RACS and SADL: Towards Robust SMR in the Wide-Area Network

Abstract

Most popular consensus algorithms deployed in the crash fault tolerant setting chose a leader-based architecture in order to achieve the lowest latency possible. However, when deployed in the wide area they face two key "robustness" challenges. First, they lose liveness when the network is unreliable because they rely on timeouts to find a leader. Second, they cannot have a high replication factor because of the high load imposed on the leader-replica making it a bottleneck. This effectively limits the replication factor allowed, for a given level of throughput, thus lowering the fault tolerance threshold (f). In this paper, we propose RACS and SADL, a modular state machine replication algorithm that addresses these two robustness challenges.

To achieve robustness under adversarial network conditions, we propose RACS, a novel crash fault-tolerant consensus algorithm. RACS consists of two modes of operations – synchronous and asynchronous – that always ensure liveness. RACS leverages the synchronous network to minimize the communication cost to $O(n)$ and matches the lower bound of $O(n^2)$ at adversarial-case executions. To avoid the leader bottleneck and to allow higher replication factor, without sacrificing the throughput, we then propose SADL, a novel consensus-agnostic asynchronous dissemination layer. SADL separates client command dissemination from the critical path of consensus and distributes the overhead evenly among all the replicas. The combination of RACS and SADL (SADL-RACS) provides a robust and high-performing state machine replication system.

We implement and evaluate RACS and SADL-RACS in a wide-area deployment running on Amazon EC2. Our evaluation shows that in the synchronous execution, SADL-RACS delivers up to 500k cmd/sec throughput, in less than 800ms latency, outperforming Multi-Paxos and Rabia by 150% in throughput, at a modest expense of latency. Furthermore, we show that SADL-RACS delivers 196k cmd/sec throughput under adversarial network conditions, whereas Multi-Paxos and Raft completely lose liveness. Finally, we show that SADL-RACS scales up to 11 replicas with 380k cmd/sec, in contrast to Multi-Paxos's 130k cmd/sec throughput.

1 Introduction

State machine replication (SMR) [11] enables a set of replicas to maintain state while remaining resilient to failures, and is widely used in distributed applications [10, 24, 30]. At the heart of the SMR lies the consensus protocol, which allows replicas to agree on a single history of commands. The widespread adoption of consensus protocols in databases [6, 15, 60], distributed filesystems [22, 34, 46], and social networks [9, 29], requires further research in the wide-area setting.

Existing popular consensus protocols face two key "robustness" challenges in the wide-area network (WAN), because they are leader-based [26, 43]. First, their liveness is fragile when the network is unreliable, or under a distributed denial-of-service (DDoS) attack [41, 50, 53]. Second, the high load imposed on the leader-replica reduces the number of redundant copies (replication factor) allowed [8, 62], thus lowering the fault tolerance threshold (f).

In this work, we propose a novel crash fault tolerant SMR framework that enables robust WAN SMR. First, **RACS** (**R**esilient **A**synchronous **C**rash fault tolerant **S**tate Machine Replication) provides robustness against adversarial network conditions using a novel randomized consensus core, and addresses the liveness challenge. Second, **SADL** (**S**imple **A**synchronous **D**issemination **L**ayer) scales RACS to a higher number of replicas such that SADL-RACS enables deploying more replicas without sacrificing throughput.

The underlying reason for the liveness challenge is that leader-based protocols such as Raft [43] employ a leader-replica to drive the protocol, and assume a partially synchronous network for liveness. In WAN, the network conditions can often become adversarial due to transient slowdowns, targeted DDoS attacks [50], or misconfigurations of networks [28]. Under such adversarial conditions, the network delays become unpredictable, hence, the leader-based protocols fail. The theoretical literature has proposed the use of leaderless randomized protocols [7] that guarantee liveness under adversarial networks, however, the need for $O(n^2)$ message overhead under normal-case synchronous executions is prohibitive for practical deployments.

In this paper, we ask the following question: can we design a protocol that can achieve the best of both worlds: optimal performance under synchronous network conditions and robustness under adversarial network conditions? Withstanding adversarial network conditions while preserving the synchronous case performance is challenging, because, to preserve the synchronous performance, one has to rely on a leader-based design, whereas, to preserve resiliency against adversarial networks, one has to use randomization. It is non-trivial to merge a leader-based protocol and a randomized protocol, given that these two paradigms assume different network conditions and make different design assumptions.

This paper proposes RACS: a novel randomized consensus protocol that concurrently achieves (1) optimal synchronous case performance and (2) robustness to adversarial network

conditions. RACS enables a leader based one round-trip fast path and provides a randomized recovery path that keeps committing new commands. Unlike Multi-Paxos and Raft that stop committing commands during the view-change phase, RACS's fallback path continues to commit new commands.

Although RACS enables robustness against adversarial networks, RACS alone cannot solve the second challenge of increasing the fault tolerant threshold (f), without sacrificing the throughput. Throughput of RACS and existing leader-based protocols such as Multi-Paxos are bottlenecked by the leader replica's available network bandwidth and computational resources. When scaling to more than 3–5 replicas RACS and existing leader-based protocols have to sacrifice the performance, as we show in Section 7, due to increased utilization of resources at the leader-replica. Hence, RACS and existing leader-based protocols face a trade-off between high throughput and high resilience: low replication factor enables high throughput but low fault tolerant threshold, whereas, a high replication factor enables high fault tolerant threshold, but low throughput.

This trade-off is inherent because of the strong coupling between data (commands to be executed) and the consensus messages. In existing leader-based protocols and RACS, consensus messages carry a batch of client commands, such that the size of a consensus message is mostly influenced by the command batch size (sizes in the order of 100KB), whereas the consensus metadata only accounts for a few bytes. Hence, with increasing replication factor, the majority of leader replica's bandwidth is spent for client command dissemination. Hence, the asymptotic linear message complexity of consensus algorithms does not necessarily reflect the experimental performance, when deployed.

This paper proposes SADL; a novel command dissemination layer. SADL decouples the command dissemination from the critical path of consensus. SADL disseminates client commands, asynchronously, and without employing a designated leader replica, thus distributing the overhead evenly across all the participating replicas. With SADL in place, the leader-replica in RACS only has to send and receive consensus metadata, which are only a few bytes. Hence SADL-RACS scales to to larger number of replicas, without sacrificing throughput.

SADL preserves throughput, but incurs an additional one round trip latency cost, which can be substantial under low replication factors (3 and 5) and under low load, as we empirically show in Section 7. To accommodate low latency in such cases, we propose a hybrid extension that combines SADL and pipelining, such that applications can dynamically switch between pipelining and SADL depending on the workload and network conditions.

We implemented and evaluated prototypes of RACS and SADL-RACS in Go [36] and compared them against the existing implementations of Multi-Paxos[26], Raft[43], EPaxos[38], and Rabia[44]. We evaluated RACS and SADL-RACS on Amazon EC2 in a multi-region WAN setting. We first show that RACS delivers 200k cmd/sec in throughput under 300ms median latency, comparable to Multi-Paxos's 200k cmd/sec throughput, under synchronous normal case conditions. Second, we show that RACS and SADL-RACS provide 28k cmd/sec and 196k cmd/sec of throughput, respectively, under adversarial network conditions, and outperform Multi-Paxos and Raft which provide 2.8k cmd/sec in the same setting. Finally, we show that SADL-RACS scales up to 11 number of replicas, while delivering a throughput of at least 380k cmd/sec, in contrast, Multi-Paxos and RACS sacrifice the throughput with increasing replication factor.

This paper makes the following key contributions:

- We propose RACS, a novel practical randomized consensus protocol that concurrently provides liveness under adversarial network conditions and high performance under normal case synchronous network conditions.
- We provide formal proofs of RACS.
- We propose SADL, a novel asynchronous command dissemination protocol that enables RACS to support higher fault tolerant threshold (f), without sacrificing the throughput.
- A working prototype and experimental analysis of RACS and SADL-RACS under both normal and adversarial network conditions.

2 Background

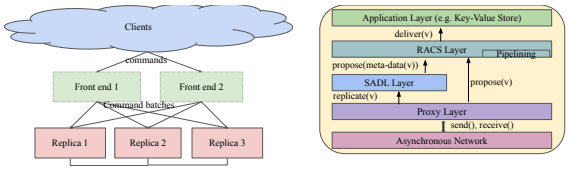
2.1 Threat Model and Assumptions

We consider a system with n replicas. Up to f (where $n \geq 2f + 1$) number of replicas can crash, but replicas do not equivocate nor commit omission faults [11].

We assume first-in-first-out (FIFO) perfect point-to-point links [11] between each pair of replicas; messages from any correct replica p_i to any correct replica p_j are eventually delivered, in the FIFO manner. In practice, TCP[51] provides FIFO perfect point-to-point links. The replicas are connected in a logical-complete graph. We say that a replica broadcasts a message m if it sends m to all n replicas.

We assume a content-oblivious [5] network adversary; the adversary may manipulate network delays, but cannot observe the message content nor the internal replica state. In practice, TLS [47] encrypted channels between each pair of replica satisfy this assumption.

Let Δ be the upper bound on message transmission delay and GST be the global stabilization time. An execution of a protocol is considered synchronous if each message sent from a correct replica p_i is delivered by replica p_j within Δ. An execution of a protocol is considered asynchronous if there exists no time bound Δ for message delivery. An execution of a protocol is said to be partially synchronous, if there is an unknown GST such that once GST is reached, each message sent by process p_i is delivered by process p_j within a known Δ [17].



(a) Architecture **(b)** Protocol Layering

Figure 1. SADL-RACS Architecture and Protocol layering

Due to the FLP impossibility result [18], any deterministic algorithm cannot solve consensus under asynchrony even under a single replica failure. In RACS, we circumvent the FLP impossibility result using randomization. In a typical wide-area network, there are periods in which the network behaves synchronously, followed by phases where the network shows asynchronous behaviour. RACS makes use of this network behaviour and operates in two modes: (1) synchronous mode and (2) asynchronous mode. During the synchronous periods, RACS employs a leader-based design to reach consensus using one round trip network delay, and during the asynchronous mode, RACS employs randomization. SADL assumes only an asynchronous network.

2.2 Consensus and SMR

Consensus enables a set of replicas to reach an agreement on a single value. A correct consensus algorithm satisfies four properties [11]: (1) *validity*: the agreed upon value should be previously proposed by a replica, (2) *termination*: every correct process eventually decides some value, (3) *integrity*: no process decides twice, and (4) *agreement*: no two correct processes decide differently.

