


Composing State Machine Replication

Caroline Martins Alves  [Universidade Federal de Santa Catarina | caroline.martins@posgrad.ufsc.br]

Matheus Antonio de Souza  [Universidade Federal de Santa Catarina | matheus.souza.m.a.s@grad.ufsc.br]

Thais Bardini Idalino  [Universidade Federal de Santa Catarina | thais.bardini@ufsc.br]

Odorico Machado Mendizabal   [Universidade Federal de Santa Catarina | odorico.mendizabal@ufsc.br]

 Department of Computer Science and Statistics, Universidade Federal de Santa Catarina, R. Delfino Conti, s/n, Trindade, Florianópolis, SC, 88040-900, Brazil.

Received: 14 April 2025 • **Accepted:** 10 October 2025 • **Published:** 04 December 2025

Abstract High availability is a fundamental requirement in large-scale distributed systems, where replication strategies are central in keeping applications operational despite a bounded number of failures. State Machine Replication (SMR) is one of the most widely adopted approaches for implementing highly available, fault-tolerant services, as it increases uptime while ensuring strong consistency. In recent years, research on SMR has yielded numerous variations tailored to enhance resilience, performance, and scalability. In this paper, we revisit SMR from a new perspective by introducing Composing State Machine Replication (CSMR), a method that enables fault-tolerant service composition. By composing SMRs, we promote the reuse of existing services to construct more complex and reliable systems. This modular approach fosters loosely coupled, flexible architectures, contributing to the theoretical foundations of SMR and aligning with common development practices in cloud computing and microservices. We formally define CSMR and demonstrate how composition can be used to extend existing SMR specifications with new features. For example, CSMR allows the semantics of a service operation to be extended by enabling different state machine replicas to execute complementary steps of the same operation. Additionally, SMR composition facilitates sharding and state partitioning by assigning disjoint state variables to separate SMRs. Beyond formalization, the paper provides illustrative examples of CSMR and introduces a high-level CSMR architecture that highlights the essential components, their responsibilities, and their interactions in supporting the composition process. To further demonstrate practicability, we present an API for building CSMR systems that combines RPC-based communication with declarative configuration in YAML format.

Keywords: Distributed Computing, Fault Tolerance, State Machine Replication, Composition

1 Introduction

Fault tolerance has become a key property for most distributed systems, ensuring correct operation despite the occurrence of failures. To achieve fault tolerance, replication strategies allow safe execution as a subset of correct replicas keep running even in the presence of a limited number of faulty replicas. In this sense, State Machine Replication (SMR) [Lamport, 1978; Schneider, 1990] is a widely adopted approach to implement highly available and strongly consistent fault-tolerant services.

In SMR, all replicas start from an identical state and process the same inputs deterministically and in the same order, ensuring consistent state transitions. By adhering to these design principles and benefiting from the ordering guarantees provided by the underlying communication protocols, such as atomic broadcast [Défago *et al.*, 2004] or consensus algorithms [Lampson, 1996; Lamport, 2001; Howard *et al.*, 2015], SMR has gained popularity and been implemented in many real-world systems. These include coordination services such as Apache Zookeeper [Hunt *et al.*, 2010] and Google Chubby [Burrows, 2006], as well as distributed storage systems such as GFS [Ghemawat *et al.*, 2003] and HDFS [Shvachko *et al.*, 2010], which rely on replicated state machines to maintain

consistent and highly available metadata. Other notable examples of SMR in practice include key-value stores like etcd [Etcd, 2013], and large-scale online services deployed by companies such as Twitter [Manhattan, 2014], Amazon [DeCandia *et al.*, 2007], and Facebook [Masti, 2021].

Over the past decades, extensive research has advanced SMR in several directions. Some works have developed protocols resilient to arbitrary and Byzantine faults [Castro and Liskov, 2002; Bessani *et al.*, 2014], while others have focused on enhancing fault tolerance through recovery and reconfiguration mechanisms [Castro and Liskov, 2002; Lamport *et al.*, 2010; Alchieri *et al.*, 2017]. From a performance standpoint, SMR's sequential execution of requests limits throughput and underutilizes multiprocessor architectures. Recognizing that some requests do not conflict (e.g., *write*(*x*, *v*) and *read*(*y*)), researchers have proposed Parallel State Machine Replication [Kotla and Dahlin, 2004; Kapritsos *et al.*, 2012; Marandi *et al.*, 2014; Mendizabal *et al.*, 2017; Batista *et al.*, 2022; Burgos *et al.*, 2021], allowing independent operations to be executed concurrently while maintaining consistency by partially ordering dependent ones.

Despite substantial research progress, the design and deployment aspects of SMR remain relatively unexplored. As service development has evolved from monolithic to service-

oriented and microservice architectures, SMR has remained a tightly coupled, single-service approach. This rigidity hampers distributed development, continuous deployment, and integration efforts.

To address this gap, this work explores an overlooked dimension: SMR composition.¹ By advancing the work in Alves *et al.* [2024], we propose Composing State Machine Replication (CSMR), a method that enables fault-tolerant service composition. By composing SMRs, we promote the reuse of existing services to construct more complex and reliable systems. The core idea is to build services by combining multiple SMR instances. From theorists' view, CSMR introduces novel perspectives, especially in modeling SMR operations and representing the combined service state. From a practical standpoint, composition aligns well with trends in cloud computing and microservices, where flexibility and modularity are needed. SMR composition promotes reuse and integration of existing implementations, enabling modular and extensible service design.

This article extends our earlier work [Alves *et al.*, 2024] by advancing the formalization of CSMR and broadening its scope. It introduces a precise definition of the model's semantics, refines cores concepts, and adds illustrative examples to show how composition can be applied in practice. The use cases have been updated to reflect more realistic scenarios, and a high-level architecture with an accompanying API is presented to demonstrate how CSMR can be implemented through declarative specifications.

The main contributions of this paper are:

- A formal definition of CSMR with composition operations for combining standard SMR components;
- A set of use cases supported by running examples that demonstrate how SMR composition enables the addition of new operations, the extension of operation semantics, and state partitioning;
- An RPC-inspired API and a declarative configuration approach in YAML for specifying services and their composition, paving the way for practical adoption of CSMR in microservices and cloud-based environments.

The remainder of the paper is organized as follows. Section 2 reviews related work, including SMR variants and the use of composition in different contexts. Section 3 defines and formalizes the SMR model. In Section 4, we introduce CSMR, including formal definitions and illustrative examples. Section 5 presents a high-level architecture for CSMR and a practical approach for declaring CSMR through YAML files. Finally, we present our conclusions in Section 6.

2 Related Work

Many researchers have proposed extensions to the traditional SMR model to meet the stricter requirements for modern applications concerning scalability, performance, and ease of use. Advances over the traditional SMR model enabled it to tolerate more severe fault models [Castro and Liskov,

2002; Bessani *et al.*, 2014] and improve resilience by incorporating recovery and reconfiguration protocols [Castro and Liskov, 2002; Lamport *et al.*, 2010; Alchieri *et al.*, 2017], among other context-specific improvements. This section reviews some of these variations, illustrating how researchers enhance performance by relaxing the sequential execution of the classic SMR model and improve scalability by exploring state partitioning. Additionally, we present approaches that facilitate the development of fault-tolerant services based on SMR. Finally, this section introduces the reader to different uses of composition, showcasing the versatility and potential developments of this approach.

2.1 State Machine Replication Extensions

In this section, we discuss approaches to improve SMR performance and scalability. The choice to present these variations is motivated by the fact that such SMR models increase concurrency without affecting replica consistency. Higher concurrency is also expected when considering composition. Further, we present some approaches to making SMR development easier. Since composition promotes modular development, reviewing approaches that simplify or make SMR development more transparent is essential.

2.1.1 Parallel and Scalable SMR

Parallelism and scalability are two different strategies for enhancing SMR performance. Parallelism allows the execution of multiple operations in parallel, benefiting from multicore architectures by executing independent requests simultaneously. Scalability, on the other hand, enables an increase in the load without substantially affecting the overall throughput in processing requests. Scalability is typically achieved through state partition, allowing replicas to operate over separate sets of data.

To the best of our knowledge, the first proposal of parallel execution in SMR is proposed in [Kotla and Dahlin, 2004]. In their proposal, the requests are delivered in the same order to all replicas using atomic broadcast. The replicas are augmented with a deterministic scheduler called parallelizer. Upon request delivery, the parallelizer dispatches independent commands to a pool of worker threads for parallel execution. Consistency across replicas is guaranteed once requests are delivered in the same order, and the parallelizer adopts the same scheduling policies to establish a partial order for requests' processing where dependent commands are serialized.

The work in [Mendizabal *et al.*, 2017] proposes a parallel SMR model using the same design principle as the parallelizer in [Kotla and Dahlin, 2004]. This parallel model introduces mechanisms to efficiently represent and calculate dependencies among commands, complemented by a lightweight scheduling mechanism. They reduced the dependency graph contention present in [Kotla and Dahlin, 2004], achieving higher throughput. Parallelism is achieved by defining separate queues, one per worker thread. Upon delivery of a batch of requests, the scheduler checks if commands in the batch conflict with pending commands in the worker thread queues. It then dispatches the batch to the appropriate queues, ensuring conflicting commands do not violate the execution

¹We use composition in the sense of constructing complex services by combining simpler, reusable components. This is distinct from other meanings, such as function or relation composition, or compositional verification.

order. The dependency check is very fast, as it uses compact bitmap structures to look up dependencies.

A parallel SMR using multicast groups was proposed in [Marandi *et al.*, 2014]. In this approach, the delivery of commands is partially ordered across the replicas by using separate multicast groups to send independent commands. Each group constitutes a different stream of commands delivered to each replica. This method eliminates the need for a scheduler within the replicas. Instead, it delegates to clients the decision of which multicast group to use based on application semantics. As a result, multiple worker threads can deliver and execute independent commands directly, reducing the overhead associated with parallelizing mechanisms.

