Practical Considerations for Implementing State Machine Replication in the Cloud

Zhiying Liang
Pennsylvania State University
zvl5490@psu.edu

Vahab Jabrayilov
Columbia University
vj2267@columbia.edu

Abutalib Aghayev
Pennsylvania State University
agayev@psu.edu

Aleksey Charapko
University of New Hampshire
aleksey.charapko@unh.edu

***Abstract*—State Machine Replication (SMR) protocols form the backbone of many distributed systems. Enterprises and startups increasingly build their distributed systems on the cloud due to its many advantages, such as scalability and cost-effectiveness. Due to their prevalence in systems, the practical aspects of SMR algorithms, such as efficiency and performance, become hugely important. These practical considerations can impact capacity planning, deployment strategies, and, ultimately, the cost of running stateful systems in the cloud. In this paper, we consider various practical choices that impact the performance and efficiency of state machine replication in the cloud. To that order, we design a language-agnostic Multi-Paxos-based state machine architecture and implement it in several languages popular for cloud deployment. In the process, we investigate the impact of threading, communication, and memory management models on replicated state machines' performance and resource efficiency. We also examine the high-level implications of virtualization on the performance of SMR implementations in various languages. We present our findings as a collection of practical lessons backed by our experimental data and analysis.**

***Index Terms*—State Machine Replication, Cloud, Consensus, Distributed Systems, Performance**

I. Introduction

State machine replication (SMR) has an unprecedented role in modern distributed and cloud computing. SMR algorithms [1]–[5] underpin most critical systems and services, ranging from configuration management [6]–[10] for the cloud infrastructure to massive data-intensive systems that handle petabytes of data and millions of requests per second [11]–[14]. With SMR protocols often on the critical path of data-intensive systems, they need to fulfill a no-compromise role of delivering high throughput with low latency, high reliability, and high efficiency.

The latter point on efficiency, however, can often be neglected in smaller systems and startups, especially in light of cutting upfront costs of development and maintenance. The cloud's unprecedented value proposition—an immediately available infrastructure at the needed scale—further enables companies to focus on their product and growth first. However, it is increasingly becoming clear that when companies grow on the cloud without considering the resource efficiency of their software infrastructure upfront, the cloud cost soars after some point [15], [16] and starts to drive down profit margins, leaving the companies with a dilemma of whether to rewrite or to "repatriate"—move from the cloud back to on-premise [17].

The resource efficiency and performance of data-intensive systems running in the cloud are significantly affected by various factors and decisions engineering teams must make. Language choice is among the most critical decisions, as it can influence many other practical considerations, such as the memory management model. While the automatic memory management offered by the languages with garbage collection (GC) is a boon to programmer productivity [18], it does not come for free: the language runtime needs to run sophisticated GC algorithms, using a considerable amount of CPU and memory in addition to those used by the application code—translating to a higher monthly cloud bill. The language choice can significantly influence other practical considerations, such as the threading and communication models.

In this paper, we focus on systematically studying the impacts of virtualization, memory management, threading, and communication models. Specifically, we first design a canonical SMR system—a MultiPaxos-based distributed in-memory key-value store called Replicant—entirely in pseudocode, relying on common data structures and concurrency primitives available in all languages, such as threads, mutexes, and condition variables, and then we port our pseudocode to C++, Java, Rust, and Go. We then study the individual effects of different communication models (sockets vs. RPCs), threading models (async user-level threads vs. OS-level threading), and memory management (manual vs. automatic with GC). We also study the effect of running Replicant in virtualized and bare-metal environments in these different languages.

We find that GC has a high cost when targeting low tail latency—a requirement for cloud systems [19]. In some memory-constrained workloads, we observed a throughput difference of as much as 10× between Replicant written in a language with manual memory management and one with GC. Other practical considerations also have a significant impact on performance and efficiency. For instance, choosing lower-level communication primitives (e.g., sockets) can have a substantial improvement over higher-level models (e.g., gRPC [20]). Replicant implementations written in async style with user-level threading are more efficient than implementations with OS threads. SMRs are IO-bound, and handling frequent switches in user space leads to better performance. Similarly, the implementations relying on frequent switches to kernel space for thread handling also seem to be impacted the most by the virtualization—as much as 20% degradation compared

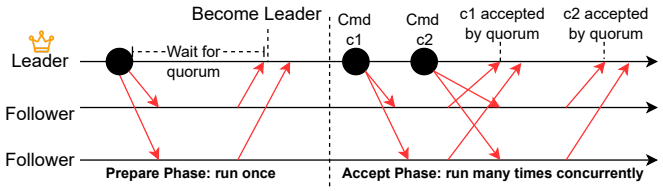


Figure 1: Overview of the MultiPaxos Protocol.

to identically spec'ed bare-metal machines. We present our findings as experimentally-backed lessons about these practical considerations for implementing and running SMR in the cloud. Each such lesson presents a practical outcome to make state machine replication more efficient and performant in the cloud setting. We make our design and implementation publicly available at https://github.com/psu-csl/replicated-store.

II. Background

State machine replications (SMR) [21] is a popular approach to ensure redundancy and reliability in distributed systems. The core idea is to have multiple nodes or replicas execute the same sequence of deterministic operations. These operations transition the state machine predictably and identically across all replicas, allowing systems to tolerate failures.

MultiPaxos [2], [3], [22] and its several "cousins" Raft [1] and Viewstamped Replication [5], [23] are among the most well-known and widely implemented SMR algorithms. They all share the fundamentals and operate in two distinct phases. In the first phase, often called the leader election phase, one replica gets chosen to lead the replication of commands. In the second, or replication phase, the leader proposes commands in sequence, forming a log of state transitions for the state machine. Both phases must be carefully constructed to ensure safety. It is important to avoid the possibility of log divergence; as such, only one active leader can succeed in replicating a command at a time. To this order, a prospective leader must get confirmation from at least a majority of replicas to become a leader. Similarly, when replicating, a leader must secure an acknowledgment for a command from at least a majority of replicas (the replicas won't acknowledge if they know of a newer leader). Since any two majority quorums intersect, any new emergent leader will make it impossible for the old leader to get the majority, forcing the old leader to step down. The quorum intersection also ensures that the new leader will see all commands replicated by the old leader. We illustrate the process in Figure 1.

Language Effects on MultiPaxos. An established approach for comparing a particular property of programming languages—be it performance [24]–[26] or energy efficiency [27]—is to create a benchmark, a set of small programs representing typical workloads with *identical* implementations in the languages being compared, and then run the programs and compare them with regards to the studied property. While such works are useful in their own right, it is hard to extrapolate the results from small programs to large and complex cloud systems. Ideally, to study programming language

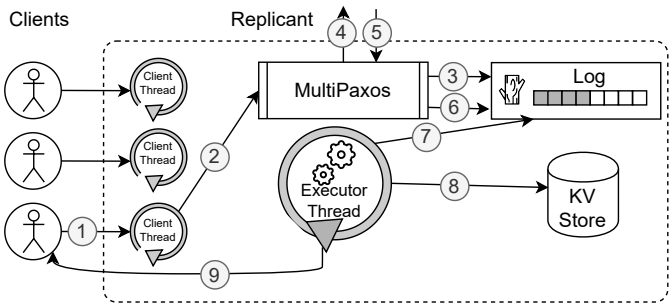


Figure 2: Replicant Design

properties as they pertain to cloud systems, one would create a benchmark consisting of typical and realistic cloud systems —e.g., databases, caches, and web servers—and implement them identically in multiple languages and compare them with regards to the studied property under realistic workloads.