SMR employs multiple instances of the consensus algorithm to agree on a series of values [11] [4]. In this paper, we solve the SMR problem. A correct SMR algorithm satisfies two properties; (1) *safety*: no two replicas commit different client commands for the same log position and (2) *liveness*: each client command is eventually committed. We assume that each client command will be repeatedly proposed by replicas until it is committed.

3 Design Overview

3.1 SADL-RACS Architecture

Figure 1a illustrates SADL-RACS's architecture, containing 2 types of nodes: (1) replicas and (2) front ends. A front end is a node that receives commands from clients scattered elsewhere on the Internet. Front-end node batches commands into client batches and submits them to replicas. Replicas, upon receiving client batches from the front end, form replica batches, and then replicate them in a majority of replicas. Finally, the state machine in each replica executes the totally-ordered commands, and responds to the front end, with the response batches, which are eventually forwarded back to the clients by the front ends. In our discussion, we assume a static set of replicas, however, reconfiguration may be supported via standard practices [43]. In most of our discussions, we omit the details of the font end nodes, and focus on the replicas.

3.2 Protocol Layering

Figure 1b illustrates the protocol layering of SADL-RACS, for one replica. SADL-RACS consists of 5 layers: the asynchronous network layer, proxy layer, SADL layer, RACS layer, and the application layer. The proxy layer receives client batches from the asynchronous network layer (originating from the front-end nodes). Upon receiving a batch of client batches, the proxy layer forms a replica batch (of size usually more than 100KB) and requests the SADL layer to reliably replicate the replica batch, among at least a majority of the replicas. SADL layer, upon reliably disseminating the replica batch, requests the RACS layer to totally order the "meta-data" of the replica batch, which are usually of size several bytes. The RACS layer then runs the consensus protocol to make a total order of replica batch meta-data. Upon reaching consensus, the RACS layer delivers the totally ordered log to the application layer, and the application layer executes the commands in the replica batch. In our implementation, we use Redis[37] and a $map[string]string$ key value store as the applications.

As we empirically show in Section 7, having SADL in the protocol stack reduces the leader bottleneck in the RACS layer, hence, delivers high throughout, and supports high replication factor. Moreover, we show in Section 7 that, under adversarial network conditions, SADL helps RACS to preserve throughput.

However, under low load, and low replication factor, SADL causes additional latency overhead , hence, may become a burden for the class of applications that require a low replication factor, moderate throughput and low latency. To accommodate such applications, our design allows the proxy layer to directly request the RACS layer (hence bypassing the SADL layer) to totally order the replica batches, as we show in Fig. 1b. When the proxy layer directly invokes RACS, RACS carries the entire replica batch in the critical path of consensus.

3.3 Robust SMR in the WAN

Given the above operational overview of SADL-RACS, in the following, we summarize how SADL-RACS handles the two robustness challenges we mentioned in Section 1.

Challenge 1: Liveness under unreliable WAN network conditions; Under synchronous network conditions, RACS reaches consensus in one round trip, similar to Raft[43], however, under adversarial network conditions, RACS falls back to a randomized path that keeps committing new requests. Hence RACS provides liveness both under synchronous and adversarial network conditions. SADL does not rely on timeouts for progress, hence provides liveness under adversarial

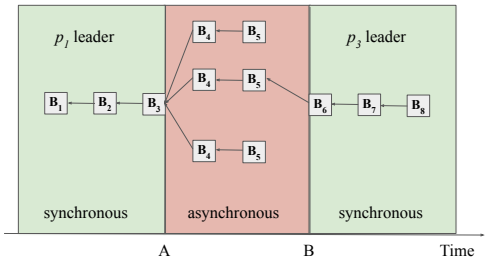


Figure 2. An execution of RACS from a replicated log perspective

network conditions. Hence, SADL-RACS combination provides resilience to unreliable WAN network conditions.

Challenge 2: Supporting higher fault tolerant threshold (f) without sacrificing throughput; SADL reliably disseminates replica batches (usually of size 100KB) asynchronously, and without relying on a leader replica. With SADL in place, RACS only totally orders meta-data of replica batches, which are usually of size several bytes. Hence, the overhead imposed on the RACS leader is significantly reduced. The low overhead imposed on the RACS leader allows SADL-RACS to scale to a higher number of replicas, without sacrificing the throughput, thus, enabling higher fault tolerant threshold (f).

In Section 4 we discuss RACS design and in the Section 5, we explain the SADL protocol. In Section 5.5 we propose a hybrid protocol that combines SADL and pipelining, that achieves the optimum performance under all workload and adversarial network conditions.

4 RACS Design

We propose RACS, a novel crash fault-tolerant consensus algorithm that guarantees liveness under adversarial network conditions. RACS employs the chaining approach to SMR similar to Raft [43], in which each new proposal to SMR has a reference to the previous proposal, and each commit operation commits the entire uncommitted history of client commands.

4.1 RACS Protocol Overview

RACS has two modes of operation; the synchronous leader-based mode that commits client commands in a single network round trip and a randomized mode that commits client commands under adversarial network conditions. RACS dynamically switches between the synchronous and the asynchronous modes depending on the network condition, automatically.

Figure 2 features an example execution of RACS going from synchronous mode to asynchronous mode, and then to synchronous mode again. After committing 3 blocks, B_1, B_2 and B_3 in the synchronous leader-based mode, with p_1 as the synchronous leader, all replicas go to the asynchronous randomized path (due to an adversarial network condition). There, all of the alive replicas, namely p_1, p_2 and p_4 propose **exactly** two asynchronous blocks: level 1 block B_4 then a level 2 block B_5. Then, a randomization process called common-coin-flip (see Section 4.2) designates p_2 as the elected leader of this asynchronous path. Hence, all replicas commit B_5 proposed by p_2 (and the causal history) and in the following synchronous mode execution commits B_6, B_7 and B_8, with p_3 as the synchronous mode leader.

4.2 Terminology

View number and **Round number**: RACS progresses as a sequence of views v and rounds r where each view has one or more rounds. A view represents the term of a leader while a round represents the successive log positions in the replicated log. The pair (v, r) is called a **rank**.

Block Format: There are two kinds of RACS blocks: (1) **synchronous blocks** and (2) **asynchronous blocks**. Both types of blocks consist of five elements: (1) batch of client commands, (2) view number, (3) round number, (4) parent link to a block with a lower rank, and (5) level. The rank of a block is (v, r) and blocks are compared lexicographically by their rank: first by the view number, then by the round number. The blocks are connected in a chain using the parent links. We denote that block A **extends** block B if there exists a set of blocks $b_1, b_2, b_3, ..b_k$ such that there exists a parent link from b_i to b_{i-1} $\forall$ i in $range(2, k)$ and $b_1 = B$ and $b_k = A$. The level element of the block refers to the asynchronous level (can take either the value 1 or 2). For the synchronous blocks, the level is always -1.

Common-coin-flip(v): We use a common-coin-flip primitive as a building block. For each view v, common-coin-flip(v) returns a positive integer in the range $(0, n-1)$ where n is the total number of replicas. The common-coin-flip(v) satisfies two properties; (1) for each v, the invocation of common-coin-flip(v) at each replica should return the same integer value and (2) output of the invocation of common-coin-flip(i) should be independent of the output of common-coin-flip(j) for $j \neq i$.

For the implementation of the common-coin-flip(v), we use the approach used by Rabia[44]: we use a pseudo-random number generator with the same seed (secretly shared at bootstrap) at each replica and pre-generate random numbers for each view number.

4.3 RACS Algorithm

Algorithm 1 depicts the pseudo-code of RACS. In the following discussion, when we say replica p_i delivers a block B from replica p_j, we imply that replica p_i delivers B and the causal history of B.

Synchronous mode: The synchronous mode of RACS is a leader-based consensus algorithm, as depicted in figure 3. The synchronous mode leader L_v for each view[1] v is predetermined and known to all replicas on bootstrap.

[1]the synchronous mode leader for view v (L_v) and the leader from the common-coin-flip(v) are different

Algorithm 1 RACS Protocol for replica p_i, $i \in 0..n-1$

1: $\mathbf{L_v} \leftarrow$ designated synchronous leader of view **v** (always exists)
2: each network link p_i-p_j is first-in-first-out ($j \in 0..n-1$)
3: **Local State:**
4: $v_{cur} \leftarrow$ the current view number
5: $r_{cur} \leftarrow$ the current round number
6: $block_{high} \leftarrow$ block with the highest rank received
7: $block_{commit} \leftarrow$ the last committed block (all blocks in the path from the $block_{commit}$ to the genesis block are considered committed blocks)
8: isAsync $\leftarrow$ **false**: the current protocol mode (boolean)
9: $B_{fall}[] \leftarrow$ all **null**, keeps the level 2 asynchronous blocks received from each node in the most recent view change
10: $argmaxrank(\mathcal{S})$ returns the block with the highest rank in the messages in $\mathcal{S}$

// initiate a new view and synchronous mode
11: **Upon** receiving a set $\mathcal{S}$ of $n-f$ <new-view, v, B> with the same v **and** v = v_{cur} **and** isAsync=**false**
12: $block_{high} \leftarrow argmaxrank(\mathcal{S})$

// leader initaites a new block
13: **Upon** a new batch of client commands are ready to be proposed **and** isAsync=**false**
14: **if** $L_{v_{cur}} == p_i$ **then**
15: cmnds $\leftarrow$ getClientCommands()
16: B$\leftarrow$(cmnds, v_{cur}, r_{cur}+1, $block_{high}$)
17: v_{cur}, $r_{cur} \leftarrow$ B.v, B.r
18: $block_{high} \leftarrow$ B
19: broadcast <propose, B, $block_{commit}$>
20: send <vote, v_{cur}, r_{cur}, $block_{high}$> to p_i // self voting
21: **end if**

// voting on a proposal from leader
22: **Upon** receiving <propose, B, $block_c$> such that B.rank > (v_{cur}, r_{cur}) **and** isAsync=**false**
23: cancel timer()
24: v_{cur}, $r_{cur} \leftarrow$ B.v, B.r
25: $block_{high} \leftarrow$ B
26: $block_{commit} \leftarrow block_c$
27: send <vote, v_{cur}, r_{cur}, $block_{high}$> to $L_{v_{cur}}$
28: start timer()