Focusing on strategies to improve scalability, Bezerra *et al.* propose Scalable SMR (S-SMR) [Bezerra *et al.*, 2014]. Typically, in SMR, all replicas execute the same requests in the same order, so increasing the number of replicas does not increase throughput. S-SMR partitions the application state, allowing each command to access any combination of partitions, and employing a caching algorithm to reduce communication between partitions. The service state is partitioned and atomic multicast primitive is used to order commands within and across partitions. S-SMR also prevents command interleaves that violate strong consistency by implementing execution atomicity. This approach achieves scalable throughput and strong consistency without adding implementation complexity or limiting service commands. To evaluate S-SMR performance, the researchers developed the Eyrie library, which allows the transparent implementation of partition-state services. They then presented an application called Volery that implements the Zookeeper API using Eyrie. Comparing Volery against Zookeeper, they observed that Volery's throughput increased with the number of partitions. Volery was able to execute 250.000 commands per second, compared to Zookeeper's 45.000 commands under the same workload.

2.1.2 Transparent replication

Another aspect addressed in SMR development research is the ability to design and deploy replicated services. Although SMR is conceptually simple, building practical SMR-based services requires experience and a solid knowledge of fault tolerance, distributed programming, and reliable communication. For example, in [Chandra *et al.*, 2007], the authors implement a fault-tolerant database using the SMR approach. They report the challenges and complexities in the service development and attribute the difficulties encountered in converting algorithms into a practical, production-ready system. Additionally, they emphasize that specification changes during software development are expected. Therefore, an implementation should be malleable, but promoting changes to intricate code designed for particular fault-tolerant applications is cumbersome. In another work [Kirsch and Amir, 2008], authors mention that technical details, often regarded as engineering considerations, have considerable liveness implications and can dramatically impact performance.

Different strategies have been proposed to reduce the complexity of developing and deploying replicated systems, such as providing transparent replication. For instance, CRANE

[Cui *et al.*, 2015] is a parallel SMR system that transparently replicates general multi-threaded programs. Within each replica, CRANE intercepts the POSIX socket and the Pthreads interface and implements deterministic versions of their synchronizing operations. To ensure total order delivery of requests across replicas, CRANE runs a distributed consensus protocol for each incoming socket call (e.g., `accept()` or `recv()`). This guarantees that all correct replicas see exactly the same sequence of calls. CRANE schedules synchronization commands using deterministic multithreading (DMT), a technique that maintains a logical time advancing deterministically with each thread's synchronization.

OpenReplica [Altinbiken and Siner, 2012] offers transparent object replication, allowing clients to interact with components without being aware of their replication status. This implementation leverages the Paxos protocol, incorporating a lightweight version optimized with asynchronous events for better performance. To ensure clients can invoke replicated objects just as they would local ones, binary rewriting is employed. OpenReplica also supports dynamic changes to the location and number of replicas at runtime, maintaining consistency through a view change protocol during reconfiguration. Like OpenReplica, in [Pereira *et al.*, 2019], authors also provide transparency by replicating SMR objects without requiring developers to implement sockets or threads to establish process communication explicitly. However, while every new application initialized by OpenReplica launches a new set of SMR replicas, in Pereira's approach [Pereira *et al.*, 2019], independent applications can share the same consensus protocol. This design is advantageous for performance and suitable for applications running in shared infrastructure.

2.2 Compositional Methods

In this section, we present related work in the literature that explore composition in different scenarios.

A proposition on building software architecture as state machine composition using Domain-Driven Design (DDD) is proposed in [Perone and Karachalias, 2023]. Their modular approach allows them to compose independent sub-components to create more extensive systems. In addition to the design, the Crem library is presented, which provides a concrete state machine implementation that is compositional and representable. As a result, the work provided a way to compose and represent software architectures in a simplified way, proposing the use of the Crem library, which is capable of keeping the graphical representation of the architecture updated as the system evolves and changes. The library uses Haskell's tools to specify allowed and forbidden transitions and encode state machine characteristics and the domain they represent. This approach relates to ours by allowing system expansion through composition, as they accept new state machines to be added to the architecture like our model accepts new replicated operations to compose SMRs.

Another approach explores composability inside a class of language acceptors [Dang *et al.*, 2004], including automata with a one-way input tape. The work is motivated by automated design issues in e-services and web services. They analyze generalizations to automata with unbounded storage, showing problems of decidability and undecidability related

to composability. A composable system $(A; A_1, \dots, A_r)$ will be the one that every string accepted by A can be attributed to its forming automata in a way that each subsequence assigned to an automaton is accepted. The notion of a k -lookahead delegator is also introduced, which can determine which automaton to assign the current symbol based on the automata's actual states, local information, and k -lookahead symbols. As a result, the work solved some problems in the literature and generalized previous results about composability and k -delegability. This work relates to ours in proving dynamic composition, as they provide an approach that facilitates the process of composing e-services/web services. Our work also aims to provide a simple way to add new services to SMRs.

The work in [Lynch and Musco, 2022] is focused on an algorithmic theory of brain networks, based on synchronous, stochastic Spiking Neural Networks (SNN) models. They present a more general computational model for SNN based on directed graphs to map a set of neurons (nodes) V with a set of synapses (edges) E , saving the potential for each neuron at some discrete time. They also present a hiding operator that can change the classification of some output behavior of an SNN as internal and a compositional operator that allows modeling SNNs as a combination of smaller SNNs. They proved that the behavior of a compositional network depends exclusively on the external behavior of the component networks. In addition, a notion of a problem solved by SNN is introduced, which helps to understand how these two operators (compositional and hiding) affect it. As a result, they present a formal, mathematical foundation for modeling SNN, define composition and hiding operators, besides proving fundamental theorems. This approach is related to the proposal in this work, as we also want to increase the modularity of our model, this is evident when they show the example of composing four logic gates in an XOR network, highlighting the possibility of increasing the complexity of a network by composing it with smaller networks.

To the best of our knowledge, no existing approaches explicitly address the composition of SMR, apart from our prior work [Alves et al., 2024]. Some proposals, however, while not explicitly framed as compositions, align with some of our ideas and can be reinterpreted within the CSMR framework. For instance, Bezerra et al. [2014] introduce S-SMR (see Section 2.1.1 for details), which partitions the state-machine state, allowing each command to access different combinations of partitions. As we illustrate in Section 4.1.3, state partitioning or sharding can be conveniently supported by SMR composition. In Xavier et al. [2020], the authors propose a decoupled logging approach for SMR. They introduce lightweight processes, called loggers, into the set of active replicas. Client requests are totally ordered and delivered both to SMR application replicas and to loggers. The loggers handle the maintenance and retrieval of command logs, offloading this responsibility from the application replicas. Scharf et al. [2023] further extend this idea by combining P-SMR and S-SMR. Their approach improves throughput by enabling parallel delivery and processing of requests from different applications at the logger processes. Our contribution with CSMR is to provide a precise and formalized model for SMR composition. A well-founded CSMR model can foster the development of new applications and composition strate-

gies in the context of active replication. The aforementioned works reinforce both the relevance and the potential of SMR composition.

3 State Machine Replication

State Machine Replication (SMR) is a general approach to implementing fault-tolerant services by replicating servers and coordinating client interactions with server replicas [Lamport, 1978; Schneider, 1990]. A state machine defines the service and consists of *state variables* that encode the state machine's state and a set of *operations* that change the state. The execution of an operation may (i) read state variables, (ii) modify state variables, and (iii) produce a response for the operation (*i.e.*, the output).

An SMR provides clients with the abstraction of a highly available service while hiding the existence of multiple replicas. In order to ensure that the execution of an operation will result in the same state changes and results at different replicas, the operations must be executed in the same order by the replicas and be *deterministic*: changes to the state and response of an operation are a function of the state variables, the operation accesses, and the operation itself. Therefore, if servers start from the same state and execute operations in the same order, they will produce the same state changes and results after executing each operation.

By starting from the same initial state and executing totally ordered commands deterministically, SMR ensures strong consistency. As widely noted in the literature [Lynch, 1996; Castro et al., 1999; Berger et al., 2021], this level of consistency aligns with linearizability, a well-established and widely used correctness criterion in concurrent and distributed systems [Herlihy and Wing, 1990; Sela et al., 2021]. Linearizability ties atomic request execution to real-time: the effects of a request must be observable by any subsequent invocation. In other words, a system is linearizable if there is a way to reorder the client commands in a sequence that (i) respects the semantics of the commands, as defined in their sequential specifications, and (ii) respects the real-time ordering of commands across all clients [Attiya and Welch, 2004].

Figure 1 shows a graphical representation of SMR and its main components. Requests issued by clients are handled by the client proxy (step ①), which multicasts the requests' operations to all servers belonging to the group of replicas S (step ②). Each state-machine replica delivers and executes the requested operation independently, updating internal state variables and producing an output (step ③). Once the operation is executed, the replica immediately replies to the proxy with the request identifier and the respective operation result (step ④). The proxy is responsible for receiving the request's output. It waits for the first response from one replica, forwards the request's output to the client, and discards the following responses to the same request, thus preventing the client from receiving duplicate responses (step ⑤).

Client proxies translate client invocations into requests that include an operation identifier and its arguments. The proxy also implements the ordering and agreement layer, represented by the *multicast* module. This layer ensures requests

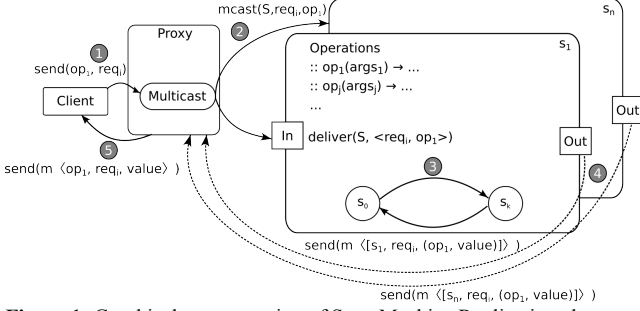


Figure 1. Graphical representation of State Machine Replication elements.

are delivered in a total order across state machine replicas. Each service replica implements operations op_1 to op_k , representing the possible service behaviors.