Creating such a benchmark is a significant undertaking. We take the first step in this direction by creating one such cloud system—a fault-tolerant and linearizable distributed key-value store we call Replicant. Such key-value stores are at the core of many cloud services, either as standalone systems [6], [7], [28] or as building blocks for larger systems [11], [12].

III. Replicant Architecture

This section describes the high-level architecture of Replicant and the MultiPaxos module. Our repository contains a detailed design document of these and other modules.

Figure 2 shows the high-level design of Replicant, which consists of the MultiPaxos module, the KVStore module, and the Log module. The KVStore module is a wrapper around an in-memory hash table. The Log module is the MultiPaxos protocol log implemented as a thread-safe unbounded producer-consumer queue. Although log is typically considered an integral part of MultiPaxos, we found that having it as a separate module outside of MultiPaxos results in a cleaner design. Finally, the MultiPaxos module encapsulates the MultiPaxos protocol, which communicates with other MultiPaxos modules embedded in other Replicant peers to reach a consensus on the value of a specific instance in the log.

Replicant creates a new thread for handling each incoming client, assigns a unique id to each client, and maintains a map from the client's id to the client's socket descriptor. Figure 2 shows a typical workflow: (1) a client issues a command to Replicant; (2) a client handler thread passes the command to MultiPaxos; (3) MultiPaxos creates an instance consisting of the command, index of the instance in Log, ballot number, and client-id, and appends the instance to Log; (4) MultiPaxos sends the *accept* message with the instance to all other peers, and (5) once it receives an acceptance from the majority of peers, (6) it commits the instance in Log; finally, when an instance is committed in Log, the Executor Thread (7) extracts the command and the client id from the committed instance, (8) executes the command on KVStore, uses the client id to find the corresponding socket descriptor, and (9) writes

the response to the client. While clients can issue commands concurrently, and thus there may be concurrent *accept phases* running in the MultiPaxos module, the execution of commands occurs serially in the Executor Thread to ensure the same sequence of state transitions in all peers.

IV. Implementation and Experiments Setup

We first designed and implemented Replicant entirely in pseudocode (in multiple iterations over many months), assuming standard concurrency primitives, such as threads, mutexes, and condition variables. Once the design stabilized, the team members concurrently ported the pseudocode into C++, Java, Go, and Rust. The implementations are identical, including in their use of algorithms, data structures, modules, variables, and variable names (sans language-specific conventions, e.g., snake case in Rust and camel case in Java).

We run all experiments with each server running Replicant on an m5.2xlarge instance (8 vCPUs, 32 GiB RAM) on AWS. We use Clang C++ compiler v14.0.6, OpenJDK v19.0.1, Go v1.19.1, and Rust v1.64.0. We collect results on both 3-node and 5-node Replicant clusters. As both configurations exhibit similar trends across all experiments, we present the results of the 3-node cluster to avoid redundancy.

As a benchmark, we run YCSB on a separate m5.2xlarge instance with a variable number of client threads. We choose key and value sizes of 23 B and 500 B, respectively, based on the recent literature [29]. We first populate the database with 7.5 million entries (unless otherwise noted) and perform a 20-second warm-up before running each experiment. Our workloads use a Zipfian request distribution. When reporting a throughput, we repeat every experiment multiple times and report the result with error bars; when reporting a latency CDF, we repeat the experiment multiple times, verify that results are closely aligned across runs, and report one of the samples that is consistent with the others.

V. Practical Lessons From SMR Implementations

Modern programming languages are highly optimized for runtime performance, and many have incrementally adopted useful features from other languages. For instance, C++ introduced coroutines, a user thread feature that existed in Go and Rust, in its recent C++20 standard. However, selecting a programming language for building an SMR system does not always guarantee the anticipated performance. Factors such as underlying library implementations and automatic garbage collection (GC) can introduce significant performance variability in practice. This section presents our findings and the lessons learned from implementing and testing Replicant.

A. Impact of Threading Models

Figure 3 shows the throughput vs. average latency graph of our Replicant implementations in four languages using gRPC for inter-peer communication, along with etcd [7]—a production key-value store implemented in Go that uses Raft for its replicated log. Since the purpose of our project is to produce a clean and modular implementation with acceptable

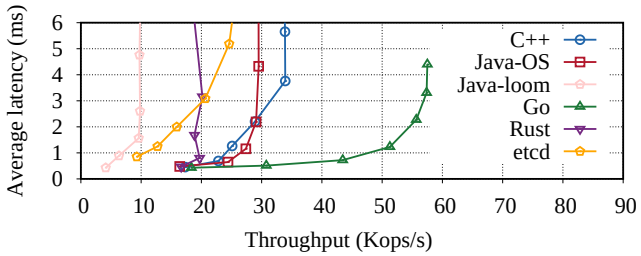


Figure 3: Throughput vs. average latency of Replicant implementations in different languages, under YCSB workload A with 2 million entries, which mitigates GC pressure. Java-OS refers to the OS thread version, and Java-loom stands for the Java's user-thread version with Project Loom. The data points correspond to 8, 16, 32, 64, 128, 192, and 256 concurrent clients. Data points with higher than 6 ms average latency are not shown.

performance in different languages and compare their resource usage, we include etcd in the graph not for performance comparison but as a sanity check to show that Replicant's performance is reasonable, and it has a similar asymptotic behavior. Unlike Replicant, which is an in-memory key-value store, etcd is a persistent key-value store; hence, to make the comparison sensible, in this experiment, etcd uses a RAM disk as the persistent store.

Since Go supports only user-mode threads (i.e., goroutines), we initially envisioned using user-mode concurrency in all languages—specifically, coroutines in C++ [30], Tokio tasks in Rust [31], and Project Loom virtual threads in Java [32]. However, Figure 3 shows lackluster results with Java virtual threads, which are lower than 10 Kop/s. Coroutines are a form of cooperative multitasking and may suffer from performance hits in designs aimed at simultaneous multitasking or parallelism. At the time of our experiments, Project Loom was still in beta and lacked enough performance optimizations. Therefore, we switched to operating system (OS) threads in C++ and Java. In retrospect, this decision allowed us to observe further advantage of user-mode threads over OS threads when building systems in the cloud (§ V-E).

Lesson 1: The performance of asynchronous frameworks across languages varies significantly for the same SMR implementation. For instance, goroutine in Go achieves throughput of 57 Kop/s, while Project Loom struggles to reach 10 Kop/s throughput.

The key observation from Figure 3 is that C++ and Java (for brevity, we refer to a specific Replicant implementation using the language of its implementation), both of which use the OS threads, are closer in their maximum throughputs (29 Kop/s vs. 33 Kop/s) compared to Go, which has a higher maximum throughput (57 Kop/s) and lower average latency at every data point. Go's advantage stems primarily from its use of user-mode threads (we omit our technical analysis to save space, but the theory behind it is well-known [33]). The performance of

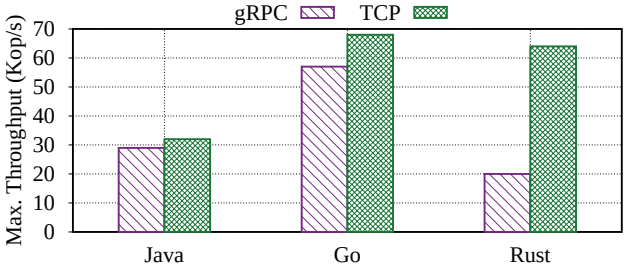


Figure 4: Maximum throughput of Replicant implementations in gRPC and TCP implementations, under YCSB workload A with 2 million entries. We omit the result of C++ using TCP for inter-peer communications, as it cannot provide comparable performance as shown in other comparisons.

