant as the number of clients increases up to 512 under strategies C and D, indicating that network propagation dominates performance in this regime. Beyond this point, throughput continues to increase, although with diminishing returns.

Summary. Across all three deployments, 512 concurrent clients provide a reasonable operating point for Orca. While 256 clients offer a slightly better throughput-latency tradeoff in local and regional deployments and 1024 clients achieve higher throughput in the geo-distributed setting, 512 clients provide the best balance across all environments. Regarding batch size, using half of the number of clients as the low batch size consistently demonstrates the best performance. For the high batch size, using the number of clients is slightly better than doubling it. Therefore, we use batching strategy C in the remaining experiments.

5.3 Performance Under Different Failures

We analyze the performance of protocols under no failure, leader failure, follower failure, oscillating leader failure, and oscillating follower failure. For leader or follower failure, we simulate a failure every 5 seconds in local, 10 seconds in regional, and 40 seconds in geo-distributed settings (as mentioned in Section 5.1) until progress halts. In the oscillating cases, we alternate between failing and recovering the leader or follower every second. We additionally evaluate the no-failure and leader-failure scenarios in regional and geo-distributed deployments, but skip the other three failure types as results follow similar patterns across deployments.

Results. Under no failure (top graph of Figure 7(a)), all protocols perform similarly in the local deployment, achieving 39.3–40.1 ktps. The same pattern is observed under oscillating and leader failure scenarios (the two bottom graphs in Figure 8), indicating that Orca’s failure detection incurs negligible overhead.

As shown in the bottom graphs of Figure 7, all deployments follow a similar pattern for leader failure. FlexiRaft stops progressing after 3 leader failures, followed by Raft at 4, and Orca at 5. This

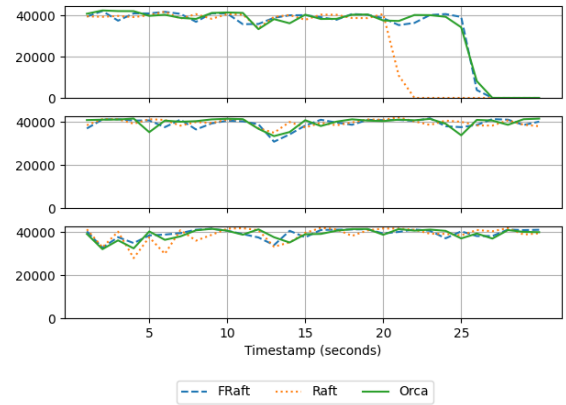


Figure 8: Throughput of different protocols with follower failure (top), oscillating follower failure (middle), and oscillating leader failure (bottom) on local settings

is attributed to FlexiRaft’s smaller replication quorum requiring a larger election quorum. Although Orca also has a smaller replication quorum, the failure detection and dynamic quorum make it more resilient to leader failure by shrinking the election quorum size upon failure detection. In follower failure scenarios, as shown in the top graph of Figure 8, Raft stops progressing after 4 failures, making it the least resilient out of the three, whereas Orca and FlexiRaft stop after 5 failures, demonstrating their resilience due to smaller replication quorum sizes allowing progress with fewer nodes if the leader remains operational. Throughput does not drop to zero immediately after a failure threshold is reached because failures are injected via RPCs and therefore do not align exactly with the vertical markers in the figures.

Figure 7 shows that results are similar under local and regional deployments because batching largely hides the additional intra-region latency. In geo-distributed deployments, Raft’s larger replication quorum becomes a disadvantage because progress depends on more geographically distant replicas. Consequently, with no failures, Raft achieves 15.3 ktps, while Orca and FlexiRaft outperform it by 6.7% and 7.3%, respectively. The same trend appears during leader failures. As failures accumulate, throughput decreases because leaders with more complete logs are more likely to be elected,