In this SMR representation, processes communicate by message passing, using either one-to-one or one-to-many communication. One-to-one communication is achieved using primitives $\text{send}(m)$ and $\text{receive}(m)$, where m is a message. If a sender sends a message enough times, a correct receiver will eventually receive the message. One-to-many communication is based on atomic multicast, whose main primitives are $\text{mcast}(g, m)$ and $\text{deliver}(g, m)$, where g refers to the multicast group.

Considering a single group g , the multicast resembles the atomic broadcast, where every message is delivered to all destination processes in the same order. When considering multiple groups, groups may overlap, *i.e.*, some pairs of messages have different destination sets with a nonempty intersection.

Atomic multicast ensures the following properties [Pacheco et al., 2022]:

- agreement*: if a process in g delivers m , then all correct processes that subscribe to g deliver m ;
- validity*: if a correct process p multicast m to g then all correct processes that subscribe to g deliver m ; and
- atomic global order*: Define relation \prec on the set of messages processes deliver as follows: $m \prec m'$ iff (i) there exists a process that delivers m before m' ; or (ii) m' is multicast after m is delivered at some destination, in real-time. The relation \prec is acyclic.

As discussed in [Pacheco et al., 2022], assuming weaker properties for atomic multicast, such as *local total order* and *global total order*, may violate linearizability in some scenarios. For instance, assuming the execution illustrated in Figure 2, adapted from Pacheco et al. [2022], the two commands $\text{write}(0, v_0)$ and $\text{write}(1, v_1)$ insert key-value pairs for keys 0 and 1, respectively. Moreover, $\text{write}(0, v_0)$ precedes $\text{write}(1, v_1)$ in real-time, that is, $\text{write}(0, v_0)$ finishes at client b before $\text{write}(1, v_1)$ starts at client c . Now consider a concurrent range query $\text{range}(0, 1)$ that tries to read previously inserted pairs for keys 0 and 1. The range query accesses both partitions, defined by groups g_1 and g_2 , and is delivered and finishes at $r_1 \in g_1$ before $\text{write}(0, v_0)$ is delivered, and is delivered at $r_2 \in g_2$ after $\text{write}(1, v_1)$ ends. When assuming *global total order*, the prefix order property of atomic multicast is not violated since all processes that deliver the same messages, do it in the same order. The atomic multicast acyclic order condition is not violated either: $\text{write}(0, v_0)$ and $\text{write}(1, v_1)$ are not directly related since they are delivered at differ-

ent groups, and thus, $\text{range}(0, 1) \prec \text{write}(0, v_0)$ at r_1 and $\text{write}(1, v_1) \prec \text{range}(0, 1)$ at r_2 . Although the \prec relation is acyclic, $\text{write}(1, v_1) \prec \text{range}(0, 1) \prec \text{write}(0, v_0)$, it does not account for the real time order in which $\text{write}(0, v_0)$ precedes $\text{write}(1, v_1)$. Even though $(0, v_0)$ is inserted before $(1, v_1)$ in the key-value store, the range query only returns the pair $(1, v_1)$, violating linearizability.

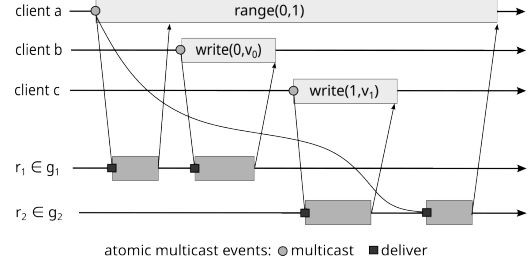


Figure 2. An execution that violates linearizability (*atomic global order violation*).

For ensuring linearizability, the *atomic global order* is required. The *atomic global order* property introduces two aspects: it relates atomic multicast primitives in real time and it relates messages multicast to possibly disjoint destinations. Notice the trace execution in Figure 2 satisfies weaker atomic broadcast variations but does not satisfy the *atomic global order* property. The $\text{write}(1, v_1)$ is multicast by client c after $\text{read}(0, 1)$ is delivered by the replica at group g_1 , it follows from *atomic global order* that $\text{read}(0, 1) \prec \text{write}(1, v_1)$; and from the delivery order of requests at g_2 , $\text{write}(1, v_1) \prec \text{read}(0, 1)$, which leads to a cycle and violates atomic multicast's global total order. Therefore, such an execution is not allowed given our multicast assumptions.

Other assumptions may be required to implement atomic multicast, such as quorum intersection (*e.g.*, $n \geq 2f + 1$) and partial synchrony. As demonstrated in Lamport [2006], a state-machine implementation that can tolerate the failure of f computers needs only $f + 1$ copies of the machine's state, although $2f + 1$ computers are required to achieve consensus on the state-machine commands. This means that no quorum intersection is required at the service level, and we do not explicitly address these assumptions as we consider atomic multicast a building block.

We assume the crash failure model and exclude malicious and arbitrary behavior (*e.g.*, no Byzantine failures). A process is *correct* if it does not crash or *faulty* otherwise. We assume the existence of up to f faulty state-machine replicas among a total of $n = f + 1$ servers. The state-machine replicas do not participate in the execution of the atomic multicast protocol itself; they only deliver multicast requests and execute them. Each server either responds (a *correct* one) or remains silent (a *faulty* one), so clients can trust any response received from a state-machine replica to reflect the execution of a valid, totally ordered request.

3.1 Use cases

Next, we define three simple applications as use cases. Firstly, we discuss these examples in general terms, illustrating the applications API and their operations. Then, they serve as examples for the demonstration of our SMR formalization.

We return to these use cases in Section 4 to illustrate how to compose state machine replicas.

Lock service: The first use case illustrates a lock service, a fundamental mechanism for maintaining data consistency in concurrent systems. A lock coordinates access to shared resources by ensuring that only one client can access a resource at a time. To interact with a shared resource safely, a client must first enter a critical section by invoking the *acquire(lock)* operation. After completing its task, the client must release the resource by calling *release(lock)*. If another client has already acquired the specified *lock*, the *acquire(lock)* operation blocks until the lock is released. This coordination prevents simultaneous access to shared resources, thereby preserving consistency and avoiding race conditions. The API for these operations is presented next.

```
boolean acquire(string key)
boolean release(string key)
```

Key-value store: A key-value service implements a data storage composed of key-value pairs. Clients can access the service to insert new pairs $\langle key, value \rangle$ or read values associated with the keys. Updating operations insert the key and value into the storage structure as a pair $\langle key, value \rangle$. If the key is already in the table, the new pair overwrites the previous one. To read a value, clients may execute the *get* operation that returns the value associated with the *key*, if any, or a *null* value. In this example, both *key* and *value* are strings. The key-value service operations are described next.

```
string get(string key)
void put(string key, string word)
```

Logging service: A logging service provides durability by allowing persistent storage of objects. Logs are helpful for many applications and may improve fault tolerance or security. For instance, logs may keep track of commands executed by a process for rollback or recovery purposes. Also, monitoring services can register the system's events or users' activity in a log for further analysis (e.g., forensics, diagnosis, or *post mortem* analysis). The log service implements an append-only structure to record new entries, keeping an ordered sequence of log entries. The operations performed by this service are *append(object entry)* to append a new entry to the log (e.g., an incoming request or an executed operation), *retrieve(int first, int last)* to retrieve a sequence of log entries from the interval ranging from *first* to *last* indexes, and *truncate(int index)* that removes from the log all entries before *index*. In this example, log entries are represented by a generic type, which we call *Object*. An object can be a string, integer, float, or a custom, possibly structured data type. Indexes are *int* values, and an array of objects represents a sequence of logged entries. The API for these operations is presented next.

```
boolean append(Object entry)
Object[] retrieve(int first, int last)
boolean truncate(int index)
```

3.2 Definitions

To formalize state machine replicas, we consider the relations between the service replicas and the operations they execute.

We abstract the client's role and the delivery of requests to the replicas. This means the replicas receive a totally ordered sequence of operation requests as input and execute each operation in the order it arrives. The total order and agreement of requests across replicas are commonly achieved using atomic multicast or consensus protocols. This feature, as well as forwarding requests' output to the client, is the responsibility of the client proxy, which is omitted in this section.

Now we present our SMR formalization. We define a *replicated service* as a relation between the operations implemented by the service and the replicas that implement such operations.

Definition 1 (Replicated service). Let O be a nonempty set of operations and M be a nonempty set of replicas, where $|M| = f + 1$. Then $R = O \times M$ is a relation from O to M that represents a replicated service.

Observe that, according to this definition, all replicas in M execute all operations in O . This is a necessary condition for implementing traditional SMR, where all replicas start in the same state (ensured at service initialization), receive the same requests in total order (ensured by the client proxy), and execute the requests as they arrive. To enable the composition of SMR, we will revisit this definition in Section 4, allowing replicas to exhibit different behaviors.

Considering the use cases in the previous section, we can formalize them as state machine replicas as follows. Assume that the set of replicas is $M = \{s_1, s_2\}$.

SMR 1 - Lock service: Let $O_1 = \{acquire, release\}$. The relation between operations and replicas, as defined by the *replicated service* definition, is: $R_1 = \{(acquire, s_1), (acquire, s_2), (release, s_1), (release, s_2)\}$.

SMR 2 - Key-value store: Let $O_2 = \{get, put\}$. The relation between operations and replicas is: $R_2 = \{(get, s_1), (get, s_2), (put, s_1), (put, s_2)\}$.

SMR 3 - Logging service: Let $O_3 = \{append, retrieve, truncate\}$. The relation between operations and replicas is: $R_3 = \{(append, s_1), (append, s_2), (retrieve, s_1), (retrieve, s_2), (truncate, s_1), (truncate, s_2)\}$.

To provide fault tolerance, at least one correct replica must be available and capable of executing any requested operation. As stated by Definition 1, all replicas perform all operations, *i.e.*, a single correct replica can execute any operation. Thus, assuming that up to f replicas can fail, we must have at least n replicas in the system, where $n = f + 1$. In the previous use cases, $M = \{s_1, s_2\}$, then $n = |M| = 2$, which tolerates up to $f = 1$ faulty replica.