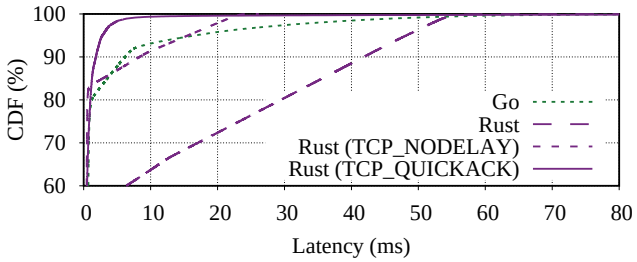


Figure 5: The latency CDFs of Replicant implementations in Go and Rust (using jemalloc allocator) with Nagle algorithm turned off and with TCP Quick ACK running YCSB workload A with 64 clients at target throughput of 50 Kop/s.

Rust Replicant was hindered by several other issues unrelated to the async user-mode threading model. We discuss these issues in the next several subsections.

Lesson 2: SMR implemented with OS threads may be significantly less efficient and performant than the async or user-mode concurrency counterparts since SMR algorithms are IO-bound and experience frequent context switching.

B. Impact of Communication Models

All of our target languages have gRPC implementations. However, Google maintains gRPC for C++, Java, and Go only; for Rust, the Tonic [34] library is a popular and feature-complete gRPC implementation. However, we were puzzled by the performance of our Rust Replicant in the earlier test (Figure 3). Our lengthy debugging and profiling of this anomaly revealed that the H2 library—the HTTP2 library on which Tonic is based—uses a global lock for managing HTTP2 streams [35], limiting its scalability (RPCs in gRPC correspond to HTTP2 streams [36]). Our SMR is highly concurrent and can dispatch hundreds of replication commands at a time, making such bottlenecks in the library visible.

We notified the H2 and Tonic maintainers to remove the global lock in H2. However, for our further experiments, we forked our code, redesigned and reimplemented the network of Replicant to use TCP for inter-peer communication with JSON for message serialization. We omit the detailed description of our redesign (available in our repository). As Rust gets rid of the bottleneck in H2 library, Rust now scales to 64 Kop/s, providing a fair comparison against Go. Although all languages employ nominally "the same" network communication libraries, an underlying implementation difference may lead to significant performance differences.

Lesson 3: Underlying implementation difference may lead to significant performance differences, especially in demanding scenarios like high-concurrency communication prescribed by SMR.

In Figure 4, we compare the maximum throughput of gRPC and TCP-socket versions of three Replicant implementations. While all languages also show some improvement from switching to a lower-level networking framework, the difference in performance for Rust is striking. Specifically, Java reaches 10% higher throughput with TCP sockets (32 Kop/s vs. 29 Kop/s), and Go has 19% improvement (68 Kop/s vs. 57 Kop/s). Rust Replicant's throughput, however, is more than tripled with TCP sockets. Some improvement from switching to sockets is expected since gRPC adds overhead over TCP. Moreover, we opted for simple RPC rather than gRPC stream APIs in our project, which also obviates some gRPC benefits.

Lesson 4: While gRPC is easier to implement, it can substantially impact the performance and efficiency of SMR systems—as much as 20% in the case of an implemented gRPC library.

When using TCP, one must consider several common settings and their impact on throughput and latency. Nagle's algorithm [37], usually exposed by a TCP_NODELAY option on a socket [38], is one such setting. When Nagle's algorithm is on, some packets may experience a delay as the algorithm combines small packets together instead of sending them immediately. Setting a TCP_NODELAY flag disables the algorithm and can improve performance at the expense of slightly higher bandwidth usage.

Unlike other languages, Go disables Nagle's algorithm [37] by default. Figure 5 shows Nagle's algorithm has an effect on tail latency in Rust when running the YCSB workload A with 64 clients and a target throughput of 50 Kop/s. Go's 99th percentile latency (47 ms) is lower than that of Rust (54 ms), and with Nagle's algorithm disabled, Rust achieves a lower 99th percentile latency (22 ms) than Go. As an aside, we also show the effect of using TCP Quick ACK (recommended by Nagle as a better alternative to his algorithm [39]), which reduces Rust's 99th percentile latency down to 7 ms. Moreover, as Figure 5 illustrates, turning off Nagle's algorithm in Rust improves the latency across the board with substantial benefit even for 60th percentile latency. In the rest of this paper, we disable Nagle's algorithm in all our implementations for all

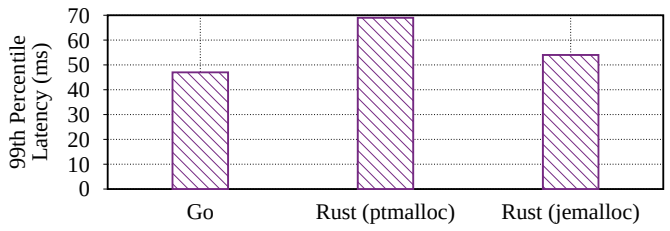


Figure 6: The 99th percentile latency in Go and Rust using memory allocators ptmalloc and jemalloc, respectively with 64 clients and target throughput of 50 Kop/s.

experiments. However, we do not enable TCP Quick ACK in subsequent experiments for a fairer comparison since Go does not readily expose that feature.

Lesson 5: Tail latency can be significantly impacted by several common options, such as disabling Nagle's algorithm and enabling TCP Quick ACK.

C. Impact of Memory Allocator

Rust is a language with manual memory management and a few additional memory management restrictions aimed at improving the memory safety of programs. However, in our investigation and profiling of Rust's performance, we found another performance culprit in memory allocation. The default Linux memory allocator, `ptmalloc` [40], is an additional bottleneck, mainly impacting the tail latency. As Figure 6 shows, replacing `ptmalloc` with `jemalloc`—a highly optimized allocator developed at Facebook [41]—results in significantly closer gap at 99th percentile latency to the one of Go with 64 clients. For the remaining experiments, the Rust implementation of Replicant uses `jemalloc`. Note that we close the latency gap between Rust and Go by disabling Nagle's algorithms on top of using `jemalloc`, as shown in Figure 5 and discussed in the previous section (§ V-B)

Lesson 6: The performance of memory allocation (i.e., `malloc` calls) may vary. Using non-standard allocators may help SMR solutions maintain lower latency, especially for SMRs that manage many small data items/commands.

D. The Impact of Automatic Garbage Collection

Having addressed the concurrency, networking, and memory allocation issues, inconsistencies, and bottlenecks, we are ready to investigate the impact of an automatic GC on the performance of SMR solutions in the cloud. In this section, we quantify the cost of GC on both 3-node and 5-node Replicant clusters running on AWS.