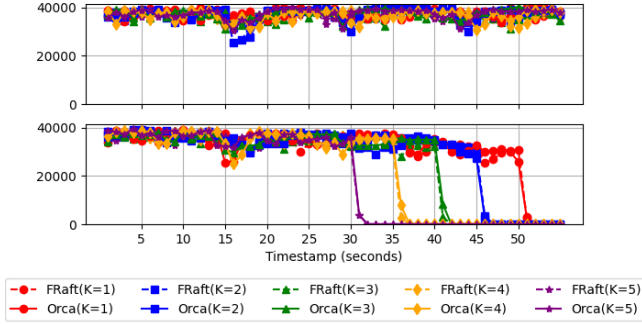


Figure 9: Throughput with no failure (top) and follower failure (bottom).

and leader failures progressively remove these replicas. Interestingly, FlexiRaft’s throughput improves after the first failure, likely because its initial leader placement was suboptimal.

Summary. This experiment demonstrates that Orca not only incurs negligible overhead but also tolerates a greater number of leader and follower failures, making it the most resilient of the three protocols.

5.4 Varying the Quorum Size

We next measure the impact of quorum size on the performance of different protocols. We run the experiment with 11 nodes and various k where the election quorum size is $|Q_e| = n - k$ and replication quorum size is $|Q_r| = k + 1$. Each benchmark runs for 55 seconds (the point at which no more progress can be made by any of the protocols under any k). We run three different failure models: no failure, leader failure, and follower failure (refer to Section 5.3 for more details) to show how the quorum size affects the resilience of the three protocols. Note that Raft is not included because FlexiRaft with $k = 5$ is equivalent to Raft.

Results. Figure 9 shows that under no failure, the throughput of both protocols across all k values exhibits only slight variations, with higher k showing lower throughput on both protocols. The average throughput for $k = 1$ is 37.6 ktps, while for $k = 5$ it drops to 36.9 ktps. This is anticipated, as a lower k indicates that fewer nodes need to be involved in the replication before responding to the client. The minimal difference across different k values is also expected due to the uniform latencies between nodes.

The number of follower failures that Orca and FlexiRaft can tolerate is the same, as demonstrated in Figure 9. Their throughput remains roughly comparable, consistent with the findings from Section 5.3. Additionally, Figure 9 indicates that a smaller replication quorum size enables both protocols to manage more follower failures. This occurs because a stable leader requires fewer nodes to remain alive when the replication quorum size is reduced.

In contrast, the results for leader failures show a different trend for FlexiRaft, while Orca maintains similar performance as seen with follower failures. As depicted in Figure 10, FlexiRaft can tolerate fewer leader failures when the replication quorum size is smaller. For instance, FlexiRaft processed only 375,000 transactions with $k = 1$ under leader failures, compared to 1,712,000 under follower failures (78% reduction). This illustrates the quorum size trade-off: a smaller replication quorum requires a larger election quorum

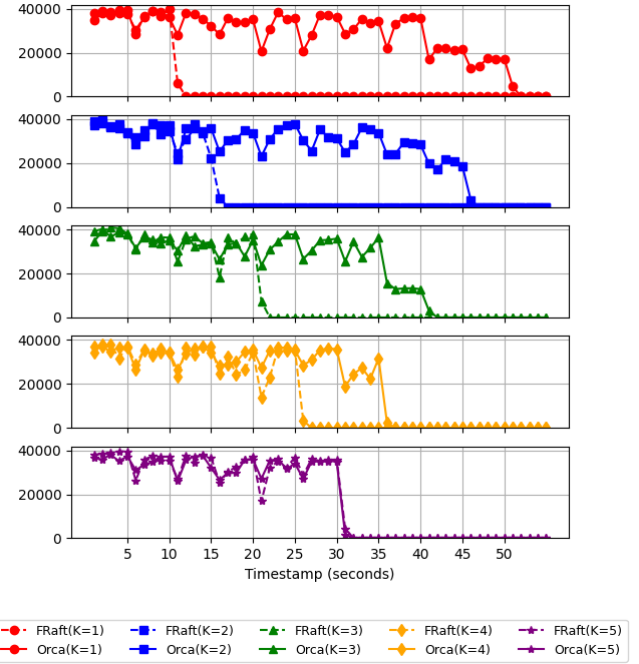


Figure 10: Throughput under leader failure.

size. However, Orca circumvents this trade-off through its failure detection and dynamic quorum mechanisms, enabling it to tolerate the same number of failures regardless of whether they involve leaders or followers. In this experiment, Orca processed 1.543 M transactions during leader failures and 1.731 M during follower failures when $k = 1$ (10% reduction).