We previously described the relations between operations and replicas and the number of replicas required to ensure fault tolerance in SMR. Now, we extend this formalization by considering the operations with arguments and the operation execution. We consider a set A of possible arguments and redefine the set of operations O as O to incorporate these arguments.

Definition 2 (Operations with arguments). Let O be a nonempty set of operations and let A be a nonempty set of

arguments, we define $O = O \times A$ as the set of operations with arguments.

Next, we show examples of operations with arguments for SMRs 1, 2, and 3. Note that the set A from Definition 2 can be defined to take as many arguments as necessary.

Example 1. When performing operations in SMR 1, we define an alphabet $A = \{a, b, \dots, z\}$, and A^* as the set of all strings over A . Then, possible pairs of elements of $O = O_1 \times A^*$ are: $(acquire, "lock")$, $(release, "lock")$, etc.

Example 2. When performing operations in SMR 2, we define an alphabet $A = \{a, b, \dots, z\}$, and A^* as the set of all strings over A . Let $A_1 = A^*$ and $A_2 = A^* \times A^*$. In this SMR, the set O_2 has two operations with different arguments, so we define the first as $O_{get} = \{get\} \times A_1$ and the second as $O_{put} = \{put\} \times A_2$. Possible pairs of elements of O_{get} are $(get, "firstkey")$, $(get, "secondkey")$; and of O_{put} are $(put, ("firstkey", "firstval"))$, $(put, ("secondkey", "secondval"))$.

Example 3. When performing operations in SMR 3, the set O_3 has three operations with different arguments, so we consider the set $A_1 = \Omega$, where Ω represents a universal set of objects; $A_2 = \mathbb{Z}^+ \times \mathbb{Z}^+$ and $A_3 = \mathbb{Z}^+$. We define the operations with arguments as follows: $O_{append} = \{append\} \times A_1$; $O_{retrieve} = \{retrieve\} \times A_2$; $O_{truncate} = \{truncate\} \times A_3$. Possible elements of O_{append} , $O_{retrieve}$, and $O_{truncate}$ are: $(append, (put, ("key", "val")))$, $(retrieve, (10, 200))$, and $(truncate, 10)$, respectively.

Finally, the replicated service from Definition 1 can be updated to consider operations with arguments.

Definition 3 (Replicated service with arguments). Let O be a nonempty set of operations with arguments and M be a nonempty set of replicas. Then $R = O \times M$ is a relation from O to M that represents a replicated service.

So far, we have formally defined replicated services as a relation between operations and replicas. As we present next, we can also define an execution of operations as a relation between operations and the response values.

Definition 4 (Execution). Let O be a nonempty set of operations with arguments, U be a nonempty set of outputs. Then $E \subseteq O \times U$ is a relation from O to U that represents the execution of the operations and their corresponding outputs.

Note that in the execution, the relation between operations and outputs is defined as a subset of the Cartesian product. This is because not all operations will produce all possible results. For instance, the pair $((truncate, 3), true)$ is part of a logging service execution in case of success, and $((truncate, 3), false)$ is part of the execution in case of error, while $((truncate, 3), 5)$ is an invalid execution.

Example 4. Consider the SMR 3 with operations with arguments as defined in Example 3. When O_{append} or $O_{truncate}$ are invoked, it could produce the relations $E \subseteq O_{append} \times \{true, false\}$ and $E \subseteq O_{truncate} \times \{true, false\}$, respectively. On the other hand the $O_{retrieve}$ could produce $E \subseteq O_{retrieve} \times$

$\{["firstobject", "secondobject", "thirdobject"]\}$. Thus, these are possible representations of executions over *append*, *truncate* and *retrieve* operations in a logging service.

When a replica executes an operation, it generates an output representing the result of that execution. We now formally define the structure of this output.

Definition 5 (Output). Let $s_n \in M$ be a replica, req_i be a unique identifier associated with each client request, $op \in O$ be an operation, and $u \in U$ be the response value from a requested operation. We define $w = [s_n, req_i, (op, u)]$ as an *output* of the clients' request.

Example 5. When a client sends a request $get("firstkey")$ to SMR 2, the proxy will generate a unique identifier for this request (e.g., 1234) and multicast the request to all replicas in M (i.e., s_1 and s_2). The replicas will execute the operation and produce the following outputs: $w_1 = [s_1, 1234, (get, "foobar")]$ and $w_2 = [s_2, 1234, (get, "foobar")]$.

Observe that the proxy will receive at least $|M| - f$ responses, but the output can be forwarded to the client as soon as the proxy receives the first response.

Definition 6 (History). Let a *history* H be a run of an SMR represented by a finite sequence of request executions E totally ordered across replicas.

Each execution E includes the requested operation and its output, denoted by $deliver(\gamma, o_i)$ and $send(m\langle o_i, u_i \rangle)$ (see the respective $deliver(S, op_1)$ and $send(m\langle op_1, output \rangle)$ illustrated in Figure 1). The arguments represent the multicast group γ , the operation o_i , and a message m containing $o_i \in O$ and its output $u_i \in U$.

From the SMR definition [Lamport, 1978; Schneider, 1990], all replicas start in the same initial state, operations are deterministic, and all replicas execute the same requests in total order. By ensuring these requirements, SMR provides strong consistency, often referred to in the literature as linearizability [Herlihy and Wing, 1990]. An implementation is linearizable if each of its executions is linearizable. Intuitively, for each operation invocation, there is a unique point within the real-time interval defined by the invocation and response of the operation, and these linearization points induce a valid sequential execution. Thus, state machine replicas essentially behave as if their operations happened atomically under any concurrent execution.

In the traditional SMR, every request in H comes from a single multicast group γ , ensuring that requests are ordered across replicas. As operations are sequentially executed as they arrive, a typical SMR implementation achieves linearizability. Modern SMR models [Kotla and Dahlin, 2004; Marandi et al., 2014; Mendizabal et al., 2017; Batista et al., 2022; Burgos et al., 2021], may enhance concurrent operations execution by relaxing the delivery or execution order, for instance, by adopting protocols other than atomic multicast or allowing the execution of non-conflicting operations in parallel. Here, we assume an SMR implementation is correct and preserves strong consistency. Discussing the correctness of a particular SMR implementation is beyond the scope of this paper, as we consider SMR services as building blocks.

In this section, we have formalized state machine replication using relations between operations, replicas, arguments, and outputs. Next, we use this formalization to reason about composition in state machine replication.

4 Composing State Machine Replicas

As observed in the literature, composition can be a powerful strategy in software development. For example, it can facilitate reuse [Dang et al., 2004; Lynch and Musco, 2022] by easily integrating previously implemented components into new software, enhance maintenance by allowing parts of complex software to be updated with low interference to other system components, and extend the features and functionalities of an already implemented service.

In this section, we propose the implementation of SMR through composition. The general idea is to enjoy the properties of existing SMRs and combine different state machine replicas to build an extended service. By extending a service, we mean incorporating additional functionalities that a single SMR cannot provide, augmenting the service state as different replicas may have different state addresses, or even allowing distinct operations to be performed over the same request.

When reasoning about Composing State Machine Replication (CSMR), the first observation is that replicas do not need to implement the same set of operations. Instead, the operations of interest can be implemented by different state machine replicas, which are then selected to compose a new, *composable replicated service*.

By allowing replicas to implement only part of the required operations, it becomes necessary to determine which replicas implement each operation of interest. Next, we define a way of obtaining this information.

Definition 7 (Replication set). Let R be a relation from O to M . Then, $R(x) = \{s_i \in M : (x, s_i) \in R\}$ is the set of all replicas that execute operation $x \in O$.

The *replication set* plays an important role in directing client requests to replicas capable of executing them. Unlike traditional SMR, where all client requests are broadcast to all service replicas, the proxy now utilizes the replication set to map each client request specifically to the target replicas capable of executing it.

When considering operations with arguments, if we are interested in obtaining information on which replicas perform a specific operation from set O , we can redefine $R(x)$ as follows.

Definition 8 (Replication set with arguments). Let R be a relation from O to M , with $O = O \times A$. Then, $R(x) = \{s_i \in M : ((x, a), s_i) \in R, \text{ with } a \in A\}$ is the set of all replicas that execute the operation with arguments $x \in O$.

Next, we define a *composable replicated service* as an extension of Definition 1. Here, replicas may execute only a subset of operations in O , which is achieved by considering the relation R as a subset of $O \times M$.

Definition 9 (Composable replicated service). Let O be a nonempty set of operations, M be a nonempty set of replicas,

and f be a non-negative integer. Then $R \subseteq O \times M$ is a relation from O to M that represents a replicated service, where $|R(x)| = f + 1$ for all $x \in O$.

We can have a similar generalization for Definition 3 to define *composable replicated services with arguments*. It simply requires replacing O with O in the relation in Definition 9.

In a traditional SMR, all replicas perform the same operations, ensuring that $|M| = f + 1$ to tolerate up to f faulty replicas. However, in SMR composition, different replicas in M may perform different operations in O . To ensure fault tolerance in this context, it is necessary to guarantee a replication factor according to the *replication set* defined in Definition 7. This means a minimum number of correct replicas is required per operation, considering up to f faulty replicas in $R(x)$. Therefore, we have $|R(x)| = f + 1$ for each operation $x \in O$.

Assuming $f = 0$ yields a replicated service with $|R(x)| = 1$. At first glance, this case may seem unusual, as it represents a replicated service without actual replication. Still, the formulation is valid, as a single pair (op, s) suffices when s is assumed correct. Allowing $f \geq 0$ provides greater generality, since different operations may require different replication factors: some may rely on a single replica if assumed fault-free, while others may demand larger values of f to ensure reliability.

4.1 Composition strategies

Composing SMR can provide the same guarantees as a traditional SMR while introducing greater flexibility in the operations performed by the replicas. Modular composition allows for the addition of new operations to a service or the implementation of data partitioning, as the service state can be divided, with operations accessing disjoint partitions.