// the leader receives the votes and commits
29: **Upon** receiving a set $\mathcal{S}$ of $n-f$ <vote, v, r, B> with the same B **and** the same (v, r) **and** rank(B)=(v,r) **and** v = v_{cur} **and** rank(B) > rank($block_{commit}$) **and** isAsync=**false**
30: $block_{commit} \leftarrow$ B // commit block

// local timeout to launch asynchronous path
31: **Upon** local timeout expiration
32: broadcast <timeout, v_{cur}, r_{cur}, $block_{high}$>

// start asynchronous mode, all replicas act as leaders and propose a new asynchronous block
33: **Upon first** receiving a set $\mathcal{S}$ of $n-f$ <timeout, v, r, B> messages with the same v such that v $\geq$ (v_{cur}) **and** isAsync = **false**
34: isAsync $\leftarrow$ **true**
35: $block_{high} \leftarrow argmaxrank(\mathcal{S})$
36: v_{cur}, $r_{cur} \leftarrow$ v, max(r_{cur}, $block_{high}$.r)
37: cmnds $\leftarrow$ getClientCommands()
38: $B_{f1} \leftarrow$(cmnds, v_{cur}, r_{cur}+1, $block_{high}$, 1) // form a new level 1 asynchronous block
39: broadcast <propose-async, B_{f1},p_i, 1>

// voting on asynchronous proposals
40: **Upon** receiving <propose-async, B, p_j, h> from p_j **and** B.v == v_{cur} **and** isAsync == **true**
41: **if** rank(B) > (v_{cur}, r_{cur}) **then**
42: send <vote-async, B, h> to p_j
43: **if** h == 2 **then**
44: $B_{fall}[p_j] \leftarrow$ B
45: **end if**
46: **end if**

// receive votes and transmit level 2 asynchronous block
47: **Upon first** receiving $n-f$ <vote-async, B, h> **and** isAsync = **true and** B.v == v_{cur}
48: **if** h == 1 **then**
49: cmnds $\leftarrow$ getClientCommands()
50: $B_{f2} \leftarrow$(cmnds, v_{cur}, B.r+1, B, 2) // form a new level 2 asynchronous block
51: broadcast <propose-async, B_{f2}, p_i, 2>
52: **end if**
53: **if** h == 2 **then**
54: broadcast <asynchronous-complete, B, v_{cur}, p_i>
55: **end if**

// common coin flip and commit block
56: **Upon first** receiving a set $\mathcal{S}$ of $n-f$ <asynchronous-complete, B, v, p_j> **and** isAsync = **true and** v == v_{cur}
57: leader $\leftarrow$ common-coin-flip(v_{cur})
58: **if** level 2 block by leader exists in $\mathcal{S}$ **then**
59: $block_{high}$, $block_{commit} \leftarrow$ level 2 block from leader
60: v_{cur}, $r_{cur} \leftarrow$ rank($block_{high}$)
61: **else if** B_f[leader] != null **then**
62: $block_{high} \leftarrow B_f$[leader]
63: v_{cur}, $r_{cur} \leftarrow$ rank($block_{high}$)
64: **end if**
65: $v_{cur} \leftarrow v_{cur}$+1
66: isAsync $\leftarrow$ **false**
67: send <new-view, v_{cur}, $block_{high}$> to $L_{v_{cur}}$
68: start timer()

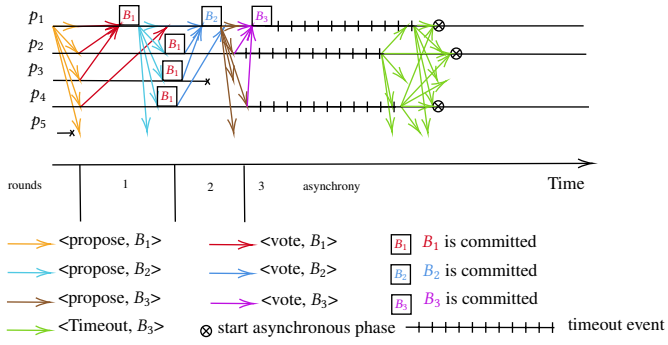


Figure 3. An example execution of the synchronous path of RACS with 5 replicas

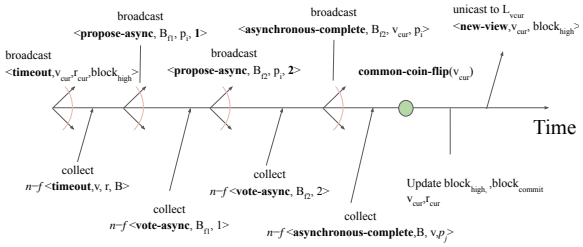


Figure 4. An execution of the asynchronous mode of RACS from the perspective of a single process

The synchronous mode begins either at the very start of the replicated log or after an asynchronous mode has ended, i.e. upon receiving a majority of <new-view> messages. Upon collecting a sufficient number of client commands, as permitted by the maximum batch size (see line 13), the leader replica forms a new block B, that extends the $block_{high}$. The leader then broadcasts a <propose> message for the block B containing a rank (v, r) and the reference of the last committed block $block_{commit}$ (line 19).

Each replica p_i delivers the <propose, B,$block_c$> message, if the rank of B is greater than the rank of p_i and if p_i is in the synchronous mode of operation (see line 22). If these two conditions are met, then p_i commits the block $block_c$ ($block_c$ has a lower rank than B) (see line 26) and sends p_i's <vote> for B to the leader replica (see line 27). Upon receiving $n - f$ <vote> messages for B (see line 29), the leader replica commits B (and the causal history).

Each replica has a timeout clock which is reset whenever the replica receives a new <propose> message (line 23). If the timeout expires (line 31), however, they will broadcast a <timeout> message containing the $block_{high}$ (see line 32).

Asynchronous mode: Upon receiving $n - f$ <timeout> messages, RACS enters the asynchronous mode of operation as depicted in figure 4 (see line 33). In the asynchronous mode, all replicas act as leaders, concurrently. Each replica takes the highest $block_{high}$ they are aware of, forms a level 1 asynchronous block B_{f1} with a monotonically increasing rank compared to the highest $block_{high}$ it received and sends a <propose-async> message (line 33-39).

Upon receiving a <propose-async> message from p_j, each replica p_i sends back a <vote-async> message to p_j **if** the rank of the proposed level 1 block is greater than the highest-ranked block witnessed so far (line 40-42).

Upon receiving $n - f$ <vote-async> messages for the level 1 asynchronous block B_{f1}, each replica will send a level 2 asynchronous fallback block B_{f2} (line 50-51). The algorithm allows catching up to a higher ranked block by building upon another replica's level 1 block. This is meant to ensure liveness for replicas that fall behind. All replicas, upon receiving a <propose-async> message for a level 2 asynchronous block from p_j send a <vote-async> message to p_j (line 40-42).

Once $n-f$ <vote-async>s have been gathered for the level 2 asynchronous block B_{f2}, the asynchronous path is considered complete and each replica p_i broadcasts an <asynchronous-complete> message (see line 54). When $n - f$ replicas have broadcast <asynchronous-complete> messages, all replicas exit the asynchronous path of RACS, by flipping a common-coin (see line 57) and committing the level 2 block (and the causal history) from the replica designated by the common-coin-flip(v). Because of the potential f faults, each replica commits a level 2 asynchronous block B if B arrived amongst the first $n - f$ <asynchronous-complete> blocks. After that, replicas exit the asynchronous path and resume the synchronous path by uni-casting a <new-view> message to the synchronous leader of the next view with the $block_{high}$.

4.4 Correctness and Complexity

We now give the proof intuitions of RACS. The formal proofs have been deferred to Appendix C.

Theorem 4.1. *RACS satisfies the safety property of SMR by ensuring that if a block B is committed in any round r, then all the blocks with round $r' \geq r$ will extend B.*

Proof. A block B can be committed in 2 instances; (1) synchronous mode and (2) asynchronous mode.

Synchronous mode: In the synchronous mode, a block is committed by the leader replica when a majority of nodes vote for the synchronous block in the same round. Hence, if the leader commits a block B, then it is guaranteed that at least a majority of the replicas have B as their $block_{high}$, or have B in the causal history of $block_{high}$. Due to quorum intersection, all the future blocks will extend B.

Asynchronous mode: In the asynchronous mode, the only asynchronous block B of level 2 sent by the elected asynchronous mode leader (through the common coin flip) is committed if it is among the first $n - f$ <asynchronous-complete> messages received in that view, which implies that a majority of the replicas have received B. Hence if at least one node commits B in the asynchronous path, then it is guaranteed

that at least a majority of the nodes set B as $block_{high}$, thus extending B in the next round. □

Theorem 4.2. *RACS ensures liveness under asynchronous network conditions.*

Proof. During the synchronous mode, commands get committed at each round by the synchronous mode leader, after the leader receives votes from a quorum of replicas.

In the asynchronous mode, if more than f replicas enter the asynchronous mode, i.e. there are less than $n - f$ replicas on the synchronous mode, i.e. the synchronous path cannot progress anymore, therefore all replicas eventually enter the asynchronous mode. The asynchronous mode (with at least $n - f$ correct replicas in it) will eventually reach a point where $n - f$ correct replicas have sent <asynchronous-complete> messages. The common-coin-flip(v) will therefore take place. If the result of the common-coin-flip(v) lands on one of the first $n - f$ replicas to have submitted an <asynchronous-complete> message, then a new block is committed. Since the coin is unbiased, there is a probability $p > \frac{1}{2}$ that a new block is committed in a given asynchronous execution. Each replica therefore commits a new block in under 2 views in expectation and commits eventually with probability 1, thus ensuring the protocol's liveness. □

Complexity The synchronous mode of RACS has a linear message and bit complexity for committing a block. The asynchronous mode of RACS has a complexity of $O(n^2)$.

Having discussed RACS that enables robust consensus over adversarial network conditions, we next present in Section 5 the design of SADL, which enables higher replication factor, without sacrificing the throughput.

5 SADL Design

SADL is a consensus-agnostic command dissemination layer whose goal is to reliably disseminate client commands. SADL does not solve consensus; rather, it ensures that a majority of the replicas are aware of each client command batch that is later proposed in the consensus by RACS. The RACS layer does not have to wait for client command dissemination because the SADL is executed ahead of time and concurrently. The RACS layer refers to a chain of SADL command batches using a fixed-sized integer array, making the communication lighter for RACS.