Due to the two different threading and networking models used, we break down our GC investigation into two groups. The first group consists of Replicant implementations in C++ and Java, the two languages with manual memory management and GC, respectively, and uses OS threads and gRPC for inter-peer communication. The second group is Rust and Go Replicant versions, which use user-mode threads and TCP for inter-peer communication.

Methodology. We focus our evaluation on two workloads: update-heavy workload A (50% reads, 50% writes) and read-heavy workload B (95% reads, 5% writes) of YCSB [42]. To better understand the cost of automatic GC, we run experiments in different configurations that reduce the memory size or the vCPU count. These configurations aim to create more memory pressure and fewer CPU resources available to perform GC. Specifically, on each m5.2xlarge instance running a Replicant leader or follower, we reduce the memory size from 32 GiB to 8 GiB, 6 GiB, and 5 GiB without changing the vCPU count (8) and run each workload in each of these configurations (we stop at 5 GiB because our raw dataset size is $\approx$4 GiB). We then reduce the vCPU count from 8 to 4 and 2 without changing the memory size (32 GiB) and run each workload in each of these configurations. We reduce the memory size for C++ and Rust with `systemd`'s `MemoryMax` directive, for Go with the `GOMEMLIMIT` environment variable, and for Java with the `-Xmx` parameter of JVM. We reduce the vCPU count by turning vCPUs off via `sysfs` [43].

Unlike previous experiments where we compared the unrestricted performance of the SMR systems, the industry rarely runs systems at full resource utilization. Instead, capacity planning often involves setting acceptable performance thresholds and benchmarking software until exceeding these thresholds. More concretely, we can expect certain acceptable latency from a service and, as such, set a threshold of never exceeding such latency. As such, in our GC experiments, we fix a target tail latency and empirically find the maximum achievable throughput that hits but does not exceed such target tail latency [44, p. 4]. Specifically, we set the target 99th percentile latency at 20 ms, meaning that 99% of requests are completed within 20 ms. The value of 20 ms is 10 times of average latency observed in Figure 4, where the GC pressure is minimal.

We follow the same practice for each configuration specified above. In each configuration, we run multiple experiments (each experiment runs for 10 minutes after a 20-second warm-up), adjusting the throughput until we hit our target tail latency. Specifically, we begin with a high throughput setting, and if it exceeds the target tail latency, we repeat the experiment with a lower throughput. For these experiments, we fix the number of client threads at 64 to generate the load and vary the throughput by changing YCSB's target throughput parameter.

GC Overhead Under an Update-heavy Workload. We first examine the performance of different languages under an update-heavy workload (Workload A). Figure 7 (a) and Figure 7 (c) show the maximum achievable throughput with the 99th percentile latency of 20 ms as we vary the available memory. Figure 7 (b) and Figure 7 (d) show Replicant leader's aggregate vCPU utilization (i.e., 100% utilization corresponds to all 8 vCPUs fully utilized) for each experiment. Since C++ and Rust's vCPU utilization stays the same, at 75% and 60%, respectively, during all experiments, we show it once.

Looking at the 32 GiB memory experiment, we already see the difference in throughput between SMR systems in manual

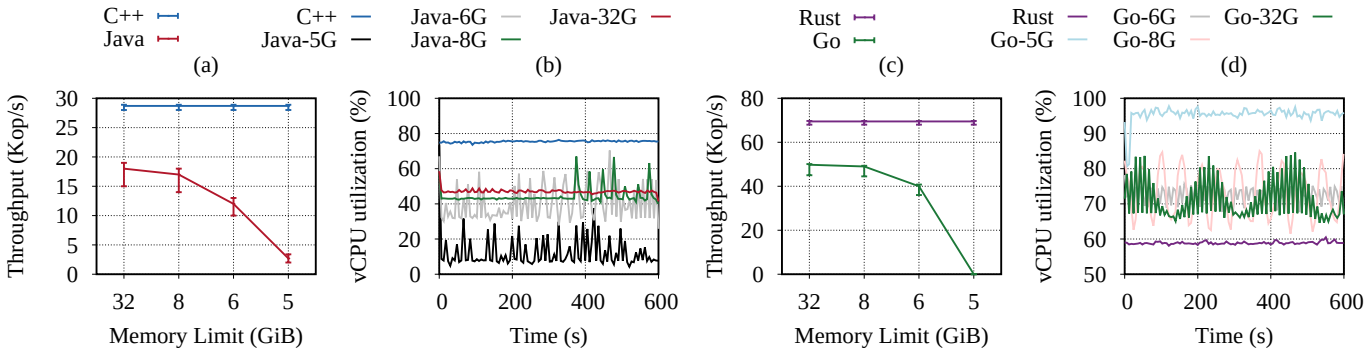


Figure 7: Maximum throughput and the aggregate vCPU utilization on a Replicant leader constrained by 99th percentile latency of 20 ms under workload A with 8 vCPUs. (a) The throughput and (b) vCPU utilization of C++ and Java. (c) The throughput and (d) vCPU utilization of Rust and Go. The vCPU utilization of C++ and Rust stays the same across experiments. We abbreviate implementation with the memory size, e.g., Java-5G corresponds to vCPU utilization with 5 GiB memory size.

and automatic GC languages. The throughput of C++ and Java are 28.7 Kop/s and 18 Kop/s, respectively, signifying a bigger gap between C++ and Java than in a test with unrestricted tail latency (Figure 7 (a)). In fact, if we do not restrict the tail latency, then C++ is only 14% faster than Java, while C++ is 60% faster when we limit the capacity based on the tail latency. A similar trend happens with Go and Rust. In the unrestricted mode, Go Replicant was slightly more performant than Rust, but when taking into account latency performance thresholds, Rust is nearly 40% faster.

Lesson 7: When tail-latency does not matter, SMR written in a language with automatic GC can compete and even outperform manual memory management counterpart. However, when taking into account practical latency limits on what may be an acceptable response time, GC SMRs start to underperform.

Continuing on the memory reduction experiments, Figure 7 (a,c) shows that as we keep reducing the available memory size, the GC Replicant (e.g., Java and GO) throughput degrades further, while the reduced memory does not impact non-GC Replicant performance. When looking at the corresponding CPU utilization figures (Figure 7 (b,d)), we can see even less CPU usage from Java as it needs more unused CPU to accommodate larger and longer GC usage resulting from higher memory pressure. Go's garbage collector has a different CPU usage pattern from Java's. In Go, we can see pronounced CPU spikes when GC is active, resulting in overall high CPU usage compared to its manual memory management counterpart (Rust). Likely, Java's GC occupies the CPU in shorter spikes that do not register at our measurement resolution (but still impact average CPU usage). At high memory pressure, Go's GC dominates the CPU, leaving no capacity to perform actual work – Go cannot sustain any meaningful throughput and can only do around 100 ops/s at 154ms latency.

This performance difference between GC and non-GC languages occurs because the GC, when active, takes up some processing resources (i.e., the CPU) that could have been used for handling user requests. As such, the lower throughput is a proxy for the CPU cost needed to accommodate the demands of GC without impacting tail latency. Even if GC does not run continuously, GC usage will temporarily use some CPU, contributing to queuing effects and affecting the latency of ongoing requests in the SMR.

Lesson 8: SMR solutions implemented in GC languages must leave enough idle capacity/resources to accommodate for GC-induced CPU usage without a negative impact on latency. Without leaving enough idle capacity for GC, tail latency will grow.