For example, a developer can enhance a legacy SMR application by combining the operations of the existing state machine replicas with new ones provided by a set of replicas implementing additional features. This kind of composition is common in the development of web services or microservices. Similarly, design diversity [Avizienis and Kelly, 1984] can be facilitated through composition, as different implementations of the same operations can coexist within the replicated system. In this scenario, all replicas implement the operations differently while still adhering to the design requirements. Regarding scalability and performance, a service can implement state partitioning [Bezerra et al., 2014]. The service state is divided into partitions so that each variable defining the state belongs to a unique partition. The composition then combines replicas operating over distinct partitions. Thus, replicas hold a partial view of the system state, and although all replicas implement the same operations, they manipulate different state variables.

Definition 10 (Composition). Let O_1 and O_2 be sets of operations, M_1 and M_2 be sets of replicas, and let $R_1 \subseteq O_1 \times M_1$ and $R_2 \subseteq O_2 \times M_2$ be two *composable replicated services* as in Definition 9. Then, $R = R_1 \cup R_2$ is a *composition*, where $O = O_1 \cup O_2$ and $M = M_1 \cup M_2$. Moreover, let f_1 and f_2 be the upper bounds on faulty servers for R_1 and

R_2 , respectively. Then, the overall upper bound of R is $f = \min\{f_1, f_2\}$.

In Definition 10, both R_1 and R_2 are replicated services, which implies that there exist upper bounds f_1 and f_2 of faulty servers that each replicated service can tolerate. However, if failures are “well-behaved” in the sense that up to f_1 of them happen in replicas from M_1 and up to f_2 of them happen in replicas from M_2 , this new composition can tolerate up to $f = f_1 + f_2$ faults.

Furthermore, Definition 10 demonstrates composition using two SMR instances, however, this definition generalizes to n SMRs as $R_1 \cup \dots \cup R_n$.

Definition 11 (Collectible Outputs). Let $w_i = [s_i, req, (op, u)]$ be an output from replica s_i for the request req . We define the set of collectible outputs as $D = \{w_1, \dots, w_n\}$ representing the collection of all outputs generated for a given request.

The *collectible outputs* play a central role in the compositional model, as a single client request may trigger additional, potentially complementary operations as part of the composition. Consequently, the resulting outputs may differ in content. To handle this, the proxy must implement a reduction or selection function that determines the appropriate output to forward to the client. Since this function depends on application-specific semantics, we revisit the topic in the section on composition types, where we present a concrete strategy for implementing the selection mechanism in our illustrative use cases.

Given Definition 10, note that the set of operations and machines available in a composed SMR can be augmented. Composition can occur in various ways: by adding new operations to an existing SMR, by assigning different semantic executions to an already defined operation, or by partitioning the service state across replicas (e.g., implementing sharding). While other composition schemes might be envisioned, this section focuses on adding new operations, extending the operations’ execution, and state partitioning. Next, we present a formal definition and detail these types of composition.

4.1.1 Adding SMR operations

This type of composition allows the addition of operations to a previously defined SMR. Figure 3 illustrates how extending operations from an SMR may occur. s_1 implements only operations op_1 and op_2 , s_2 implements op_1 and op_3 , and s_3 implements op_2 and op_3 . When a client issues operation $op_1(args_1)$, the proxy multicasts op_1 to servers s_1 and s_2 belonging to group g_1 as they implement this operation. Similarly, a request for $op_3(args_3)$ is multicast to group g_3 , i.e., servers s_2 and s_3 . In this illustration, clients see the service as composed of operations op_1 to op_3 , regardless of which replicas implement each operation. This setup tolerates up to 1 faulty replica, as at least two replicas implement each operation. For example, if s_1 crashes, there is still at least one correct server implementing op_1 , op_2 , and op_3 .

Next, in Example 6 we combine SMR 1 with SMR 2. Note that both operations from the O_1 and O_2 sets are included in the composing SMR.

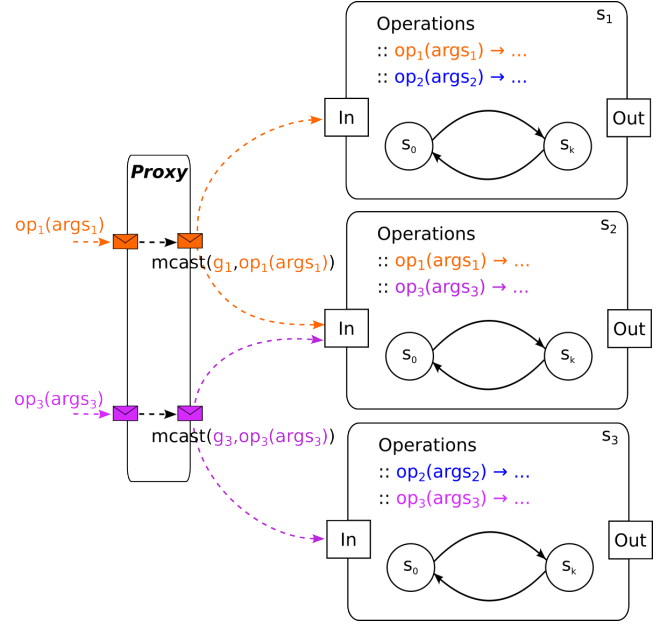


Figure 3. Combining SMR operations and replicas.

Example 6. Now we define an improved key-value store that performs operations related to the storage and retrieval of data, as well as synchronization operations to concurrency control. Let $O_1 = \{get, put\}$ and $O_2 = \{acquire, release\}$. Let $M_1 = \{s_1, s_2, s_3\}$ and $M_2 = \{s_4, s_5, s_6\}$. Consider the relations $R_1 \subseteq O_1 \times M_1$ and $R_2 \subseteq O_2 \times M_2$, and replication sets $R_1(x_1)$ and $R_2(x_2)$ with $|R_1(x_1)| = f + 1$ and $|R_2(x_2)| = f + 1$, for each $x_1 \in O_1$ and $x_2 \in O_2$. Thus, we can compose the replicated service by performing $R = R_1 \cup R_2$, or, in other words, a key-value store with a lock service.

Remark (Disjoint composition). From Definition 10, if $O_1 \cap O_2 = \emptyset$ and $M_1 \cap M_2 = \emptyset$, then the composition R is fully partitioned. This condition is expected when extending legacy services or combining services from separate vendors. An SMR implemented by a third party can even be hosted in a separate infrastructure, which might lead to disjoint operations and replicas across the SMRs used in the composition.

Remark (Joint operations and joint replicas). From Definition 10, if $O_1 \cap O_2 \neq \emptyset$ or $M_1 \cap M_2 \neq \emptyset$, then the composition R is not fully partitioned. There are at least two practical cases to illustrate here. First, different sets of replicas may implement the same operations. An example is geo-replicated services, where distant replicas deploy the same service (i.e., the same set of operations) to reduce latency, allowing clients to receive answers from the nearest replicas first. The second example is when the same set of servers runs multiple services (i.e., different sets of operations). This situation is expected when deploying different SMR services in a shared infrastructure, such as those typically found in cloud or container-based applications.

4.1.2 Extending SMR operations’ execution

This composition strategy supports multiple implementations for the same operation invocation. The goal is either to provide complementary functionalities for a given operation or to

implement the same functionality in different ways. The latter approach is commonly applied in design diversity [Avizienis and Kelly, 1984] to improve fault tolerance. Figure 4 illustrates this concept, where replicas s_2 to s_m (gray servers) provide alternative versions of operations op_1 and op_2 , denoted as op_1' and op_2' , respectively. For instance, when a client requests the execution of op_1 , the proxy multicasts the request to all the serves in g_1 and g_1' , as they are involved in handling op_1 and op_1' . Although the white and gray replicas are all triggered, the gray replicas execute a different implementation, op_1' , which may result in different state updates and outputs. Similarly, when a client calls op_2 , all servers receive the request, but s_2 to s_m execute a different code from s_1 to s_n for this operation.

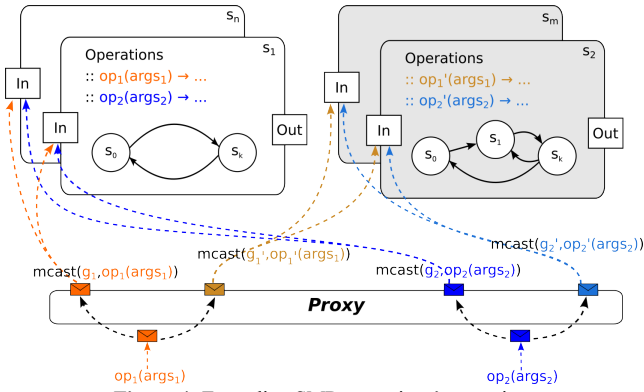


Figure 4. Extending SMR operations' execution.

This type of composition alters the behavior of an operation by executing n versions of the same operation whenever it is invoked. Each version of the operation affects only the individual states of the replicas that implement that specific version. Consequently, the outputs generated by replicas from different versions can also be different.

From Definition 11, D is the set of all possible outputs for the n versions of a certain operation. To handle multiple outputs, the proxy must implement a deterministic function $f(D)$ to decide which output $u \in D$ should be returned to the client. For instance, if the n versions of the operation are used for design diversity purposes, $f(D)$ could be a majority function. As another example, suppose that the n versions implement different heuristics to estimate a value, such as finding the shortest path between two nodes in a graph. In this case, $f(D)$ would return u_i if it is the minimal value in D . In this example, the function behavior is analogous to a reduction function in the map/reduce paradigm or a scatter/gather design pattern. Considering that some replicas may crash, $f(D)$ should be able to return a value based on a subset of outputs given by a quorum of correct replicas. For instance, $f + 1$ matching outputs are sufficient under silent faults (e.g., when assuming crash or fail-stop failures), while a strict majority (e.g., $2f + 1$) may be required to prevent conflicting outputs.