5.1 SADL Overview

The SADL algorithm runs asynchronously. In SADL, each replica acts as a leader, and disseminates the set of client commands it receives from the front-end servers among a majority of replicas. All replicas run the same algorithmic steps concurrently and do not need to wait for the progress of another replica to move on. This characteristic is what allows the SADL to yield a much higher throughput than RACS.

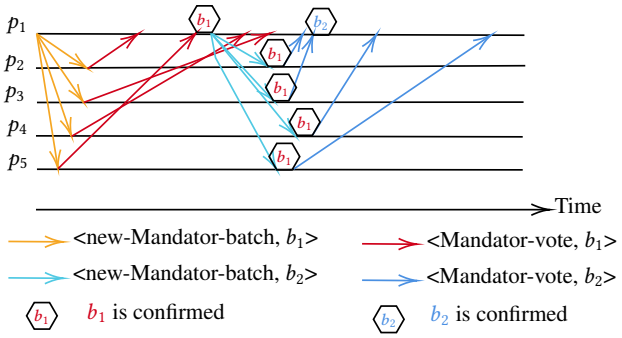


Figure 5. An execution of SADL with 5 replicas. All replicas act as leaders, simultaneously. For clarity we only show the execution with only p_1 as the leader

5.2 Terminology and Interface

We introduce **SADL-batch**. A SADL-batch contains four fields: (1) round number, (2) reference to the parent SADL-batch, (3) one or more client commands, and (4) the unique identifier. SADL provides the following generic interface.

- **replicate**(B): strongly replicate a new SADL-batch. We say that the replicate(B) is successful if the replica that replicates B receives at least $n - f$ <SADL-votes> for B.
- **fetch**(B): fetch the SADL-batch corresponding to the identifier of B. A fetch is successful if it returns a SADL-batch B which was successfully replicated previously using replicate(B)
- **fetch_causal**(B): fetch the set of SADL-batches that causally precede B. fetch_causal(B) succeeds if fetch(B) succeeds and if all the SADL-batches that were replicated before fetch(B) with which B has a causal dependency are in the returned SADL-batch list.

The SADL algorithm has two properties; (1) **Availability**: if fetch(B) is invoked after completing replicate(B), then fetch(B) eventually returns B and (2) **Causality**: a successful fetch_causal(B) returns all the SADL-batches with which B has a causal dependency.

5.3 SADL Algorithm

We explain SADL algorithm using Fig. 5. In Fig. 5, we consider a 5 replica setup with p_1 as the sender (in SADL all replicas act as senders, however, for clarity, we only show the execution with only p_1 as the leader). p_1 has received a batch of commands from front-end servers (line 6) and broadcast them as a <new-SADL-batch> B_1 with round number r_1 (line 12). All replicas eventually receive B_1 (line 13) and send back a <SADL-vote> to p_1 (line 16). Here p_3 and p_4 are slow and send their vote the latest. p_1 receives p_2's <SADL-vote> first, then p_5's. The <SADL-votes> from p_1 (self voting), p_2 and p_5 represent $n - f = 5 - 2 = 3$ votes, so p_1 can consider the round r_1 completed (line 19).

Replica p_1 receives another batch of commands from the front ends and broadcasts a <new-SADL-batch> B_2 to all

Algorithm 2 SADL Algorithm for process p_i, $i \in 0..n-1$

1: **Local State:**
2: lastCompletedRounds[], an integer array of n elements (with n the number of replicas) that keeps track of the last SADL-batch for which at least $n - f$ SADL-votes were collected for each replica
3: chains[][], a 2D array that saves the i^{th} SADL-batch created by replica p_j
4: buffer, a queue storing incoming client commands
5: awaitingAcks ← **false**, a boolean variable which states whether this replica is waiting for SADL-votes

Require: maximum batch time and batch size

6: **Upon** receiving a batch of client commands cl:
7: push cl to buffer

8: **Upon** (size of incoming buffer reaching batch size **or** maximum batch time is passed) **and** awaitingAcks is **false**:
9: B_{parent} ← the SADL-batch corresponding to chains[p_i][lastCompletedRounds[p_i]] (i.e. B_{parent} is the last SADL-batch proposed by p_i that received $n - f$ SADL-votes)
10: B ← (lastCompletedRounds[p_i]+1, B_{parent}, buffer.popAll()) // create new SADL-batch
11: isAwaiting ← **true**
12: broadcast <new-SADL-Batch, B>

13: **Upon** receiving <new-SADL-batch, B> from p_j
14: chains[p_j][B.round] ← B
15: lastCompletedRounds[p_j] ← B.parent.round
16: send <SADL-vote, B.round> to p_j

17: **Upon** receiving $n - f$ <SADL-vote, r> for the same r **and** r = lastCompletedRounds[p_i]+1 **and** awaitingAcks is **true**
18: awaitingAcks ← **false**
19: lastCompletedRounds[p_i]+=1

20: **procedure** getClientCommands()
21: **return** lastCompletedRounds

replicas (line 12). B_2 contains the lastCompletedRound[p_1] which indicates that B_1 was completed. Upon receiving B_2, p_2, p_3, p_4 and p_5 learn that B_1 was completed (has received a majority of votes) and update their own $lastCompletedRounds[p_1]$ to B_1 (line 15). p_2, p_3, p_4 and p_5 send a <SADL-vote> for B_2 to p_1 (line 16) and the algorithm continues. Appendix A.1 provides the proofs of SADL.

5.4 Using SADL with RACS

SADL provides the getClientCommands() interface to the RACS layer. getClientCommands() returns the lastCompletedRounds (see line 21) which contains the last completed SADL-Batch for each replica. Due to the availability and the causality properties of SADL, each SADL batch (and its causal history) indexed by the $lastCompletedRounds[p_i][j]$ is guaranteed to be available in at least a majority of the replicas. Hence, RACS proposes lastCompletedRounds for agreement among replicas. Since the lastCompletedRounds is a fixed-sized integer array of n elements, the consensus blocks become lightweight. In contrast, in monolithic protocols such as Multi-Paxos, the consensus messages carry the entire batch of client commands, thus sacrificing the throughput.

We evaluated the performance of RACS, under two configurations; (1) RACS with SADL and (2) RACS with pipelining [2]. With SADL enabled, RACS only agrees on the lastCompletedRounds, in contrast, with pipelining enabled, RACS agrees on individual command batches. As shown in Fig. 6b, we observe that for arrival rates less than 200k cmd/sec, in a 5-replica deployment, the median latency of pipelined-RACS is below 300ms, in contrast, SADL-RACS has 450ms median latency. We also observe that pipelined-RACS cannot support arrival rates greater than 250k cmd/sec, in contrast, SADL-RACS delivers stable throughput up to 500k cmd/sec. Hence, we derive that the throughput of SADL-RACS is optimal for higher arrival rates, in contrast, for lower arrival rates, the latency of pipelined-RACS is optimal. Can we achieve the best of both worlds, and have optimal throughput and latency for all arrival rates? Section 5.5 addresses this, by proposing a hybrid pipelined-SADL architecture.

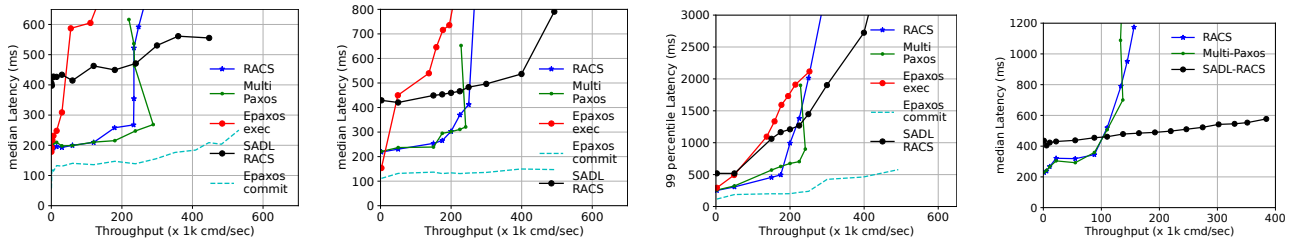
5.5 Hybrid SADL-pipelining protocol

The hybrid SADL-pipelining protocol involves two steps: (1) a calibration phase and (2) a deployment phase. In the calibration phase, the system administrator first deploys SADL-RACS and pipelined-RACS protocols, separately, in the given replica and front-end setup, and obtains the throughput versus median latency relationship, which we refer to as the *performance table*.

Then, in the hybrid SADL-pipelining deployment, the RACS layer first starts to replicate command batches using pipelining (without SADL) and the synchronous path leader L_v of RACS monitors the throughput and the median latency. When the median latency reaches the saturation median latency of RACS according to the *performance table*, the RACS layer automatically proposes a reconfiguration command to enable the SADL instead of pipelining. To consistently enable the reconfiguration across all the participating replicas, we use a similar method to Raft replica set reconfiguration (see [43]: section 6 Cluster membership changes). Once the reconfiguration takes effect, all the replicas switch to SADL. Similarly, if the throughput drops beyond a threshold w.r.t the *performance table*, the RACS leader automatically proposes a reconfiguration to switch back to pipelining.

With hybrid SADL-pipelining enabled, SADL-RACS delivers the optimal throughput and latency, for all arrival rates.

[2] pipelining is an existing technique used with Multi-Paxos, where the leader replica sends block B_i, before getting the $n - f$ votes for the block B_{i-1}



(a) Median Latency 3 replicas **(b)** Median Latency 5 replicas **(c)** 99% Latency 5 replicas **(d)** Median Latency 11 replicas

Figure 6. Throughput versus latency for WAN normal-case execution, comparing pipelined RACS and SADL-RACS to pipelined Multi-Paxos, and pipelined EPaxos, with 3, 5 and 11 replica ensembles

6 Implementation

We implemented RACS and SADL-RACS using Go version 1.18 [36], in 3661 and 4631 lines of codes, respectively, as counted by CLOC [16]. We use the Go network library and TCP [51] for reliable FIFO point-to-point links between replicas. We used Protobuf encoding [20] for encoding and decoding messages.

Both RACS and SADL-RACS implement batching in both front ends and replicas as in existing implementations of Rabia [45], Multi-Paxos, Raft[52], and EPaxos [39]. RACS implements pipelining, where more than one consensus instance can be activated, concurrently, before the previous instance is committed, an optimization available in Multi-Paxos and EPaxos. SADL-RACS does not implement pipelining.