Summary. The experiment illustrates that Orca does not face the same trade-off in leader failure tolerance that FlexiRaft does, even while utilizing a smaller replication quorum size. Essentially, Orca benefits from the advantages of a smaller replication quorum without incurring the associated drawbacks.

5.5 Varying the Network Size

We measure the impact of network size on the performance of the protocols by varying the network size from 5 to 11, 17, and 23 while keeping k as 2 for all experiments. We do not consider 3-node deployments, as they can not benefit from the mechanisms introduced by flexible quorums or Orca. Each experiment runs for 60 seconds with no failure.

Results. As shown in Figure 11, the performance across all three protocols is nearly identical with 5 and 11 nodes. However, at 17 nodes, Raft performs 10% worse than both Orca and FlexiRaft, and this degradation increases to 36% with 23 nodes. This trend aligns with previous experiments, indicating that a smaller replication quorum size enhances performance. Notably, Orca and FlexiRaft experience significant performance degradation as the number of nodes increases, albeit not to the same extent as Raft, despite maintaining the same replication quorum size without failures. In our investigation, we analyzed the elapsed time for various phases: (a)

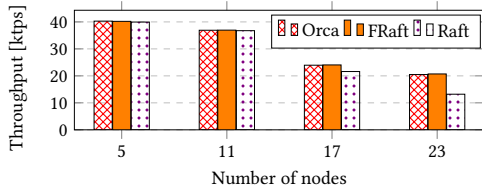


Figure 11: Throughput with different n with no failure.

the leader preparing the AppendEntries, (b) sending it, and waiting for acknowledgments, (c) processing the acknowledgments, and (d) the followers receiving and acknowledging the AppendEntries. We found that AppendEntries preparation and acknowledgment processing time on the leader is approximately $50\mu s$ with 23 nodes, contributing minimally to latency. Additionally, the time for followers to begin sending acknowledgments averages around $80\mu s$. The primary source of performance decline lies in the interval between the leader sending the AppendEntries and receiving acknowledgments, primarily due to network bandwidth limitations.

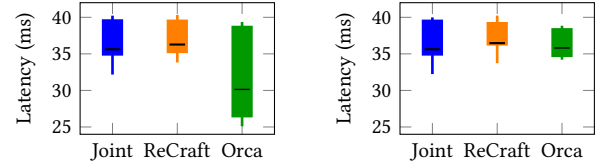
Summary. The smaller replication quorum size enables Orca and FlexiRaft to outperform Raft as the number of nodes increases.

5.6 Comparing Reconfiguration Protocols

Existing reconfiguration protocols are not directly translatable to Orca’s setting because Orca uses flexible (asymmetric) quorums. Majority-based protocols such as Joint Consensus [45] and ReCraft [58] assume symmetric majority quorums, while protocols such as Vertical Paxos [34] and Matchmaker Paxos [57] primarily modify acceptor sets, which more closely correspond to replication quorums. In contrast, Orca dynamically adjusts voting membership and election quorum sizes independently. More generally, Orca can be mapped onto a conventional reconfiguration setting by decomposing membership changes into quorum-specific steps. Instead of changing both quorums simultaneously, an Orca-style reconfiguration first adjusts one quorum dimension and then the other while preserving quorum intersection throughout the transition.