Example 7. Now we define a composition of SMR 2 and SMR 3 to extend the key-value store service with logging functionality. The operations put and get from O_2 were introduced in the SMR 2 definition. This composition overrides them in SMR 3 to behave as the $append$ method. When a client invokes $put("k", "v")$, the proxy multicasts this request to both SMR 2 and SMR 3 replicas. The replicas in

SMR 2 associate a value " v " with the key " k ", as usual. For the same request, the replicas in SMR 3 invoke the $append$ method, passing the object $put("k", "v")$ as an argument, thereby appending the operation to a log of operations, i.e., $(append, (put, ("k", "v"))) \in O_{append}$.

This example demonstrates the need for an output selection function to determine the appropriate response to return to the client – specifically, the output from the kv-store. For instance, consider a client request with $req = 10$ invoking the get operation. The proxy forwards it to both SMR 2 and SMR 3. The resulting set of collectible outputs would be $D = \{[s_1, 10, ("get", "value")], [s_2, 10, ("get", "value")], [s_3, 10, ("get", "value")], [s_4, 10, ("append", true)], [s_5, 10, ("append", true)], [s_6, 10, ("append", true)]\}$. To select the output of the " get " operation, the proxy can implement a selection function $f : D \rightarrow D$. This function will filter the output as follows: $f(D) = \{u \in D : s_n \in R(get), op = get\}$, in other words, by operation type (i.e., get) and source of replica group (i.e., $\{s_1, s_2, s_3\}$).

This particular example imitates the approach presented [Xavier et al., 2020; Scharf et al., 2023], where the authors proposed decoupled logging services for SMR. Although the authors had not intentionally composed SMRs at that time, their proposals fit well in this context.

4.1.3 Argument Partition

This composition strategy partitions a large set of data into subsets that can be stored and managed independently by different state machine replicas. The operations implemented by the SMRs are the same, but they access disjoint variables. This approach can increase scalability and performance by allowing horizontal growth and faster access to partitioned data from different replicas in parallel.

Figure 5 illustrates state partitioning by arguments. All replicas execute the operation op_1 , but the arguments they handle differ. When an operation involves arguments $args_{11}$, requests are broadcast to group g_1 so servers s_1 and s_2 handle the invocation. Analogously, when the arguments $args_{13}$ are passed in the operation request, the proxy broadcasts the request to group g_3 , with servers s_2 and s_3 handling it.

Example 8. Here we recall SMR 2 to compose SMRs and implement a key-value store service with partitioned state. Let $A = \{a, b, \dots, z\}$ be an ordinary alphabet. We define $A^n = A \times A \times \dots \times A$ to be the set of all words of size n . We can partition A^n into subsets of words starting with each letter of the alphabet as follows: $A_\alpha^n = \{\alpha\} \times A \times \dots \times A$, $\alpha \in A$. Therefore, we have: $A^n = A_\alpha^n \cup A_b^n \cup \dots \cup A_z^n$. Consider $O = \{get, put\}$, $A_1 = A_\alpha^n$, $A_2 = A_\alpha^n \times A^*$. We can now define operations with restricted arguments. For instance, $O_{get} = \{get\} \times A_1$ and $O_{put} = \{put\} \times A_2$ represent operations in O with arguments restricted to words starting with the letter a . The same can be done with all other letters in A of any size. This example describes a sharding strategy. In the context of SMR, it approximates the scalable SMR approach presented in [Bezerra et al., 2014]. Again, although the authors did not intend to apply composition strategies to

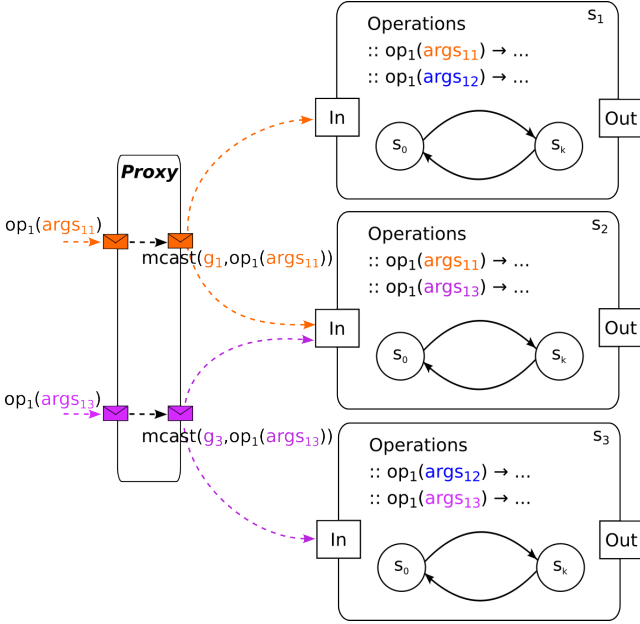


Figure 5. Implementing SMR with state partitioning.

SMR when defining Scalable SMR (SSMR), SSMR also fits well with our approach and can be described as a CSMR.

Other sharding-based approaches could benefit from our CSMR. For example, DynaTree [Eslahi-Kelozazi *et al.*, 2020] uses sharding to partition the nodes of a B+ tree, distributing them over a set of partitions, with each partition being replicated. DynaTree showed good scalability results when compared to BerkeleyDB [Olson *et al.*, 1999]. However, developing this structure involves challenges, such as the overhead of ordering and coordination to execute operations consistently, ensuring linearizability. Similarly to our proposal, an atomic multicast mechanism is required to ensure that all involved partitions execute the same requests in the same order.

A common application of the B+ tree is in file systems, such as the famous BTRFS [Rodeh *et al.*, 2013], the open source file system for Linux based on copy-on-write, which primarily aims to be flexible, allowing it to be installed on both smartphones and enterprise servers while supporting different workloads. Regarding distributed B+ trees, a well-known application is the distributed database Spanner [Corbett *et al.*, 2013], enabling global distributed transactions in the database. Additionally, Spanner ensures strong consistency between partitions, with clocks synchronized in each of them.

The applications of the model described in this work are not limited to the approaches discussed in this section, but extend to other scenarios where it is interesting or necessary to combine functionalities or even entire systems. This aligns with concepts explored in microservices, where small, specialized applications combine to form a larger and more robust system.

4.2 Composition and consistency

In this section, we state that a CSMR constructed using the composition strategies presented in this paper ensures strong consistency.

Definition 12 (Subhistory). Given a history H , as defined in Definition 6, a subhistory of a history H , H_γ , is the subsequence of all events in H for a multicast group γ .

Proposition 1. Given an SMR replica with history H , the subhistories H_{γ_i} of the server replicas in M_i , $0 \leq i \leq n$, are sequential.

For every $deliver(\gamma_i, o_k)$, there is an immediate reply $send(m\langle o_k, u_k \rangle)$ in the subhistory H_{γ_i} . This is because there exists a $(o_k, s_i) \in R$, for some $s_i \in M_i$, and s_i is not faulty. Thus, s_i executes o_k and produces the output u_k .

Proposition 2. Given state machine replicas in M_0 to M_n , any two operations o_i and o_j such that there is a $(o_i, s_i) \in R$, for some $s_i \in M_a$, and o_j such that there is a $(o_j, s_j) \in R$, for some $s_j \in M_b$, and $M_a \neq M_b$, o_i and o_j do not depend on each other.

Two operations are independent if they either access different variables or only read variables commonly accessed; conversely, two operations are dependent if they access one common variable v and at least one of the operations changes the value of v . For example, two read operations are independent, while a read and an update operation on the same variable are dependent. In the proposed composition strategies, each group of state machine replicas operates only over independent variables. As illustrated in Section 4.1.1, operations from different sets O_i and O_j do not share variables when combining operations from different state machine replicas. The composition illustrated in Section 4.1.2 allows the operations in a set O_i to be implemented with different versions, but the operation versions do not share variables. Finally, a valid composition by state partitioning, as described in Section 4.1.3, must guarantee that the intersection of variables handled by different SMRs is empty.

Proposition 3. A replica's execution is consistent with the linearizability criterion.

Linearizability states that an execution respects the real-time ordering of operations across all clients and the semantics of the operations as defined in their sequential specifications. Operations o_i and o_j do not overlap in time if one operation, say o_i is submitted by a client and responded by the service before o_j is submitted by another client. Linearizability holds since: (a) non-overlapping operations are submitted and responded sequentially as demonstrated in Proposition 1; (b) overlapping in time, independent operations can be executed in different moments in different replicas, but preserve the semantics of their sequential specifications since they are independent of the concurrent operations being processed; (c) overlapping in time, dependent operations do not exist in the composition strategies proposed as demonstrated in Proposition 2.

Linearizability has become the reference standard for the correctness of concurrent objects and is commonly used to discuss the correctness of SMR implementations as well [Lynch, 1996; Castro *et al.*, 1999; Berger *et al.*, 2021]. Regarding composition, authors in [Oliveira Vale *et al.*, 2024] have demonstrated that linearizable concurrent objects can be horizontally composed while preserving linearizability, a property

Herlihy and Wing [Herlihy and Wing, 1990] refer to as locality. This means that if the operations of individual SMRs combined in the composition are independent, they behave as if their operations occurred atomically under any concurrent execution. This observation supports SMR composition, as each SMR implementation can be verified against its linearized specification individually. Once proven correct, it can be adopted as an individual component in the composition.

5 CSMR Architecture

This section introduces a high-level architecture for CSMR, serving as a blueprint for how the model can be realized in practice. Although it does not yet provide a concrete implementation, the specification clarifies the essential components, their responsibilities, and the way they interact. Such a structured view not only guides future implementations but also demonstrates how CSMR can be integrated with modern development practices. We illustrate the composition of SMR services using a declarative API, showing how system designers could configure and reason about compositions.

The Composing State Machine Replication (CSMR) architecture is based on two essential modules, as illustrated in Figure 6. The first is the *proxy*, which manages client requests and forwards them to the second module, the *CSMR layer*. The CSMR layer hosts the set of replicated services. These services will effectively execute client requests as if they were independent SMRs. For illustration purposes, Figure 6 depicts a *CSMR layer* composed of two services: a Key-Value Store and a Logging Service. This example helps to ground the discussion, but the architecture is not limited to this setup.