Both RACS and SADL-RACS do not implement replica reconfiguration, but could be extended to do so by using a special consensus instance to agree on new configurations, by following the existing practices [26, 43].

We plan to make our source codes open source, upon acceptance.

7 Experimental evaluation

This evaluation demonstrates the following 4 claims.

- **C1**: Under synchronous network conditions RACS performs comparably to state-of-the-art leader-based algorithms.
- **C2**: RACS offers robustness against adversarial network conditions.
- **C3**: SADL-RACS offers robustness and preserves throughput under adversarial network conditions.
- **C4**: SADL improves the scalability of RACS in two dimensions: (1) **C4.1**: scalability w.r.t increasing replica count and (2) **C4.2**: scalability w.r.t increasing payload size.

Since adversarial networks are much more common in the WAN than in the local-area network (LAN), we focus on the WAN deployments in our evaluation, however, for completeness of experiments, we also compared the performance of RACS in a LAN, see Section 7.7 .

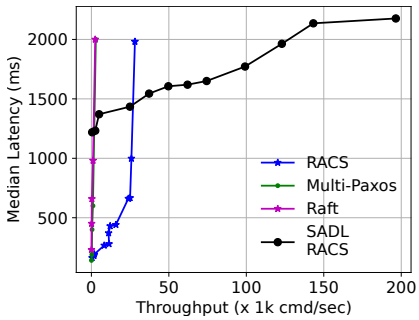


Figure 7. Adversarial Performance in the WAN with 5 replicas – note that Multi-Paxos and Raft lines overlap

We compare RACS's and SADL-RACS's performance against four state-of-the-art SMR algorithms: Raft [43] (existing implementation [52]), Multi-Paxos [26] (existing implementation [52]), Rabia [44] (existing implementation [45]), and EPaxos [38] ((existing implementation [39])). Similar to RACS, Raft is a leader-based chain-replication algorithm based on view-stamped replication [42]. Multi-Paxos is a leader-based protocol that runs the consensus protocol one instance at a time. Rabia is a randomized protocol, that specializes in low latency data center context. EPaxos is a multi-leader protocol that enables parallel commits of non-interfering commands.

7.1 Setup

We test both a WAN setup where the replicas and front-ends are distributed globally across AWS regions Sydney, Tokyo, Seoul, Osaka, and Singapore and a LAN setup where all replicas and front-ends are located in North California.

We use Amazon EC2 virtual machines [3] of type t2.xlarge (4 virtual CPUs, 16 GB memory) for replicas and front-ends, for WAN experiments. For LAN experiments, we use instances of type c4.4xlarge (16 virtual CPUs, 30 GB memory) for replicas and front-ends. We use Ubuntu Linux 20.04.5 LTS [56].

7.2 Benchmarks and Workloads

Following the existing implementations of Rabia [45], Multi-Paxos, and Raft [52], we use a $map[string]string$ key-value store and Redis[37] as backend applications.

In our experiments, we have n replicas and n front ends. Front-ends generate client requests with a Poisson distribution in the open-loop model [49]. All algorithms employ batching in both front-ends and replicas. EPaxos, Multi-Paxos, and RACS support pipelining, while Raft, SADL-RACS, and Rabia implementations do not. A single client request is a 17 bytes string: 1-byte GET/PUT opcode plus 8-byte keys and values, consistent with request sizes used in prior research and production systems [9, 44].

For RACS, SADL-RACS, Multi-Paxos, Raft, and Rabia we measure the front-end observed end-to-end execution latency, which accounts for the latency overhead for total ordering and executing commands. EPaxos provides two modes of operations: (1) partial ordering of commands without execution (denoted "EPaxos-commit" in the graphs) and (2) partial ordering of commands with execution (denoted "EPaxos-exec" in the graphs). Trivially, "EPaxos-commit" outperforms RACS, SADL-RACS, Raft, and Multi-Paxos because EPaxos-commit only provides a partial order of commands, which enables higher parallelism. Hence, "EPaxos-commit" provides an apples-to-oranges comparison, however, we present the results in this evaluation, for completeness. We also found and reported bugs in the existing implementation of EPaxos code that prevent execution under attacks, crashes, and when deployed with more than 5 replicas. Hence we use EPaxos only under normal-case performance evaluation.

We run each experiment for one minute, repeating it 3 times. We measure throughput in commands per second (cmd/sec), where a command is one 17-byte request. We measure the latency in milliseconds.

7.3 RACS WAN Normal Case Performance

In this experiment, we evaluate the normal-case synchronous performance of RACS deployed in 5 geographically distant AWS regions. Fig. 6b and Fig. 6c depict the experimental results using 5 replicas and 5 front-ends.

RACS vs Multi-Paxos: We observe in Fig. 6b that RACS delivers a saturation throughput of 200k cmd/sec throughput under 300ms median latency, which is comparable to the performance of Multi-Paxos (200k cmd/sec under 300ms latency). In the synchronous execution both RACS and Multi-Paxos have 1 round trip latency per batch of commands, hence share the same performance characteristics. Hence the experimental claim **C1** holds.

RACS vs Epaxos commit: We observe in Fig. 6b that EPaxos-commit (without command execution) delivers a throughput of 500k+ cmd/sec under 170ms median latency. The EPaxos-commit experiment employs a conflict rate of 2%[55] hence 98% of the time, commands are committed in one round trip, without serializing through a leader replica. In contrast, RACS builds a total order of commands, serialized using a leader-replica, hence naturally the performance is bottlenecked by the leader replica's capacity.

RACS vs Epaxos exec: As shown in Fig. 6b, the median latency of EPaxos-exec (with command execution) is 300ms higher on average than RACS in the 50k–200k cmd/sec throughput range. This higher latency stems from EPaxos's dependency management cost [35, 55]. Hence, we conclude that when measured for execution latency, RACS outperforms EPaxos.

RACS vs Rabia: Finally, we observe that Rabia achieves less than 100 cmd/sec throughput under 1000ms median latency (hence not shown in Fig. 6). Rabia makes a design assumption that network delay is smaller than the interval between two consecutive requests (see section 3.2 of Rabia[44]), a condition that holds only in the LAN deployment. The Rabia paper acknowledges this limitation in their paper and claims performance only in a LAN setting. Section 7.7 evaluates Rabia against RACS in a LAN setting.

7.4 Asynchronous Performance

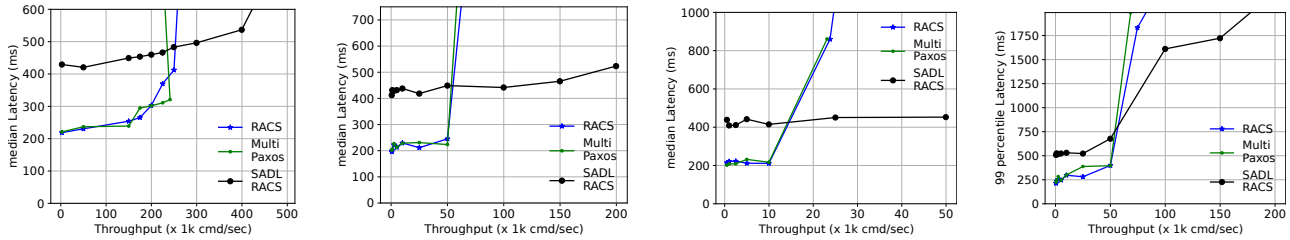
This experiment evaluates RACS and SADL-RACS under simulated network attacks, similar to attacks used in [50, 53]. Our simulated attacker increases the egress packet latency of a minority of replicas chosen at random, by 500ms, dynamically, in time epochs. In each time epoch i, the attacker randomly selects up to $n/2$ replicas (n the total number of replicas) and launches the attack. This experiment runs in the WAN setting with 5 replicas and 5 front ends. We depict the results in Figure 7. [3] [4]

RACS vs Multi-Paxos and Raft:We observe that RACS provides 28k cmd/sec saturation throughput, in contrast, Multi-Paxos and Raft have saturation throughput at 2.8k cmd/sec. Under adversarial network conditions Multi-Paxos and Raft undergo repeated view changes, and fail at successfully committing requests. In contrast, due to asynchronous liveness guarantees, RACS provides liveness under asynchrony. Hence we prove the claim **C2**.

SADL-RACS vs RACS: we observe that SADL-RACS delivers 196k saturation throughput, thus providing 168k cmd/sec more throughput than RACS. SADL disseminates client commands to a majority of replicas, asynchronously, and in parallel, without relying on a leader. Hence 3 out of 5 replicas in SADL disseminate requests at the speed of the network, in each time epoch, while the other two replicas progress slowly due to the attack, hence SADL-RACS preserves throughput to the maximum level the network conditions allow. This result proves our claim **C3**.

[3] we do not use Rabia in this experiment, given that Rabia only performs well in the LAN.

[4] we do not employ EPaxos in this experiment because of a dependency management bug we found in the EPaxos code under adversarial network conditions.



(a) Median Latency (17B) **(b)** Median Latency (73B) **(c)** Median Latency (265B) **(d)** 99% Latency (73B)

Figure 8. Throughput versus latency for WAN normal-case execution, comparing SADL-RACS to pipelined Multi-Paxos and pipelined RACS using 17B, 73B and 265B command sizes, using 5 replicas

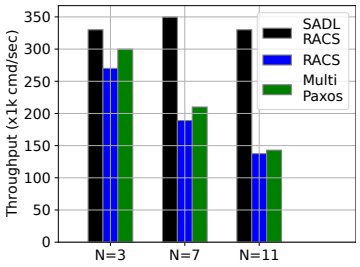


Figure 9. WAN scalability with Redis backend

7.5 Scalability of SADL

In this experiment, we aim to quantify the scalability of SADL-RACS. We consider two factors of scalability: (1) scalability w.r.t increasing replication factor and (2) scalability w.r.t increasing payload size.

Scalability w.r.t increasing replication factor: In this experiment, we evaluate the scalability of SADL-RACS by running it with an ensemble of three (minimum replication allowed), five (common replication factor) and eleven replicas (improved robustness to concurrent replica failures), located in geographically separated AWS regions. [5] [6] [7] Fig. 6 (and Fig. 11 in appendix) compare the scalability of SADL-RACS with pipelined RACS, pipelined Multi-Paxos, and pipelined EPaxos, for different replication factors.