We evaluate this translation in a geo-distributed setting with 7 nodes. Each experiment runs for 300 seconds and alternates every second between removing four nodes and adding four nodes. We compare against Joint Consensus and ReCraft because both perform reconfiguration through the same replicas that participate in consensus, without requiring separate reconfiguration services. We repeat each experiment three times. The leader placement is identical across all protocols, while each run uses a different leader location to capture the effect of geo-distributed latency.

Results. During add-node reconfiguration, Orca requires only two nodes to enter and complete the first transition step, since only the election quorum is resized initially. Joint Consensus requires two nodes to enter transition and four to complete, while ReCraft requires six nodes to complete. Consequently, Orca achieves lower reconfiguration latency in geo-distributed environments with noticeable latency variance, as depicted in Figure 12(a). Although Orca achieves the lowest median latency, it also exhibits a wider latency distribution. This behavior is caused by the leader placement in the geo-distributed deployment. When the leader can satisfy the required quorum using nearby replicas, reconfiguration completes



(a) Adding nodes

(b) Removing nodes

Figure 12: Latency distribution: thin colored lines show min-max latency, thick colored bars show p50–p95 latency, and black horizontal ticks show average latency.

quickly. However, when one of the required quorum members is geographically distant, the benefit of the smaller quorum is reduced, increasing the latency. During remove-node reconfiguration, Orca and Joint Consensus both require 4 nodes to complete reconfiguration, while ReCraft requires 6. As a result, Orca performs similarly to Joint Consensus, while ReCraft incurs slightly higher latency due to its larger quorum requirements, as shown in Figure 12(b).

Summary. By decomposing reconfiguration into quorum-specific steps, Orca reduces the quorum requirements during transition and achieves lower latency compared to Joint Consensus and ReCraft.

6 CONCLUSION

In this paper, we presented Orca, a resilient crash-fault-tolerant consensus protocol that integrates flexible quorum and dynamic quorum mechanisms with effective failure detection and recovery strategies to improve resilience by tolerating more failures. Orca integrates Raft-style log replication with Flexible Paxos quorum construction. Additionally, Orca leverages the independence of node failures and the time window between consecutive failures to identify faulty nodes, temporarily excluding them from the set of voting nodes and dynamically adjusting the election quorum size. Through extensive experimental evaluations, we have illustrated that Orca not only tolerates a greater number of failures but also minimizes system overhead, ensuring seamless transaction processing without compromising safety or liveness. More broadly, temporal failure assumptions may serve as a general design principle beyond Orca, enabling consensus protocols to dynamically adapt quorum construction based on observed failure patterns. In particular, for strong-leader protocols, where leader election and replication are distinct phases, this approach applies most directly. Such protocols can be transformed to use flexible quorums, separating election and replication requirements, then augmented with failure detection to exclude failed replicas and dynamically adjust quorum sizes. Other protocol families may require different mechanisms; those lacking a strong leader or clear election/replication separation may not support quorum adaptation as directly. Still, the core principle holds: detected failed replicas need not participate in all quorum calculations, allowing systems to exploit temporal failure patterns and tolerate additional failures over time.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback. This work is funded by NSF under grant number 2541586.