We use GC traces to confirm further that GC consumes significant CPU resources when active and causes performance variance. We enable the `GODEBUG=gctrace` option for Go and `-Xlog:gc*` for Java, allowing us to capture detailed GC activities during runtime. Specifically, once we determine the maximum achievable throughput under workload A and different memory configurations for Go and Java, we repeat the experiment with the GC trace option enabled, maintaining identical experiment settings.

Figure 8 vCPU utilization by GC and the frequency of GC events at a Replicant leader during our experiment. Each point represents a GC event occurrence during the runtime, with denser points indicating more frequent GC activities. Given the high frequency and even distribution of these events, we select a segment of runtime data to present the graph more effectively. Note that, in contrast to previous vCPU utilization graphs, Figures 8 (a) and (c) exclusively show the **instantaneous** utilization caused by GC, capturing the vCPU usage during GC activities, rather than throughout the entire runtime. In contrast,Figures 8 (b) and (d) capture the cumulative CPU time spent in GC activities every 5 seconds—a GC-only component of CPU utilization.

The data for Go shows that under high memory pressure (e.g., having only 5 GiB of RAM for a 4 GiB dataset), Go spends nearly all of its CPU time on GC. With more available memory, the frequency of GC and the cumulative CPU usage

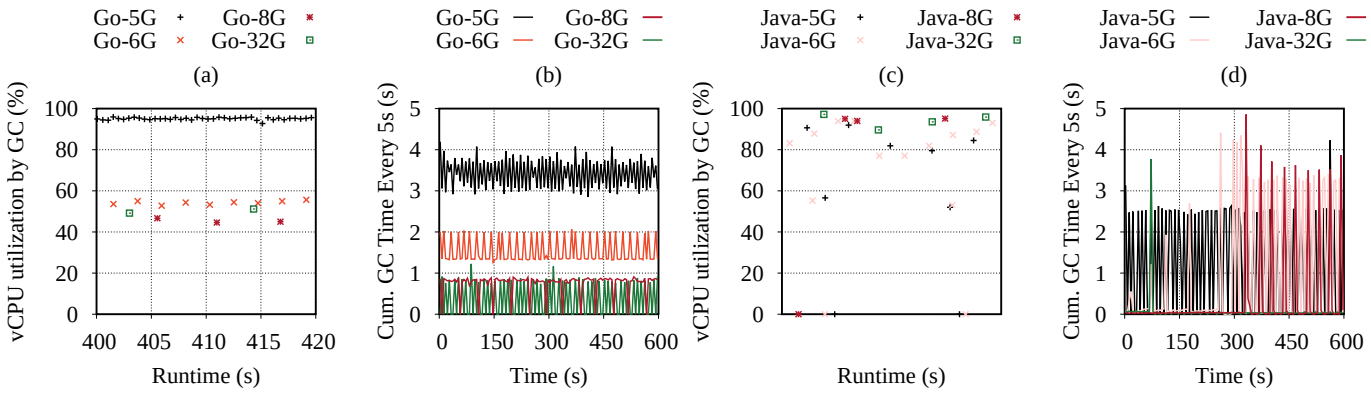


Figure 8: Real-time instantaneous vCPU utilization and cumulative runtime of the garbage collector under workload A with 8 vCPUs and varied memory configurations. The cumulative runtime reflects the aggregation of the duration of all GC activities within a tumbling 5-second window. (a) and (b) Go garbage collector. (c) and (d) Java garbage collector.

decreases. In fact, going from 6 GiB to 5 GiB, the frequency of GC activities increases 5-fold. Figure 8 (b) illustrates that at 5 GiB memory, Go's garbage collector runs between 3 and 4 seconds in each 5-second window, representing a GC-only CPU usage of between 60% and 80%, leaving practically no resources for useful work. The CPU demands of Go's GC reduce drastically under less memory pressure, hovering around 1 second of cumulative time per window (roughly 20% CPU usage) for 8 GiB setup.

Lesson 9: CPU utilization by GC may significantly exceed CPU utilization of core application/system under severe memory pressure, so nodes must be sized accordingly.

Unlike Go, Java's GC traces do not offer comprehensive data on CPU usage. Consequently, Figure 8 (c) shows the vCPU utilization of the clean-up phase only, excluding the concurrent mark phase in Java GC. Although Java spends less time on GC compared to Go, it still exhibits notable vCPU spikes, as illustrated in Figure 8 (c).

It is important to note that we are not conducting comparison or analysis on each garbage collector. Java's default G1 garbage collector aims to balance between minimizing pause time and higher throughput. It is consistent with our key observation in our experiments. The **instantaneous** vCPU utilization is amortized in Figure 7 (b) because the GC duration is short. However, Figure 8 (c) shows that Java needs to keep available CPU to deal with those regular high GC spikes. Furthermore, the cost of full GC remains huge. When the full GC is triggered, it results in more time spent on clean-up and marking in the latter runtime shown in Figure 8 (d), leading to much longer 99th percentile latency. These longer marking phases and full GC events inevitably cause significantly higher tail latency in Java.

Another way to demonstrate GC's need for idle CPU to facilitate low tail latency is by reducing the available vCPUs. For example, looking at Java's 48% vCPU utilization in Figure 7 (b), one may erroneously assume that halving the vCPU

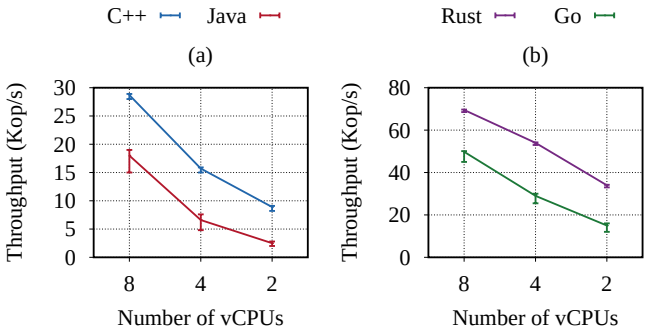


Figure 9: vCPU reduction experiment constrained by 99th percentile latency of 20 ms under workload A and 32 GiB of memory. (a) Maximum throughput of C++ and Java. (b) Maximum throughput of Rust and Go.

count won't impact the throughput. Figure 9 shows the results of vCPU reduction experiments and proves the assumption wrong. Figure 9 (a) shows the maximum throughput for C++ and Java using different vCPU configurations and 32 GiB of memory, and Figure 9 (b) shows the vCPU utilization at a Replicant leader during the same experiment. When dropping the vCPU count from 8 to 4 while still keeping ample memory, the throughput of Java drops from 18 Kop/s to 7 Kop/s, and its utilization drops from 48% to 38%, showing that Java needs to keep a large portion of idle CPU around to handle continuous GC while satisfying the target tail latency. Similar trends are seen for Go as compared to Rust in Figure 9 (a).