Multi-Paxos and RACS: We observe that the saturation throughput of Multi-Paxos and RACS decreases from 230k to 130k cmd/sec when the replication factor is increased from 3–11. With increasing replica count, the leader replica in RACS and Multi-Paxos has to send and receive more messages, due to increased quorum sizes, hence the performance is bottlenecked by the leader's bandwidth capacity.

SADL-RACS vs RACS: We observe that SADL-RACS provides a throughput of at least 380k cmd/sec, when increasing the number of replicas from 3–11. SADL-RACS outperforms pipelined RACS and pipelined Multi-Paxos by 192% in the 11 replica scenario. This confirms that separating the command dissemination from the critical path of consensus can indeed improve the scalability. Hence we prove the claim **C4.1**.

We also evaluated the scalabliity of RACS and SADL-RACS using Redis data store as the backend (see Fig. 9), and reconfirmed the claim **C4.1**.

Scalability w.r.t increasing payload size: In this experiment, we evaluate the impact of payload size for the SADL-RACS performance. We experiment with 3 key sizes: 8B, 64B, and 256B, used in recent SMR work [2]. Combined with 1B opcode and 8B value, these key sizes result in 17B, 73B, and 265B command sizes. We deploy SADL-RACS, pipelined Paxos and pipelined RACS in a WAN setting with 5 replicas and 5 front ends. Fig. 8 (and Fig. 12 in appendix) depict the results. [8]

We observe that for each command size, the saturation throughput of SADL-RACS is at least 2 times the throughput of RACS and Multi-Paxos. With increasing command size, the leader replica's bandwidth of RACS and Multi-Paxos becomes the bottleneck. In contrast, thanks to the decoupling of command dissemination from consensus, SADL-RACS evenly distributes the bandwidth overhead among all the replicas, and sustains higher throughput. This proves our final claim **C4.2**.

7.6 Latency overhead of SADL

In Fig. 6 and Fig. 8, we observe that SADL-RACS provides better throughput than pipelined RACS, under all replica configurations and under all payload sizes. Moreover, in Section 7.4, we observed that SADL-RACS sustains higher throughput than pipelined RACS under adversarial network conditions.

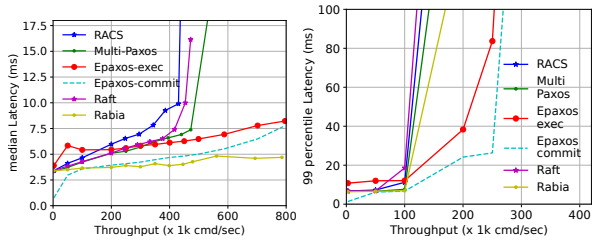
We also observe in Fig. 6, that for low arrival rates, the median latency of SADL-RACS is higher than the median

[5] Note that, unlike blockchain algorithms where consensus algorithms are measured for up to a hundred nodes[19], crash fault tolerant protocols are designed to scale up to 9–11 nodes in practice [25, 31]

[6] We did not use Rabia in this experiment, given that Rabia only performs well in the LAN

[7] We use EPaxos only in the 3 and 5 replica deployments, due to a dependency-checking bug, we found in the existing code of Epaxos[39] that appears when the number of replicas are greater than 5

[8] we do not employ EPaxos in this experiment because EPaxos implementation uses fixed sized fields for the payload



(a) Median Latency on LAN **(b)** 99% Latency on LAN

Figure 10. Throughput versus latency for LAN normal-case execution, comparing RACS to Rabia, Multi-Paxos, EPaxos, and Raft using 5 replicas

latency of pipelined-RACS, for instance, in Fig. 6b for arrival rates less than 200k cmd/sec, the median latency of pipelined-RACS is below 300ms, in contrast, SADL-RACS has 450ms median latency.

To achieve the best of both worlds: (1) high saturation throughput under normal and adversarial network conditions and (2) low latency under low arrival rates, we employ the hybrid SADL-pipelined protocol outlined in Section 5.5. Our evaluation enables the compilation of the "performance table" of the hybrid SADL-pipelined protocol in Section 5.5. We leave the implementation and evaluation of hybrid SADL-pipelining as future work.

7.7 RACS LAN Normal Case Performance

We designed RACS and SADL for the WAN, however, for the completeness of the evaluation, we also present the LAN performance. Fig. 10 depicts the experiment results, where we experiment with 5 replicas and 5 clients, deployed in the same AWS region.

RACS vs Multi-Paxos and Raft: We first observe that RACS achieves a saturation throughput of 420k cmd/sec, under a median latency upper bound of 10ms, that is comparable to the saturation throughput of Multi-Paxos (450k) and Raft (440k). Under normal case executions, RACS, Multi-Paxos, and Raft have 1 round-trip latency, serialized through a leader, hence provide comparable performance.

RACS vs EPaxos: Second, we observe that Epaxos outperforms RACS, both in terms of latency and throughput. We use similar reasoning as Section 7.3, to illustrate this behaviour.

RACS vs Rabia: Finally, we observe that Rabia delivers 800k cmd/sec throughput under 5ms median latency, outperforming RACS by 380k cmd/sec in throughput, and by 5ms in median latency. Rabia uses multiple leaders, thus avoids the leader bottleneck present in RACS. Moreover, Rabia relies on the "natural" ordering of messages inside a data centre, where it is guaranteed that a majority of the replicas receive the same message, sent at the same time. Rabia exploits this data-centre-specific "natural" message ordering and uses Ben-Or[7] to check whether the network ordering indeed has achieved consensus (or not). This makes Rabia messages lightweight because the consensus messages no longer have to carry the payload. In contrast, RACS, Multi-Paxos and Raft carry the request payload inside the consensus messages, thus incurring higher latency. Rabia's approach of using consensus to confirm the "natural" ordering inside a data centre can readily be combined with RACS, Multi-Paxos and Raft, however, falls outside the scope of this work.

8 Related Work

Leader based consensus: Multi-Paxos [26], Raft [43], and View Stamp Replication [42] use a single leader node to totally order requests. To reduce the computational and network bottlenecks at the single leader, Baxos [53], Mencius [31], EPaxos [38], Generalized Paxos [27] and Multi-Coordination Paxos [12] use multiple leaders, by partitioning the replicated log using state[27, 38], log position [31] and by using randomized backoff[53]. Leader based and the multi-leader protocols lose liveness under adversarial network conditions, in contrast, RACS maintains liveness under adversarial network conditions. OmniPaxos [40] solves consensus under partial network partitions, a contribution outside the scope of this paper. OmniPaxos loses liveness under asynchrony. In contrast, RACS maintain liveness under all adversarial network conditions.

Asynchronous consensus: Ben-Or[7] is a binary randomized consensus algorithm that provides liveness under asynchrony. Rabia [44] is the first practical multi-valued randomized protocol, that employs Ben-Or as its core. Rabia and RACS achieves orthogonal goals: Rabia provides simplified SMR design for the low latency data center context (hence providing little to no performance in the WAN, and under advasarial network conditions), whereas RACS aims at providing robustness against adversarial networks in both LAN and WAN. Turtle consensus [41] provides robustness to adversarial network conditions by switching between Paxos and Ben-Or, however, provides sub-optimal performance due to static mode switching between Paxos and Ben-Or, and also due to overhead of switching between two different protocols. QuePaxa[54] is a concurrent work to RACS, and solves asynchronous consensus problem using randomization. QuePaxa uses local randomization (local-coins), hence enables each replica to trigger the asynchronous path of the protocol, whereas RACS employs shared randomization (common-coins), such that a majority of replicas enter and exit the asynchronous path, simultaneously. Both QuePaxa and RACS have $O(n^2)$ worst case message complexity in the slow path, and $O(n)$ complexity in the fast path. QuePaxa has a low fault tolerant threshold (f) for a given level of throughput because of the leader replica bottleneck, in contrast, SADL-RACS enables higher f by decoupling request propagation from the critical path of consensus.

Request dissemination Sharding based protocols [32] [1] [15] achieve better performance by concurrently committing

transactions that touch different shards. Sharding is orthogonal to SADL-RACS's contributions, and can be readily integrated with SADL-RACS to achieve more concurrency. Overlay based protocols [8, 14, 21, 33, 48, 59, 62] improve the performance by delegating message propagation to non-leader nodes, however, achieve sub-optimal performance and resiliency against adversarial networks, compared to SADL.

Orthogonal goals and future work RACS and SADL focus on the robustness and performance of consensus, however, it does not address many other useful goals, which we believe will be interesting future work to explore: *e.g.*, reducing SADL message cost via erasure coding [57, 58], reducing the RACS leader's load by outsourcing work [61], reducing quorum size needed in the synchronous path [1, 23], and hardware optimizations[2]. RACS and SADL only focus on crash-failures, and we leave extending RACS to Byzantine replica failures [13] as interesting future work. RACS reconfiguration and crash-recovery requires securely sending the common-coin values to the new replica, and we leave it as future work.

9 Conclusion

We presented RACS and SADL, a modular wide-area SMR algorithm that achieves robustness under network asynchrony and supports higher replication factors without compromising the throughput. RACS achieves optimum one round-trip consensus in the synchronous network setting, as well as liveness in the face of network asynchrony. SADL allows higher replication factors without sacrificing the throughput. Our evaluation shows that SADL-RACS delivers 500k cmd/sec under 800ms in the wide-area, and out-performs Multi-Paxos by 150%, at a modest expense of latency while remaining live under adversarial network conditions and scaling up to 11 replicas, without sacrificing the throughput.

References

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

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

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

[4] Karolos Antoniadis, Rachid Guerraoui, Dahlia Malkhi, and Dragos-Adrian Seredinschi. State machine replication is more expensive than consensus. In *32nd International Symposium on Distributed Computing*, October 2018.

[5] James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175, 2003.

[6] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Conference on Innovative Data system Research*, pages 223–234, 2011.

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

[8] Martin Biely, Zarko Milosevic, Nuno Santos, and André Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 111–120, 2012.

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

[10] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *7th Symposium on Operating Systems Design and Implementation*, pages 335–350, 2006.

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

[12] Lásaro Jonas Camargos, Rodrigo Malta Schmidt, and Fernando Pedone. Multicoordinated paxos. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 316–317, 2007.

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

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

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

[16] Al Danial. Counting lines of code (CLOC). http://cloc.sourceforge.net/.