REFERENCES

- [1] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tefvik Kosar. 2019. WPaxos: Wide area network flexible consensus. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 211–223.
- [2] Carlos Almeida and Paulo Verissimo. 1996. Timing Failure Detection and Real-Time Group Communication in. In *Euromicro Workshop on Real-Time Systems*, Vol. 96. IEEE, 230.
- [3] Algirdas Avizienis. 1985. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering* 12 (1985), 1491–1501.
- [4] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services. In *Conf. on Innovative Data Systems Research (CIDR)*.
- [5] Christian Berger, Hans P Reiser, João Sousa, and Alysson Bessani. 2019. Resilient wide-area Byzantine consensus using adaptive weighted replication. In *Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 183–18309.
- [6] Kenneth P Birman, Thomas A Joseph, Thomas Raechle, and Amr El Abbadi. 1985. Implementing fault-tolerant distributed objects. *Trans. on Software Engineering* 6 (1985), 502–508.
- [7] F. Brasileiro, F. Greve, A. Mostéfaoui, and M. Raynal. 2001. Consensus in one communication step. In *Int. Conf. on Parallel Computing Technologies (PaCT)*. Springer, 42–50.
- [8] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, and Harry Li. 2013. TAO: Facebook’s Distributed Data Store for the Social Graph. In *Annual Technical Conf. (ATC)*. USENIX Association, 49–60.
- [9] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Symposium on Principles of Distributed Computing (PODC)*. ACM, 398–407.
- [10] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)* 43, 2 (1996), 225–267.
- [11] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2021. PigPaxos: Devouring the communication bottlenecks in distributed consensus. In *SIGMOD Int. Conf. on Management of Data. ACM*, 235–247.
- [12] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. 2002. On the quality of service of failure detectors. *IEEE Transactions on computers* 51, 5 (2002), 561–580.
- [13] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Symposium on Cloud computing (SoCC)*, 143–154.
- [14] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, and Peter Hochschild. 2013. Spanner: Google’s globally distributed database. *Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.
- [15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon’s highly available key-value store. *Operating Systems Review (OSR)* 41, 6 (2007), 205–220.
- [16] Dan Dobre, Matthias Majumke, Marco Serafini, and Neeraj Suri. 2010. HP: Hybrid paxos for WANs. In *European Dependable Computing Conf. (EDCC)*. IEEE, 117–126.
- [17] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. 1987. On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)* 34, 1 (1987), 77–97.
- [18] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.
- [19] Michael Eischer and Tobias Distler. 2018. Latency-aware leader selection for geo-replicated Byzantine fault-tolerant systems. In *Int. Conf. on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 140–145.
- [20] Christof Fetzer and Flaviu Cristian. 1996. Fail-awareness in timed asynchronous systems. In *Symposium on Principles of Distributed Computing (PODC)*. ACM, 314–321.
- [21] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32, 2 (1985), 374–382.
- [22] Stephanie Forrest, Anil Somayaji, and David H Ackley. 1997. Building diverse computer systems. In *Workshop on Hot Topics in Operating Systems*. IEEE, 67–72.
- [23] Indranil Gupta, Tushar D Chandra, and Germán S Goldszmidt. 2001. On scalable and efficient distributed failure detectors. In *ACM symposium on Principles of distributed computing (PODC)*. 170–179.
- [24] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2017. Flexible Paxos: Quorum Intersection Revisited. In *Int. Conf. on Principles of Distributed Systems (OPODIS)*.
- [25] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE, 245–256.
- [26] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, and Yang Zhang. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proc. of the VLDB Endowment* 1, 2 (2008), 1496–1499.
- [27] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. 1997. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *Int. Workshop on Distributed Algorithms*. Springer, 126–140.
- [28] A.D. Kshemkalyani and M. Singhal. 2011. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press. <https://books.google.com/books?id=G7SZ32dPuLgC>
- [29] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.
- [30] Leslie Lamport. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.
- [31] Leslie Lamport. 2005. Generalized consensus and Paxos. (2005).