Furthermore, our findings reveal that Java and Go are less resource-efficient and cost-efficient than C++ and Rust. To quantify resource efficiency, we normalize the throughput by runtime vCPU utilization, i.e. throughput-per-unit-of-vCPU-utilization [45]. Rust achieves a consistent 1200 op/s/vCPU% with 8 vCPUs, while Go's efficiency declines from 700 op/s/vCPU% to 555 op/s/vCPU% as the memory size decreases. While Go consumes more vCPU utilization, Java uses less vCPU utilization than C++ but at the cost of reduced

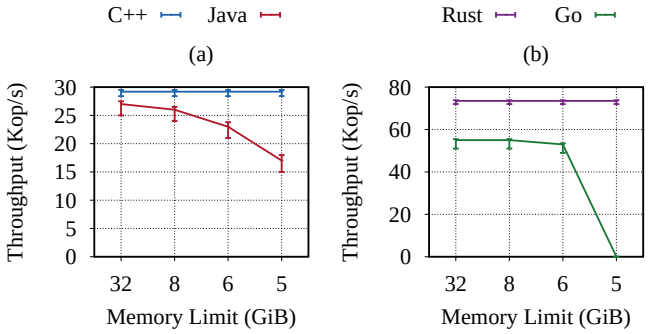


Figure 10: Maximum throughput constrained by 99th percentile latency of 20 ms under workload B with different memory configurations and 8 vCPUs. (a) Maximum throughput of C++ and Java. (b) Maximum throughput of Rust and Go.

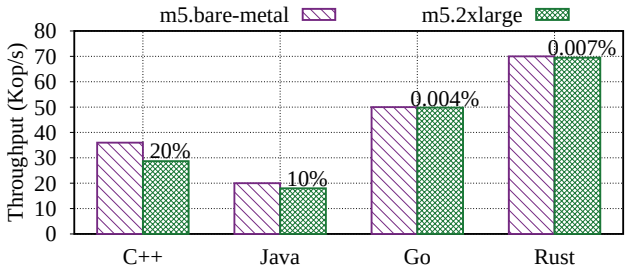


Figure 11: Maximum throughput constrained by 99th percentile latency of 20 ms in workload A on m5.metal instances with 8 online logical CPU cores (4 physical cores due to hyperthreading) and memory limited to 32 GiB and on m5.2xlarge instances, including the percentage decrease in throughput.

throughput. However, cloud service providers charge based on whole virtual machine sizes instead of actual runtime resource utilization, even when resources are underutilized. Figure 9 shows that, to reach the same maximum achievable throughput as C++, Java needs to double the number of vCPUs. Go demands even more vCPU resources to achieve Rust's performance. Doubling resource allocations directly translates into doubling costs. Consequently, achieving comparable performance with GC-based languages necessitates provisioning larger machines, leading to higher costs.

For both experiment types—memory and vCPU reduction—we observe similar trends in the vCPU utilization on Replicant followers, but at a smaller scale; e.g., for memory reduction experiments, C++'s vCPU utilization remains at 30% while Java's vCPU utilization starts at 12% and drops to 2%. Rust maintains steady at 17% vCPU while Go fluctuates within a lower range from 40% to 50%. We omit the graphs to save space.

Lesson 10: In a write-heavy workload with a restricted maximum allowed operation latency, SMR implementations in non-GC languages achieve more than 1.4× higher throughput than the SMRs in GC-based languages, even with ample memory.

GC Overhead Under a Read Heavy Workload. Figure 10 shows the results of memory reduction experiments for a read-heavy workload (workload B). The experiments are identical to those of Figure 7 with the only change being switching from update-heavy workload A to read-heavy workload B. Overall, we can observe similar trends – GC-based Replicant implementations have smaller throughput than non-GC versions. One substantial difference is the overall performance of the Java version compared to C++. Go and Rust had similar performance discrepancies as with the write-heavy workload at ample memory. However, Go managed to maintain its performance without degradation with reduced available memory except for severe memory reduction (the 5 GiB) deployment.

Similarly, for the vCPU reduction experiments of workload B we observe the same trend to the vCPU reduction experiments of workload A, but with a smaller delta between the throughput of C++ and Java as well as Rust and Go at each datapoint. We omit the vCPU reduction experiment graph for workload B to save space. Similar to workload A experiments, we observe identical trends for both experiment types on Replicant followers, but at a smaller scale.

Lesson 11: Read-heavy SMR workloads require less cleanup, allowing SMRs in languages with automatic GC to perform closer to their non-GC counterparts.

E. Virtualization Overhead

We conclude this section by exploring how the virtualization overhead on the cloud affects languages. To this end, we repeat the workload A experiments with no resource limitations on m5.2xlarge instances and on m5.metal instances (96 CPUs, 384 GiB memory) with 8 online logical CPU cores (we turn off the rest via `sysfs` [43]) and with memory size reduced to 32 GiB—to match the m5.2xlarge configuration. Using `lscpu`, we confirm that both instance types have identical CPUs (Intel Xeon Platinum 8259CL CPU at 2.50GHz).

Figure 11 shows the maximum achievable throughputs with 99th percentile latency of 20 ms on bare metal and virtualized instances and the percentage decrease in throughputs. We observe that C++ suffers the most from the virtualization overhead followed by Java, with drops of 20% and 10% in throughputs, respectively. Go and Rust, on the other hand, have a negligible drop in throughput. While it appears that the languages with user-mode threads are at an advantage, we leave a data-backed explanation of this phenomenon as future work. Our key observation from these results is that language comparisons on bare metal may not translate to the cloud.

Lesson 12: Languages using system-level threads may suffer from larger virtualization overhead. The benchmarking results collected on cloud VM may not be comparable with bare-metal machines.

VI. Language Experience and Discussion

In our subjective experience of building Replicant in multiple languages, we avoided language-specific optimizations and tried to use as many out-of-the-box features of languages as possible. While each language can be tuned with expert knowledge of compiler or garbage collector internals, it would require a level of sophistication that is not available to most programmers. To put our experience into context, the programming experience of our team members includes 10+ years of C++, 7+ years of Java, and 4+ years of Go—some of these at hyperscalers and medium-sized software companies. We had no prior Rust experience, but we studied it [46], [47] for a month before starting our Rust implementation.

Go, by far, was the quickest language to build Replicant in, with the least friction and without resorting to third-party libraries, and it provided a decent performance from the get-go. Java and C++ were next in terms of ease of development and project completion time. For C++, we additionally integrated AddressSanitizer [48] into our development workflow, which found a few use-after-free bugs at different stages of the development. Rust was the most challenging language to work with initially, and not only because we lacked prior experience, but because it requires doing routine things differently; e.g., in most languages, knowing how to use a mutex and a structure is enough for implementing a monitor [49]; in Rust, on the other hand, it requires an unobvious pattern [50].

To overcome our initial friction with Rust, we purchased a one-month Rust consultancy (for 512 €s), which helped us to overcome early roadblocks in our implementation; we got introduced to advanced techniques and idioms not covered in the books, which were essential to our implementation. As we became comfortable with Rust, it started to have a significant impact—in a good way—on our project. It forced us to rethink our interfaces (and thereby our design) to satisfy the complaints of the Rust compiler, which made the interfaces simpler, less sloppy, and less error-prone, and we had to go back and implement those changes in other languages. Finally, profiling efforts for identifying the H2 and `ptmalloc` bottlenecks (§ V-B) were substantial.

Focusing on the manual memory management overhead of the languages, while C++ successfully hides most of this overhead from programmers using the RAII idiom [51]—our C++ codebase never explicitly allocates or frees memory—we still had to think about the lifetime and ownership of objects during the design. Furthermore, since there is no memory-safety guarantee, we had to integrate AddressSanitizer into our development workflow.