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

[18] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[19] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and Ditto: Network-adaptive efficient consensus with asynchronous fallback. In *26th International Conference on Financial Cryptography and Data Security: (FC)*, pages 296–315. Springer, May 2022.

[20] Google. Protocol buffers. https://developers.google.com/protocol-buffers/, 2020.

[21] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. Fastfabric: Scaling hyperledger fabric to 20 000 transactions per second. *International Journal of Network Management*, 30(5):e2099, 2020.

[22] Andrew Grimshaw, Mark Morgan, and Avinash Kalyanaraman. Gffs—the xsede global federated file system. *Parallel Processing Letters*, 23(02):1340005, 2013.

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

[24] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. {ZooKeeper}: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[25] Marios Kogias and Edouard Bugnion. HovercRaft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–17, April 2020.

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

[27] Leslie Lamport. Generalized consensus and Paxos. *Microsoft Research Technical Report MSR-TR-2005-33*, page 60, 2005.

[28] Tom Lianza and Chris Snook. Cloudflare outage. https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/, November 2020.

[29] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416, 2011.

[30] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Symposium on Operating Systems Design and Implementation OSDI*, volume 4, pages 8–8, 2004.

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

[32] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. Multi-ring paxos. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012.

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

[34] Ali José Mashtizadeh, Andrea Bittau, Yifeng Frank Huang, and David Mazieres. Replication, history, and grafting in the ori file system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 151–166, 2013.

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

[36] Jeff Meyerson. The Go programming language. *IEEE Software*, 31(5):104–104, 2014.

[37] Vladimir Mihailenco, Denissenko, and Dimitrij. Go lang Redis. https://github.com/redis/go-redis, 2023.

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

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

[40] Harald Ng, Seif Haridi, and Paris Carbone. Omni-Paxos: Breaking the barriers of partial connectivity. In *Eighteenth European Conference on Computer Systems (EuroSys)*, pages 314–330, May 2023.

[41] Stavros Nikolaou and Robbert Van Renesse. Turtle consensus: Moving target defense for consensus. In *Proceedings of the 16th Annual Middleware Conference*, pages 185–196, December 2015.

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

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

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

[45] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, Roberto Palmieri, and Pasindu Tennage. Rabia. https://github.com/dedis/quepaxa-rabia-open-loop, September 2023.

[46] Dino Quintero, Matteo Barzaghi, Randy Brewster, Wan Hee Kim, Steve Normann, Paulo Queiroz, Robert Simon, Andrei Vlad, et al. *Implementing the IBM General Parallel File System (GPFS) in a Cross Platform Environment*. IBM Redbooks, 2011.

[47] Eric Rescorla and Tim Dierks. The transport layer security (TLS) protocol version 1.3, August 2018. RFC 8446.

[48] Sajjad Rizvi, Bernard Wong, and Srinivasan Keshav. Canopus: A scalable and massively parallel consensus protocol. In *Proceedings of the 13th International Conference on Emerging Networking Experiments and Technologies*, pages 426–438, 2017.

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

[50] Alexander Spiegelman and Arik Rinberg. ACE: Abstract consensus encapsulation for liveness boosting of state machine replication. *International Conference on Principles of Distributed Systems, OPODIS*, December 2020.

[51] Transmission control protocol, September 1981. RFC 793.

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

[53] Pasindu Tennage, Cristina Basescu, Eleftherios Kokoris Kogias, Ewa Syta, Philipp Jovanovic, and Bryan Ford. Baxos: Backing off for robust and efficient consensus. *arXiv preprint arXiv:2204.10934*, April 2022.

[54] Pasindu Tennage, Cristina Basescu, Lefteris Kokoris-Kogias, Ewa Syta, Philipp Jovanovic, Vero Estrada, and Bryan Ford. QuePaxa: Escaping the tyranny of timeouts in consensus. *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, October 2023.

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

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

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

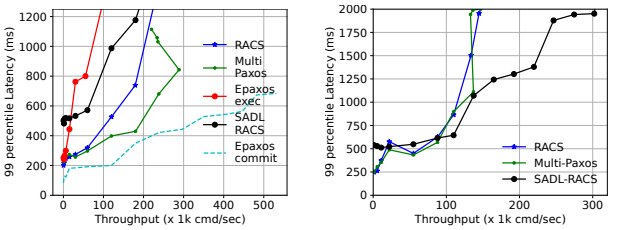
[58] Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shangming Cai, Zihan Xu, and Dongsheng Wang. CRaft: An Erasure-coding-supported version of Raft for reducing storage cost and network cost. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*,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

[59] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. Scaling replicated state machines with compartmentalization. *Proceedings of the VLDB Endowment*, 14(11):2203–2215, 2021.

[60] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining {ACID} and {BASE} in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 495–509, 2014.

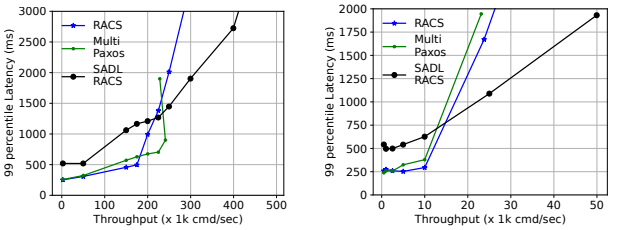
[61] Zichen Xu, Christopher Stewart, and Jiacheng Huang. Elastic, geo-distributed RAFT. In *Proceedings of the International Symposium on Quality of Service*. Association for Computing Machinery, 2019.

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



(a) 99% Latency 3 replicas **(b)** 99% Latency 11 replicas

Figure 11. Throughput versus tail latency for WAN normal-case execution, comparing pipelined RACS and SADL-RACS to pipelined Multi-Paxos and pipelined Epaxos with 3 and 11 replica ensembles



(a) 99 % Latency 17B **(b)** 99% Latency 265B

Figure 12. Throughput versus tail latency for WAN normal-case execution, comparing pipelined RACS and SADL-RACS to pipelined Multi-Paxos with 17B and 265B payload sizes

A SADL Appendix

A.1 Correctness and Complexity

Proof of Availability: A replicate(B) operation succeeds when B is created and sent to all the replicas, and only after receiving at least $n - f$ SADL-votes. Since each replica saves B in the *chains* array, it is guaranteed that B will persist as long as $n - f$ replicas are alive due to quorum intersection. Hence fetch(B) eventually returns.

Proof of Causality: Causality follows from the fact that each replica extends its chain of SADL-batches, and because each replica creates a batch with round r only after completing the replicate operation of the batch with round $r - 1$.

Complexity: The SADL algorithm has a linear complexity: for each batch of client commands, one SADL-batch is broadcast to all replicas and each of these replicas replies to the sender with a <SADL-vote>.

B Evaluation Supplementary

B.1 Scalability Tail Latency

Fig. 11 depicts the tail latency of RACS and SADL-RACS under different replica sizes.

Fig. 12 depicts the tail latency of RACS and SADL-RACS under different payload sizes.

C RACS Formal Proofs

C.1 Definition

elected-asynchronous block: We refer to an asynchronous block B_f generated in view v with level 2 as an elected-asynchronous block, if the common-coin-flip(v) returns the index of the proposer p_l who generated B_f in the view v and if the <asynchronous-complete> for B_f exists in the first $n-f$ <asynchronous-complete> messages received. An elected-asynchronous block is committed same as a synchronously-committed block.

C.2 Proof of safety

We first show that for a given rank (v,r), there exists a unique block. In the following lemmas C.1,C.2,C.3,C.4,C.5, and C.6 we consider different formations of blocks with the same rank.

Lemma C.1. *Let B, $\tilde{B}$ be two synchronous blocks with rank (v, r). Then B and $\tilde{B}$ are the same.*

Proof. Assume by contradiction B and $\tilde{B}$ to be different. Then, according to line 16, both B and $\tilde{B}$ were created by the leader of view v, L_v. Assume L_v created B first and then $\tilde{B}$. Then, by construction, $\tilde{B}$ has a rank greater than (v,r). Hence, a contradiction. Thus $B = \tilde{B}$. □

Lemma C.2. *Let B, $\tilde{B}$ be two elected-asynchronous blocks with rank (v, r). Then B and $\tilde{B}$ are the same.*

Proof. Assume by contradiction B and $\tilde{B}$ to be different. Then, according to line 57-59, both leaders who sent B and $\tilde{B}$ in an <asynchronous-complete> message were elected with the same common-coin-flip(v). Since no replica can equivocate, i.e. no replica sends <asynchronous-complete> message for two different blocks with the same rank (v, r), and because the common-coin-flip(v) returns a unique leader for each v, this is a contradiction. Thus $B = \tilde{B}$. □

Lemma C.3. *Let B, $\tilde{B}$ be two asynchronous blocks with rank (v, r) and level 1, such that both blocks are parents of an elected-asynchronous block of the same view v. Then B and $\tilde{B}$ are the same.*

Proof. Assume by contradiction B and $\tilde{B}$ to be different. Then both B and $\tilde{B}$ can have a distinct child level 2 elected asynchronous block with rank $(v, r + 1)$ (see line 50). According to lemma C.2, this is a contradiction. Thus $B = \tilde{B}$. □

Lemma C.4. *Let B be a synchronous block which receives $n-f$ <vote>s. Then there cannot exist a level 1 asynchronous block $\tilde{B}$ that is a parent of a level 2 elected-asynchronous block where B and $\tilde{B}$ have rank (v, r).*

Proof. Assume by way of contradiction that $\tilde{B}$ exists. Because B received $n - f$ votes, at least $n - f$ replicas saw B before $\tilde{B}$ (see line 22). $\tilde{B}$ has received $n - f$ <vote-async> (see line 47) from replicas who could not have seen B before (see line 41).