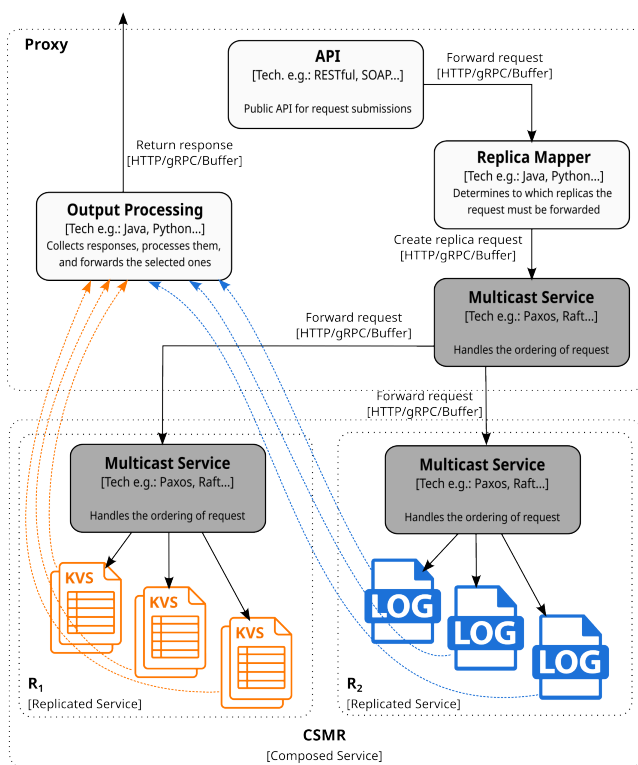


Figure 6. CSMR Architecture

We now detail the components of each module:

API: Public API exposed to the client. This service provides access to CSMR operations (e.g., *get*, *put*) and can be implemented using RESTful, SOAP, gRPC, or other technologies, depending on project or developer requirements. Later, the API forwards the request to the Replica Mapper.

Replica Mapper: Keeps track of all replicas and the operations each one executes (Definition 9). When receiving a request, it identifies the replicas responsible for executing the operation and generates a corresponding request for each one. For example, a *put* request will generate individual *put* requests for replicas that handle the Key-Value Store operations, while producing *append* requests for replicas that manage Logging operations. Once the requests are prepared, the Replica Mapper forwards them to the Multicast Service. This service can be implemented in any programming language (e.g., Java, C, Python, Golang).

Multicast Service: Must implement multicast communication. It is considered as a black box, i.e., existing implementations for multicast or consensus protocol can be used, such as Paxos (e.g., Multi-Ring Paxos), Raft, or others. This service guarantees that all replicas deliver requests in the same order and may operate as an independent external service, separate from the proxy or the CSMR system. Once requests are correctly ordered, the multicast service forwards them to the replicated services, which subsequently perform the same delivery order before execution.

Replicated Services: Represents the composition (Definition 10), where R_1 denotes the replicas from the Key-Value Store and R_2 represents the replicas from the Logging Service. In this illustration, the two replicated services are entirely separate. However, mixed configurations (i.e., R_1 or R_2 containing both KVS and logging functionalities) are allowed. Upon receiving a request from the internal multicast service, the triggered replicas execute the operation and forward their responses to the Output Processing.

Output Processing: Collects all responses received from the replicated services (Definition 11). An application-dependent function must be implemented to determine the correct response to forward to the API, which will then be sent to the client. Note that in this configuration, both the Key-Value Store and Logging Service return responses. However, only the KVS responses are sent to the client.

5.1 Declarative API

This section outlines how CSMR can be expressed declaratively and how clients can use the service. The proposed approach resembles microservices orchestration and make use of widely adopted technologies such as RPC and YAML configuration files, aligning well to modern services development practices. The proposed CSMR adopts a proxy-mediated architecture for coordinating operations across composable replicated services. Clients interact with the system via a JSON-RPC-inspired API, while backend compositions are specified declaratively using YAML configuration files. The proxy acts as an interceptor, responsible for:

1. validating and routing client requests using configurable mapping functions;
2. orchestrating the composable replicated services, and;

3. processing the collected responses through dedicated output functions before delivering the final result to the client.

The API structure follows a JSON-RPC-like structure, where clients send requests specifying the operation name and arguments, and receive responses containing the result. Each request indicates an `id` field, the service operation name (method), operation parameters (params), and expects a corresponding response containing the same `id` and a result value (result). As an example, the request `get("key1")` and its corresponding response `"value1"` can be represented in JSON as follows:

```
// Request
{
  "id": 1,
  "method": "Get",
  "params": {
    "Key": "key1"
  }
}
// Response
{
  "id": 1,
  "result": "value1"
}
```

System behavior is configured through YAML files specifying the composable replicated services operations and the compositions. The following YAML section defines the interface for each replicated service, including their available operations with typed parameters and return values.

```
services:
  lock_service:
    addresses: ["node1:3000",
               "node2:3001", "node3:3002"]
    operations:
      - method: "Acquire"
        params:
          - name: key
            type: string
        returns: boolean
      - method: "Release"
        params:
          - name: key
            type: string
        returns: boolean
  kv_store:
    addresses: ["node3:4000",
               "node4:4001", "node5:4002"]
    operations:
      - method: "Get"
        params:
          - name: key
            type: string
        returns: string
      - method: "Put"
        params:
          - name: key
            type: string
          - name: value
            type: string
        returns: void
```

```
logger:
  addresses: ["node5:5000",
             "node6:5001", "node7:5002"]
  operations:
    - method: "Append"
      params:
        - name: entry
          type: string
      returns: bool
    - method: "Retrieve"
      params:
        - name: first
          type: int
        - name: last
          type: int
      returns: []string
    - method: "Truncate"
      params:
        - name: index
          type: int
      returns: bool
```

The configuration file defines a declarative configuration for three replicated services, `lock_service`, `kv_store`, and `logger`, used in a CSMR system. Each service specifies a set of replica addresses and a list of supported operations with their input parameters and return types. All replicas operate on distinct addresses, as specified by the `node address` followed by the designated port number. Each service is deployed with three replicas.

- The `lock_service` replicas run on endpoints given by `node1:3000`, `node2:3001`, and `node3:3002`. They implement operations `Acquire(key: string) → boolean`, that acquires a lock for a given key, and `Release(key: string) → boolean`, that releases the lock for the key;
- The `kv_store` replicas run on endpoints `node3:4000`, `node4:4001`, and `node5:4002`. They implement operations `Get(key: string) → string`, that retrieves the value for a given key, and `Put(key: string, value: string) → void`, that stores a key-value pair;
- The `logger` replicas run on endpoints `node5:5000`, `node6:5001`, and `node7:5002`. They implement operations `Append(entry: string) → bool`, that appends a log entry, `Retrieve(first: int, last: int) → []string`, that retrieves log entries within a range, and `Truncate(index: int) → bool`, that truncates the log at a given index.

Next we describe how to specify SMR composition. The `compositions` section in the YAML file defines how individual services are combined to implement client-facing operations. Each composition specifies the client API signature, the input mapper function that coordinates service calls, and the output function that processes the collected responses.

```
compositions:
  - name: "ExposeLockAcquire"
    type: "Addition"
    method: "Acquire" # What clients call
    service_operation: "lock_service.Acquire"
    # Directly maps to this service operation
```

```

# Params/return are inferred from
#lock_service.Acquire

- name: "ExposeLockRelease"
  type: "Addition"
  method: "Release"
  service_operation: "lock_service.Release"

- name: "GetWithLogging"
  type: "Composition"
  # Client-facing operation
  # (defines input/output for mappers)
  method: "Get"
  params:
    - name: key
      type: string
  returns: string
  # Output function must return this

# Mapper functions (no signatures needed,
# inferred from above + service section)
input_mapper: "GetMapper"
output_function: "GetOutputFunction"

```

The section compositions includes three entries:

1. ExposeLockAcquire
 - *Type*: Addition
 - *Purpose*: Exposes the Acquire method from lock_service directly to clients.
 - *Behavior*: No transformation or mapping – params and return types are inferred from the service definition.
2. ExposeLockRelease
 - *Type*: Addition
 - *Purpose*: Similarly exposes Release from lock_service to clients as-is.
3. GetWithLogging
 - *Type*: Composition
 - *Purpose*: Composes a new client-facing Get operation that combines multiple services (e.g., kv_store and logger).
 - *Behavior*: Defines its own input/output signature. Uses a custom input_mapper (GetMapper) to split or modify requests from services. Applies an output_function (GetOutputFunction) to process and return the final result to the client.

The input and output functions should use a plugin system. Developers implement these functions separately and register them with the proxy. The YAML simply references these predefined functions by name, allowing custom compositions without modifying the proxy code.

6 Conclusion

In this work, we introduce Composable State Machine Replication, a novel approach that enables the construction of more complex applications while preserving fault tolerance and consistency guarantees of existing SMR implementations.

Through composable SMRs, this modular approach encourages the development of loosely coupled, flexible architectures and contributes both to the theoretical foundations of SMR and to practical, contemporary system design in cloud and microservice-based platforms.

We presented a comprehensive formalization of SMR and its compositionality, expanding a prior formalization outlined in Alves *et al.* [2024]. Unlike the prior work, which focused on the concept of combining SMRs, this paper delves into the execution model and output semantics, addressing how client replies are correctly derived using output functions managed by the proxy component.

To demonstrate the expressive power of CSMR, we introduced a set of running examples designed to reflect realistic application scenarios. These examples illustrate various composition strategies: (i) augmenting existing SMRs with new operations (Section 4.1.1); (ii) enriching operation semantics by enabling different replicas to execute complementary logic for the same operation (Section 4.1.2); and (iii) partitioning application state across independent SMRs (Section 4.1.3).

The examples not only highlight the potential for composing SMRs but also reveal useful insights. For example, once individual SMRs are verified against their linearizable specifications, they can be safely composed without violating linearizability. Moreover, the strategies proposed here align with related work on improving SMR scalability and resource sharing [Xavier *et al.*, 2020; Scharf *et al.*, 2023; Bezerra *et al.*, 2014]. While such works do not explicitly frame their approaches as compositions, they are compatible with and can be reinterpreted under our CSMR framework. By offering a precise and formalized model for SMR composition, we hope to encourage the emergence of new applications and strategies for composition in this area.