Rust, unlike C++, (largely) guarantees memory safety and data-race-freedom at compile time. These guarantees, however, come at the cost of a steep learning curve with unfamiliar and perplexing concepts and a generally slower pace of development because the compiler will not allow "hacking away" without thinking through the lifetimes and concurrency implications of the new code—and thinking is hard and takes time. The assistance we got from consultants helped us to overcome the initial impedance with the language, and, e.g., we replaced gRPC in Replicant to use TCP for inter-peer communication (§ V-B) relatively quickly (it was not trivial and required a redesign). We also use jemalloc in place of ptmalloc, as the underlying memory allocator affects the performance significantly (§ V-C. After writing many thousands of lines of Rust, our subjective opinion is that we feel comfortable building large projects in Rust, but others have reported a different experience [52].

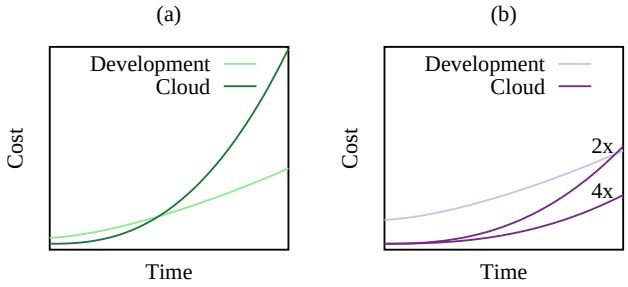


Figure 12: A simplistic model for the long-term development and cloud costs of building a cloud system in a language (a) with GC and (b) with manual memory management.

To sum up, using a language with manual memory management does have a higher development cost than using a language with GC. Therefore, our observation is that the choice of a programming language for building a cloud system is a trade-off between the long-term development cost and cloud cost. The types of request workloads also have impact on the GC overhead, where GC languages suffer more from update-heavy workloads. We illustrate this trade-off with a simplistic model in Figure 12. Figure 12 (a) shows the long-term costs of building a cloud system in a language with GC: the development cost starts higher than the cloud cost and grows slowly; the cloud cost starts low, but as the system grows in popularity, the cloud cost ends up dwarfing the development cost in the long run. Figure 12 (b) shows the same for a language with manual memory management, where the development cost starts 2× higher than Figure 12 (a) and grows slowly; we show how the cloud cost would look like with the same growth rate as in Figure 12 (a) if the system were using 2× fewer resources and 4× fewer resources.

In conclusion, our observation is that if a cloud system is expected to grow to a large volume of users, using a language with manual memory management would significantly reduce long-term cloud cost despite high development cost. Conversely, if the system is expected to grow up to a certain small bound, then using a language with GC would reduce long-term development cost.

VII. Related Work

Our work is the first study that implements a large cloud system in languages with GC and manual memory management to systematically quantify the overhead of GC.

Language choice. Several prior works use micro and macro benchmarks to discuss and showcase the impact of language choice on systems. Cutler et al. [53] compare the kernel implemented in Go and the Linux kernel (implemented in C). The authors emphasize Go's security and ease of implementation while briefly touching on the performance cost incurred by the Go garbage collection and safety checks. Lion et al. [26] present the first study analyzing and comparing multiple widely used languages with the instrumented benchmarks. In this work, the authors introduce LangBench, a new benchmark suite for evaluating language performance, and use the suite to study the overheads and the advantages of having a managed runtime. Like us, the LangBench uses idiomatic and identical implementations across languages. However, all their tested problems are single-machine applications and may not represent all workloads in the modern cloud. The study finds that while the type and bounds checking, interpreter, and GC write barriers incur an overhead, applications benefit from improved cache locality due to GC and have better scalability (in Go) due to the runtime scheduler. The LangBench results relied on the in-house bare metal servers, and as we show in our work, the virtualization used in the cloud can significantly impact performance. Similar to other studies, the experience report by Hundt [54] compares identical micro-benchmark implementations in various languages. The report, however, highlights the dramatic impact of language-specific optimizations and tuning when deviating from generic implementation. Such potential for different performance mirrors our experience with the first gRPC-based implementation of Replicant.

Efficiency in the cloud. The efficiency of languages and runtimes has gained traction in the past few years. Most works [27], [55] have analyzed various languages for their energy consumption using different benchmark problems [56], [57]. Maas et al. [58] advocate rethinking language runtimes for better performance in modern data centers. The authors argue against using JIT compilation in the cloud due to the short-lived nature of cloud workloads. While they also highlight the performance overhead of GC, they don't favor the idea of improving or replacing the GC. Instead, they propose to have lightweight abstractions to reduce the number of objects. The Treehouse project [59] calls for a new software infrastructure that treats energy and carbon as first-class resources along with traditional computing resources. A recent report by Amazon [60] emphasizes the need for energy-efficient implementations for sustainable computing. The report advocates that one can have a C-like performance with Rust while getting more sustainability and security benefits. Matte et al. [45] investigates the resource efficiency of popular SMR protocols in the cloud. The authors conclude that fast systems may not always be the most efficient, as speed can be achieved by the ability to consume more resources instead of their efficient use. Unlike that study, we compare the efficiency of a specific SMR algorithm implemented in different languages and runtime paradigms.

VIII. Conclusion

This paper presents several experimentally backed lessons and observations on the performance and efficiency of state machine replication protocols in the cloud. Our findings focus on threading, communication, and garbage collection and show many impactful considerations an engineering team must make when designing SMR-based data-intensive systems.

IX. Acknowledgements

This project is in part sponsored by the National Science Foundation (NSF) under award numbers CNS-2149443, CNS-2149389, and CNS-2440896.

References

[1] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[2] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: An engineering perspective," in *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 398–407. [Online]. Available: https://doi.org/10.1145/1281100.1281103

[3] R. Van Renesse and D. Altinbuken, "Paxos made moderately complex," *ACM Comput. Surv.*, vol. 47, no. 3, feb 2015. [Online]. Available: https://doi.org/10.1145/2673577

[4] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 358–372. [Online]. Available: https://doi.org/10.1145/2517349.2517350

[5] B. M. Oki and B. 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*, 1988, pp. 8–17.

[6] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10. USA: USENIX Association, 2010, p. 11.

[7] etcd Authors, "etcd: A distributed, reliable key-value store for the most critical data of a distributed system," https://etcd.io/, 2023.

[8] zeus Authors, "Zeus - Build dashboards from an SQL database," https://www.zeusdash.com/, 2016.

[9] delos Authors, "Delos: Simple, flexible storage for the Facebook control plane," https://engineering.fb.com/2019/06/06/data-center-engineering/delos, 2019.

[10] M. Brooker, T. Chen, and F. Ping, "Millions of tiny databases," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 463–478. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/brooker

[11] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, D. Woodford, Y. Saito, C. Taylor, M. Szymaniak, and R. Wang, "Spanner: Google's globally-distributed database," in *OSDI*, 2012.

[12] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis, "CockroachDB: The Resilient Geo-Distributed SQL Database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD 2020)*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1493–1509. [Online]. Available: https://doi.org/10.1145/3318464.3386134

[13] mongodb Authors, "MongoDB: The Developer Data Platform — MongoDB," https://www.mongodb.com/, 2009.