Because $n - f > \frac{n}{2}$, this is a contradiction. Hence $\tilde{B}$ does not exist. □

Lemma C.5. *Let B be a synchronous block which receives $n-f$ votes with rank (v, r). Then there cannot exist an elected-asynchronous block $\tilde{B}$ of level 2 with rank (v, r).*

Proof. Assume by way of contradiction that $\tilde{B}$ exists. Because B received $n - f$ votes, at least $n - f$ replicas saw B before $\tilde{B}$ (see line 22). $\tilde{B}$ has received $n - f$ <vote-async> (see line 47) from replicas who could not have seen B before (see line 41). Because $n - f > \frac{n}{2}$, this is a contradiction. Hence $\tilde{B}$ does not exist. □

Lemma C.6. *Let B be a level 1 asynchronous block that is the parent of level 2 elected-asynchronous block in the same view. Then there cannot exist a level 2 elected-asynchronous block $\tilde{B}$ with rank (v, r).*

Proof. Assume by way of contradiction that $\tilde{B}$ exists. The level 1 parent block of $\tilde{B}$ had rank $(v, r-1)$ (see line 50) and was created after receiving $n - f$ timeout messages with rank $(v, r-2)$ (see line 38). On the other hand, B was created after receiving $n-f$ timeout messages with rank $(v, r-1)$. Because $n - f > \frac{n}{2}$, this is a contradiction. Hence $\tilde{B}$ does not exist. □

Theorem C.7. *Let B and $\tilde{B}$ be two blocks with rank (v, r). Each of B and $\tilde{B}$ can be of type: (1) synchronous block which collects at least $n-f$ votes or (2) elected-asynchronous block or (3) level 1 asynchronous block which is a parent of an elected-asynchronous block. Then $\tilde{B}$ and B are the same.*

Proof. This holds directly from Lemma C.1, C.2, C.3, C.4, C.5 and C.6. □

Theorem C.8. *Let B and $\tilde{B}$ be two adjacent blocks, then $\tilde{B}.r = B.r + 1$ and $\tilde{B}.v \geq B.v$.*

Proof. According to the algorithm, there are three instances where a new block is created.

- Case 1: when *isAsync* = false and L_v creates a new synchronous block by extending the $block_{high}$ with rank (v,r)(see line 16). In this case, L_v creates a new block with round $r+1$. Hence the adjacent blocks have monotonically increasing round numbers.
- Case 2: when *isAsync* = true and upon collecting $n - f$ <timeout> messages in view v (see line 33). In this case, the replica selects the $block_{high}$ with the highest rank (v, r), and extends it by proposing a level 1 asynchronous block with round $r + 1$. Hence the adjacent blocks have monotonically increasing round numbers.
- Case 3: when *isAsync* = true and upon collecting $n - f$ <vote-async> messages for a level 1 asynchronous block (see line 47-48). In this case, the replica extends the level 1 block by proposing a level 2 block with round $r + 1$. Hence the adjacent blocks have monotonically increasing round numbers.

The view numbers are non decreasing according to the algorithm. Hence Theorem C.8 holds. □

Theorem C.9. *If a synchronous block B_c with rank (v, r) is committed, then all future blocks in view v will extend B_c.*

Proof. We prove this by contradiction.

Assume there is a committed block B_c with $B_c.r = r_c$ (hence all the blocks in the path from the genesis block to B_c are committed). Let block B_s with $B_s.r = r_s$ be the round r_s block such that B_s conflicts with B_c (B_s does not extend B_c). Without loss of generality, assume that $r_c < r_s$.

Let block B_f with $B_f.r = r_f$ be the first valid block formed in a round r_f such that $r_s \geq r_f > r_c$ and B_f is the first block from the path from genesis block to B_s that conflicts with B_c; for instance B_f could be B_s. L_v forms B_f by extending its $block_{high}$ (see line 16). Due to the minimality of B_f (B_f is the first block that conflicts with B_c), $block_{high}$ contain either B_c or a block that extends B_c. Since $block_{high}$ extends B_c, B_f extends B_c, thus we reach a contradiction. Hence no such B_f exists. Hence all the blocks created after B_c in the view v extend B_c. □

Theorem C.10. *If a synchronous block B with rank (v, r) is committed, an elected-asynchronous block $\tilde{B}$ of the same view v will extend that block.*

Proof. We prove this by contradiction. Assume that a synchronous block B is committed in view v and an elected-asynchronous block $\tilde{B}$ does not extend B. Then, the parent level 1 block of $\tilde{B}$, $\tilde{B_p}$, also does not extend B.

To form the level 1 $\tilde{B_p}$, the replica collects $n - f$ <timeout> messages (see line 33), each of them containing the $block_{high}$. If B is committed, by theorem C.9, at least $n - f$ replicas should have set (and possibly sent) B or a block extending B as the $block_{high}$. Hence by intersection of the quorums $\tilde{B_p}$ extends B, thus we reach a contradiction. □

Theorem C.11. *At most one level 2 asynchronous block from one proposer can be committed in a given view change.*

Proof. Assume by way of contradiction that 2 level 2 asynchronous blocks from two different proposers are committed in the same view. A level 2 asynchronous block B is committed in the asynchronous phase if the common-coin-flip(v) returns the proposer of B as the elected proposer (line 57). Since the common-coin-flip(v) outputs the same elected proposer across different replicas, this is a contradiction. Thus all level 2 asynchronous blocks committed during the same view are from the same proposer.

Assume now that the same proposer proposed two different level 2 asynchronous blocks. According to the line 50, and since no replica can equivocate, this is absurd.

Thus at most one level 2 asynchronous block from one proposer can be committed in a given view change. □

Theorem C.12. *Let B be a level 2 elected-asynchronous block that is committed, then all blocks proposed in the subsequent rounds extend B.*

Proof. We prove this by contradiction. Assume that level two elected-asynchronous block B is committed with rank (v, r) and block $\tilde{B}$ with rank $(\tilde{v}, \tilde{r})$ such that $(\tilde{v}, \tilde{r}) > (v, r)$ is the first block in the chain starting from B that does not extend B. $\tilde{B}$ can be formed in two occurrences: (1) $\tilde{B}$ is a synchronous block in the view $v + 1$ (see line 16) or (2) $\tilde{B}$ is a level 1 asynchronous block with a view strictly greater than v (see line 38). (we do not consider the case where $\tilde{B}$ is a level 2 elected-asynchronous block, because this directly follows from C.7)

If B is committed, then from the algorithm construction it is clear that a majority of the replicas will set B as $block_{high}$. This is because, to send a <asynchronous-complete> message with B, a replica should collect at least $n - f$ <vote-async> messages (see line 47). Hence, its guaranteed that if $\tilde{B}$ is formed in view v+1 as a synchronous block, then it will observe B as the $block_{high}$, thus we reach a contradiction.

In the second case, if $\tilde{B}$ is formed in a subsequent view, then it is guaranteed that the level 1 block will extend B by gathering from the <timeout> messages B as $block_{high}$ or a block extending B as the $block_{high}$ (see line 38), hence we reach a contradiction. □

Theorem C.13. *There exists a single history of committed blocks.*

Proof. Assume by way of contradiction there are two different histories H_1 and H_2 of committed blocks. Then there is at least one block from H_1 that does not extend at least one block from H_2. This is a contradiction with theorems C.9, C.10 and C.12. Hence there exists a single chain of committed blocks. □

Theorem C.14. *For each committed replicated log position r, all replicas contain the same block.*

Proof. By theorem C.8, the committed chain will have incrementally increasing round numbers. Hence for each round number (log position), there is a single committed entry, and by theorem C.7, this entry is unique. This completes the proof. □

C.3 Proof of liveness

Theorem C.15. *If at least $n - f$ replicas enter the asynchronous phase of view v by setting isAsync to true, then eventually they all exit the asynchronous phase and set isAsync to false.*

Proof. If $n - f$ replicas enter the asynchronous path, then eventually all replicas (except for failed replicas) will enter the asynchronous path as there are less than $n - f$ replicas left on the synchronous path due to quorum intersection, so no progress can be made on the synchronous path (see line 29) and all replicas will timeout (see line 31). As a result, at least $n - f$ correct replicas will broadcast their <timeout> message and all replicas will enter the asynchronous path.

Upon entering the asynchronous path, each replica creates a asynchronous block with level 1 and broadcasts it (see line 38-39). Since we use perfect point-to-point links, eventually all the level 1 blocks sent by the $n - f$ correct replicas will be received by each replica in the asynchronous path. At least $n - f$ correct replicas will send them <vote-async> messages if the rank of the level 1 block is greater than the rank of the replica (see line 41-42). To ensure liveness for the replicas that have a lower rank, the algorithm allows catching up, so that nodes will adopt whichever level 1 block which received $n - f$ <vote-async> arrives first. Upon receiving the first level 1 block with $n - f$ <vote-async> messages, each replica will send a level 2 asynchronous block (see line 47-51), which will be eventually received by all the replicas in the asynchronous path. Since the level 2 block proposed by any block passes the rank test for receiving a <vote-async>, eventually at least $n-f$ level 2 blocks get $n - f$ <vote-async> (see line 42). Hence, eventually at least $n - f$ replicas send the <asynchronous-complete> message (see line 54), and exit the asynchronous path. □

Theorem C.16. *With probability $p > \frac{1}{2}$, at least one replica commits an elected-asynchronous block after exiting the asynchronous path.*

Proof. Let leader L be the output of the common-coin-flip(v) (see line 57). A replica commits a block during the asynchronous mode if the <asynchronous-complete> message from L is among the first $n - f$ <asynchronous-complete> messages received during the asynchronous mode (see line 58), which happens with probability at least greater than $\frac{1}{2}$. Hence with probability no less than $\frac{1}{2}$, each replica commits a chain in a given asynchronous phase. □

Theorem C.17. *A majority of replicas keep committing new blocks with high probability.*

Proof. We first prove this theorem for the basic case where all replicas start the protocol with $v = 0$. If at least $n - f$ replicas eventually enter the asynchronous path, by theorem C.15, they eventually all exit the asynchronous path, and a new block is committed by at least one replica with probability no less than $\frac{1}{2}$. According to the asynchronous-complete step (see line 65), all nodes who enter the asynchronous path enter view $v = 1$ after exiting the asynchronous path. If at least $n - f$ replicas never set *isAsync* to true, this implies that the sequence of blocks produced in view 1 is infinite. By Theorem C.8, the blocks have consecutive round numbers, and thus a majority replicas keep committing new blocks.

Now assume the theorem C.17 is true for view $v = 0, ..., k - 1$. Consider the case where at least $n - f$ replicas enter the view $v = k$. By the same argument for the $v = 0$ base case, $n - f$ replicas either all enter the asynchronous path commits

a new block with $\frac{1}{2}$ probability, or keeps committing new blocks in view k. Therefore, by induction, a majority replicas keep committing new blocks with high probability. □

Theorem C.18. *Each client command is eventually committed.*

Proof. If each replica repeatedly keeps proposing the client commands until they become committed, then eventually each client command gets committed according to theorem C.17. □