- [32] Leslie Lamport. 2006. Fast paxos. *Distributed Computing* 19, 2 (2006), 79–103.
- [33] Leslie Lamport. 2019. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*. 277–317.
- [34] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2009. Vertical paxos and primary-backup replication. In *Symposium on Principles of distributed computing (PODC)*. ACM, 312–313.
- [35] Leslie Lamport and Mike Massa. 2004. Cheap paxos. In *Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE, 307–314.
- [36] Butler Lampson. 2001. The ABCD’s of Paxos. In *Symposium on Principles of Distributed Computing (PODC)*, Vol. 1. 13.
- [37] Harry C Li, Allen Clement, Amitanand S Ayier, and Lorenzo Alvisi. 2007. The paxos register. In *IEEE Int. Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 114–126.
- [38] Barbara Liskov and James Cowling. 2012. Viewstamped replication revisited. (2012).
- [39] MongoDB. 2026. MongoDB. <https://www.mongodb.com/>. Accessed: 2026-05-06.
- [40] Louise E Moser, Peter M Melliar-Smith, Priya Narasimhan, Lauren A Tewksbury, and Vana Kalogeraki. 1999. The Eternal system: An architecture for enterprise applications. In *Int. Enterprise Distributed Object Computing Conf. (EDOC)*. IEEE, 214–222.
- [41] Le Narzul et al. 1999. Probabilistic analysis of a group failure detection protocol. In *Int. Workshop on Object-Oriented Real-Time Dependable Systems*. IEEE, 156–162.
- [42] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2018. Dpaxos: Managing data closer to users for low-latency and mobile applications. In *SIGMOD Int. Conf. on Management of Data. ACM*, 1221–1236.
- [43] Brian M Oki and Barbara H Liskov. 1988. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Symposium on Principles of Distributed Computing (PODC)*. ACM, 8–17.
- [44] Diego Ongaro. 2014. *Consensus: Bridging Theory and Practice*. Ph.D. Dissertation. Stanford University. <https://doi.org/10.5555/AAI28121474>
- [45] Diego Ongaro and John K Ousterhout. 2014. In search of an understandable consensus algorithm. In *Annual Technical Conf. (ATC)*. USENIX Association, 305–319.
- [46] Fernando Pedone. 2001. Boosting system performance with optimistic distributed protocols. *Computer* 34, 12 (2001), 80–86.
- [47] PingCAP. 2025. go-ycsb: A Go port of Yahoo! Cloud Serving Benchmark (YCSB). <https://github.com/pingcap/go-ycsb>. Accessed: 2025-08-31.
- [48] André Schiper. 2006. Dynamic group communication. *Distributed Computing* 18, 5 (2006), 359–374.
- [49] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *Computing Surveys (CSUR)* 22, 4 (1990), 299–319.
- [50] William Schultz, Siyuan Zhou, Ian Dardik, and Stavros Tripakis. 2021. Design and analysis of a logless dynamic reconfiguration protocol. *arXiv preprint arXiv:2102.11960* (2021).
- [51] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. 2008. BFT Protocols Under Fire.. In *Symposium on Networked Systems Design and Implementation (NSDI)*, Vol. 8. USENIX Association, 189–204.
- [52] João Sousa and Alysson Bessani. 2015. Separating the WHEAT from the chaff: An empirical design for geo-replicated state machines. In *Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 146–155.
- [53] Pasindu Tennage, Cristina Basescu, Leftis Kokoris-Kogias, Ewa Syta, Philipp Jovanovic, Vero Estrada-Galananes, and Bryan Ford. 2023. QuePaxa: Escaping the tyranny of timeouts in consensus. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 281–297.
- [54] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. 1998. A gossip-style failure detection service. In *IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware)*. Springer, 55–70.
- [55] Robbert Van Renesse, Nicolas Schiper, and Fred B Schneider. 2014. Vive la différence: Paxos vs. viewstamped replication vs. zab. *IEEE Transactions on Dependable and Secure Computing* 12, 4 (2014), 472–484.
- [56] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2010. EBAWA: Efficient Byzantine agreement for wide-area networks. In *Int. Symposium on High Assurance Systems Engineering (HASE)*. IEEE, 10–19.
- [57] Michael J. Whittaker, Neil Girdharan, Adriana Szekeres, Joseph M Hellerstein, Heidi Howard, Faisal Nawab, and Ion Stoica. 2021. [Solution] Matchmaker Paxos:

- A Reconfigurable Consensus Protocol. *J. Syst. Res.* 1 (2021).
- [58] Kezhi Xiong, Soonwon Moon, Joshua H. Kang, Bryant Curto, Jieung Kim, and Ji-Yong Shin. 2025. ReCraft: Self-Contained Split, Merge, and Membership Change of Raft Protocol. In *Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE, 157–170.
- [59] Ritwik Yadav and Anirban Rahut. 2023. FlexiRaft: Flexible Quorums with Raft.. In *CIDR*.