[14] M. Elhemali, N. Gallagher, N. Gordon, J. Idziorek, R. Krog, C. Lazier, E. Mo, A. Mritunjai, S. Perianayagam, T. Rath, S. Sivasubramanian, J. C. S. III, S. Sosothikul, D. Terry, and A. Vig, "Amazon DynamoDB: A scalable, predictably performant, and fully managed NoSQL database service," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 1037–1048. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/elhemali

[15] J. O. Willard, "How to respond to the increasing costs of cloud: a CIO guide," https://www.ibm.com/blogs/internet-of-things/increasing-cloud-costs-cio-guide/, 2022.

[16] K. Gulen, "Cloud costs have started to become a heavy burden for the IT sector," https://dataconomy.com/2022/09/cloud-costs-are-skyrocketing-it-industry/, 2022.

[17] S. Wang and M. Casado, "The cost of cloud, a trillion dollar paradox," *Andreessen Horowitz*, vol. 27, 2021.

[18] G. Phipps, "Comparing observed bug and productivity rates for Java and C++," *Software: Practice and Experience*, vol. 29, no. 4, pp. 345–358, 1999.

[19] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013. [Online]. Available: http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[20] Google, "gRPC: A high performance, open source universal RPC framework," https://grpc.io/, 2023.

[21] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[22] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, may 1998. [Online]. Available: https://doi.org/10.1145/279227.279229

[23] B. Liskov and J. Cowling, "Viewstamped replication revisited," 2012.

[24] R. Hundt, "Loop Recognition in C++/Java/Go/Scala," in *Proceedings of Scala Days 2011*, 2011. [Online]. Available: https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf

[25] L. Prechelt, "An empirical comparison of C, C++, Java, Perl, Python, Rexx and Tcl," *IEEE Computer*, vol. 33, no. 10, pp. 23–29, 2000.

[26] D. Lion, A. Chiu, M. Stumm, and D. Yuan, "Investigating managed language runtime performance: Why JavaScript and python are 8x and 29x slower than c++, yet java and go can be faster?" in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 835–852. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/lion

[27] M. Couto, R. Pereira, F. Ribeiro, R. Rua, and J. a. Saraiva, "Towards a green ranking for programming languages," in *Proceedings of the 21st Brazilian Symposium on Programming Languages*, ser. SBLP '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3125374.3125382

[28] T. Authors, "Tikv is a highly scalable, low latency, and easy to use key-value database." https://tikv.org, 2023.

[29] J. Yang, Y. Yue, and K. V. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at twitter," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 191–208. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/yang

[30] cppreference.com, "Coroutines (C++20)," https://en.cppreference.com/w/cpp/language/coroutines, 2023.

[31] Tokio, "A runtime for writing reliable network applications without compromising speed," https://docs.rs/tokio/latest/tokio/, 2023.

[32] Oracle, "Loom - Fibers, Continuations and Tail-Calls for the JVM," https://openjdk.org/projects/loom/, 2023.

[33] R. Pressler, "Why user-mode threads are good for performance," https://www.youtube.com/watch?v=07V08SB1l8c, 2022.

[34] T. Authors, "A rust implementation of gRPC," https://github.com/hyperium/tonic, 2023.

[35] h2 Authors, "Streams Mutex," https://github.com/hyperium/h2/blob/a6b414458fd7687f53df68861f4833cf142e5b76/src/proto/streams/streams.rs#L25, 2023.

[36] J. de Klerk, "gRPC on HTTP/2 Engineering a Robust, High-performance Protocol," https://grpc.io/blog/grpc-on-http2/, 2023.

[37] J. Nagle, "Congestion control in ip/tcp internetworks," https://datatracker.ietf.org/doc/html/rfc896, 1984.

[38] R. Cox, "preliminary network - just dial for now," https://github.com/golang/go/blob/e8a02230f215efb075cccd4146b3d0d1ada4870e/src/lib/net/net.go#L398, 2008.

[39] J. Nagle, "That still irks me. the real problem is not tinygram prevention." https://news.ycombinator.com/item?id=10608356, 2015.

[40] G. C. L. Authors, "Overview of Malloc," https://sourceware.org/glibc/wiki/MallocInternals, 2023.

[41] J. Evans, "Scalable memory allocation using jemalloc," https://engineering.fb.com/2011/01/03/core-data/scalable-memory-allocation-using-jemalloc/, 2023.

[42] S. Busbey, "Core Workloads," https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads, 2015.

[43] S. Andrzej, R. Russell, S. Vaddagiri, A. Raj, J. Schopp, and T. Gleixner, "Cpu hotplug in the kernel," https://docs.kernel.org/core-api/cpu_hotplug.html, 2021.

[44] R. L. Sites, *Understanding Software Dynamics*. Addison-Wesley Professional, 2021.

[45] V. S. Matte, A. Charapko, and A. 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*, ser. HotStorage '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 42–49. [Online]. Available: https://doi.org/10.1145/3465332.3470882

[46] S. Klabnik and C. Nichols, "The rust programming language," https://doc.rust-lang.org/stable/book/, 2023.

[47] J. Orendorff, J. Blandy, and L. F. S. Tindall, "Programming rust, 2nd edition," https://learning.oreilly.com/library/view/programming-rust-2nd/9781492052586/, 2021.

[48] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. USA: USENIX Association, 2012, p. 28.

[49] Wikipedia, "Monitor," https://en.wikipedia.org/wiki/Monitor_(synchronization), 2023.

[50] A. Ryhl, "Shared mutable state in rust," https://draft.ryhl.io/blog/shared-mutable-state/, 2022.

[51] cppreference.com, "Raii," https://en.cppreference.com/w/cpp/language/raii, 2023.

[52] M. Welsh, "Using rust at a startup: A cautionary tale," https://mdwdotla.medium.com/using-rust-at-a-startup-a-cautionary-tale-42ab823d9454, 2022.

[53] C. Cutler, M. F. Kaashoek, and R. T. Morris, "The benefits and costs of writing a POSIX kernel in a high-level language," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 89–105. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/cutler

[54] R. Hundt, "Loop recognition in c++/java/go/scala," in *Proceedings of Scala Days 2011*, 2011. [Online]. Available: https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf

[55] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. a. P. Fernandes, and J. a. Saraiva, "Energy efficiency across programming languages: How do energy, time, and memory relate?" in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 256–267. [Online]. Available: https://doi.org/10.1145/3136014.3136031

[56] T. C. L. B. Game, "The computer language benchmarks game," 2022. [Online]. Available: https://sschakraborty.github.io/benchmark/index.html

[57] R. Code, "Rosetta code — rosetta code,," 2022, [Online; accessed 17-April-2023]. [Online]. Available: https://rosettacode.org/w/index.php?title=Rosetta_Code&oldid=322370

[58] M. Maas, K. Asanović, and J. Kubiatowicz, "Return of the runtimes: Rethinking the language runtime system for the cloud 3.0 era," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 138–143. [Online]. Available: https://doi.org/10.1145/3102980.3103003

[59] T. Anderson, A. Belay, M. Chowdhury, A. Cidon, and I. Zhang, "Treehouse: A case for carbon-aware datacenter software," in *Workshop on Sustainable Computer Systems Design and Implementation*, July 2022. [Online]. Available: https://www.microsoft.com/en-us/research/publication/treehouse-a-case-for-carbon-aware-datacenter-software/

[60] C. L. Shane Miller, "Sustainability with rust," https://aws.amazon.com/blogs/opensource/sustainability-with-rust, 2022.