We also acknowledge that not all composition schemes are straightforward. In particular, designs that involve shared state or coordinated updates across multiple SMRs pose challenges for consistency and modularity. These strategies may undermine transparency by requiring detailed knowledge of component internals and jeopardize atomicity guarantees when updates span independent SMRs. Lastly, we described a high-level CSMR architecture and an API supporting configuration model for implementing CSMR in practice. Although a full implementation is outside the scope of this paper, our declarative configuration approach, using technologies such as YAML, reflects modern practices in service orchestration and microservice development. As a next step, we intend to build a complete proxy service capable of supporting CSMR-based applications.

Declarations

Authors' Contributions

All the authors developed the work as a whole in a collaborative effort.

Competing interests

The authors declare that they have no conflicts of interest.

References

- Alchieri, E., Dotti, F., Mendizabal, O. M., and Pedone, F. (2017). Reconfiguring parallel state machine replication. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 104–113, China. IEEE. DOI: 10.1109/SRDS.2017.23.
- Altinbeken, D. and Siler, E. G. (2012). Commodifying replicated state machines with openreplica. Technical report, Cornell University. Available at: <https://hdl.handle.net/1813/29009>.
- Alves, C. M., Idalino, T. B., and Mendizabal, O. (2024). Extending state machine replication through composition. In *Proceedings of the 13th Latin-American Symposium on Dependable and Secure Computing, LADC '24*, page 231–240, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3697090.3697106.
- Attiya, H. and Welch, J. (2004). *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley-Interscience. Book.
- Avizienis, A. and Kelly, J. P. (1984). Fault tolerance by design diversity: Concepts and experiments. *Computer*, 17:67–80. DOI: 10.1109/MC.1984.1659219.
- Batista, E., Alchieri, E., Dotti, F., and Pedone, F. (2022). Early scheduling on steroids: Boosting parallel state machine replication. *Journal of Parallel and Distributed Computing*, 163:269–282. DOI: 10.1016/j.jpdc.2022.02.001.
- Berger, C., Reiser, H. P., and Bessani, A. (2021). Making reads in bft state machine replication fast, linearizable, and live. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 1–12. IEEE. DOI: 10.1109/srds53918.2021.00010.
- Bessani, A., Sousa, J., and Alchieri, E. E. (2014). State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, Atlanta, GA, USA. IEEE. DOI: 10.1109/DSN.2014.43.
- Bezerra, C. E., Pedone, F., and Van Renesse, R. (2014). Scalable state-machine replication. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 331–342, USA. IEEE. DOI: 10.1109/DSN.2014.41.
- Burgos, A., Alchieri, E., Dotti, F., and Pedone, F. (2021). Exploiting concurrency in sharded parallel state machine replication. *IEEE Transactions on Parallel and Distributed Systems*, 33:2133–2147. DOI: 10.1109/TPDS.2021.3135761.
- Burrows, M. (2006). The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, USA. USENIX Association. DOI: 10.5555/1298455.1298487.
- Castro, M. and Liskov, B. (2002). Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20:398–461. DOI: 10.1145/571637.571640.
- Castro, M., Liskov, B., et al. (1999). Practical byzantine fault tolerance. In *OSDI*, pages 173–186, USA. USENIX Association. DOI: 10.5555/296806.296824.
- Chandra, T. D., Griesemer, R., and Redstone, J. (2007). Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, USA. ACM. DOI: 10.1145/1281100.1281103.
- Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al. (2013). Spanner: Google’s globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22. Available at: <https://static.googleusercontent.com/media/research.google.com/en//archive/spanner-osdi2012.pdf>.
- Cui, H., Gu, R., Liu, C., Chen, T., and Yang, J. (2015). Paxos made transparent. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 105–120, USA. ACM. DOI: 10.1145/2815400.2815427.
- Dang, Z., Ibarra, O. H., and Su, J. (2004). Composability of infinite-state activity automata. In *International Symposium on Algorithms and Computation*, pages 377–388, Berlin. Springer. DOI: 10.1007/978-3-540-30551-4_34.
- DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Voshall, P., and Vogels, W. (2007). Dynamo: Amazon’s highly available key-value store. *ACM SIGOPS operating systems review*, 41:205–220. DOI: 10.1145/1323293.1294281.
- Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36:372–421. DOI: 10.1145/1041680.1041682.
- Eslahi-Kelorazi, M., Le, L. H., and Pedone, F. (2020). Developing complex data structures over partitioned state machine replication. In *2020 16th European Dependable Computing Conference (EDCC)*, pages 9–16. DOI: 10.1109/EDCC51268.2020.00012.
- Etc (2013). etcd: A distributed, reliable key-value store for the most critical data of a distributed system. Available at: <https://etcd.io/>.
- Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, USA. ACM. DOI: 10.1145/1165389.945450.
- Herlihy, M. P. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12:463–492. DOI: 10.1145/78969.78972.
- Howard, H., Schwarzkopf, M., Madhavapeddy, A., and Crowcroft, J. (2015). Raft refloated: Do we have consensus? *ACM SIGOPS Operating Systems Review*, 49:12–21. DOI: 10.1145/2723872.2723876.
- Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). Zookeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, page 11, USA. USENIX Association. DOI: 10.5555/1855840.1855851.
- Kapritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L., and Dahlin, M. (2012). All about eve: execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 237–250. USENIX Association. Available at: <https://dl.acm.org/doi/10.5555/2387880>.

- 2387903.
- Kirsch, J. and Amir, Y. (2008). Paxos for system builders: An overview. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 1–6, USA. ACM. DOI: 10.1145/1529974.1529979.
- Kotla, R. and Dahlin, M. (2004). High throughput byzantine fault tolerance. In *DSN*, pages 575–584, Florence, Italy. IEEE. DOI: 10.1109/DSN.2004.1311928.
- Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565. DOI: 10.1145/359545.359563.
- Lamport, L. (2001). Paxos made simple. *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121, December 2001), 32:51–58. Available at: <https://lamport.azurewebsites.net/pubs/paxos-simple.pdf>.
- Lamport, L. (2006). Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125. DOI: 10.1007/s00446-006-0155-x.
- Lamport, L., Malkhi, D., and Zhou, L. (2010). Reconfiguring a state machine. *ACM SIGACT News*, 41:63–73. DOI: 10.1145/1753171.1753191.
- Lampson, B. W. (1996). How to build a highly available system using consensus. In *International Workshop on Distributed Algorithms*, pages 1–17, Berlin, Heidelberg. Springer. DOI: 10.5555/645953.675640.
- Lynch, N. and Musco, C. (2022). *A Basic Compositional Model for Spiking Neural Networks*, pages 403–449. Springer Nature Switzerland, Cham. DOI: 10.1007/978-3-031-15629-8_2.
- Lynch, N. A. (1996). *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. DOI: 10.1007/bfb0022433.
- Manhattan, T. (2014). Manhattan, our real-time, multi-tenant distributed database for twitter scale. Available at: https://blog.x.com/engineering/en_us/a/2014/manhattan-our-real-time-multi-tenant-distributed-database-for-twitter-scale [Accessed: Jun. 2024].
- Marandi, P. J., Bezerra, C. E. B., and Pedone, F. (2014). Rethinking state-machine replication for parallelism. In *ICDCS*, pages 368–377, Madrid, Spain. IEEE. DOI: 10.1109/ICDCS.2014.45.
- Masti, S. (2021). How we built a general purpose key value store for facebook with zippydb. Available at: <https://engineering.fb.com/2021/08/06/core-infra/zippydb/> [Accessed: Jun. 2024].
- Mendizabal, O. M., De Moura, R. S. T., Dotti, F. L., and Pedone, F. (2017). Efficient and deterministic scheduling for parallel state machine replication. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 748–757, USA. IEEE. DOI: 10.1109/IPDPS.2017.29.
- Oliveira Vale, A., Shao, Z., and Chen, Y. (2024). A compositional theory of linearizability. *Journal of the ACM*, 71(2):1–107. DOI: 10.1145/3571231.
- Olson, M. A., Bostic, K., and Seltzer, M. I. (1999). Berkeley db. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191. Book.
- Pacheco, L., Dotti, F., and Pedone, F. (2022). Strengthening atomic multicast for partitioned state machine replication. In *Proceedings of the 11th Latin-American Symposium on Dependable Computing*, pages 51–60. DOI: 10.1145/3569902.3569909.
- Pereira, P. M., Dotti, F. L., Meinhardt, C., and Mendizabal, O. M. (2019). A library for services transparent replication. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 268–275, USA. ACM. DOI: 10.1145/3297280.3297308.
- Perone, M. and Karachalias, G. (2023). Crème de la crem: Composable representable executable machines. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Functional Software Architecture*, page 11–19, USA. ACM. DOI: 10.1145/3609025.3609480.
- Rodeh, O., Bacik, J., and Mason, C. (2013). Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32. DOI: 10.1145/2501620.2501623.
- Scharf, J. a. L., Xavier, L. G. C., and Mendizabal, O. M. (2023). Joining parallel and partitioned state machine replication models for enhanced shared logging performance. In *Proceedings of the 12th Latin-American Symposium on Dependable and Secure Computing*, page 90–99, USA. ACM. DOI: 10.1145/3615366.3615422.
- Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM CSUR* 1990, 22:299–319. DOI: 10.1145/98163.98167.
- Sela, G., Herlihy, M., and Petrank, E. (2021). Brief announcement: Linearizability: A typo. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 561–564. DOI: 10.1145/3465084.3467944.
- Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. In *MSST*, pages 1–10, USA. IEEE. DOI: 10.1109/MSST.2010.5496972.
- Xavier, L. G., Dotti, F., Meinhardt, C., and Mendizabal, O. (2020). Scalable and decoupled logging for state machine replication. In *Proceedings of the 38th Brazilian Symposium on Computer Networks and Distributed Systems*, pages 267–280, Brasil. SBC. DOI: 10.5753/s-brc.2020.12288.