UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Caroline Martins Alves

Extending State Machine Replication through Composition

Florianópolis
2026

Caroline Martins Alves

Extending State Machine Replication through Composition

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina para a obtenção do título de mestra em Ciência da Computação.
Orientador: Prof. Odorico Machado Mendizabal, Dr.
Coorientadora: Prof.(a) Thaís Bardini Idalino, Dr.(a)

Florianópolis
2026

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Martins Alves, Caroline
Extending state machine replication through composition / Caroline Martins Alves ; orientador, Odorico Machado Mendizabal, coorientadora, Thaís Bardini Idalino, 2026.
79 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, 2026.

Inclui referências.

1. Ciência da Computação. 2. Computação distribuída. 3. Tolerância a falhas. 4. Replicação máquina de estados. 5. Composição. I. Machado Mendizabal, Odorico. II. Bardini Idalino, Thaís. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. IV. Título.

Caroline Martins Alves

Extending State Machine Replication through Composition

O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Allan Edgard Silva Freitas, Dr.
Universidade Federal da Bahia

Prof. Fernando Luis Dotti, Dr.
Pontifícia Universidade Católica do Rio Grande do Sul

Prof. Frank Augusto Siqueira, Dr.
Universidade Federal de Santa Catarina

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de mestra em Ciência da Computação.

Documento assinado digitalmente
Carina Friedrich Dorneles
Data: 13/04/2026 17:50:15-0300
CPF: ***.834.250-**
Verifique as assinaturas em https://v.ufsc.br

Prof.(a) Carina Friedrich Dorneles, Dr.(a)
Coordenadora do Programa

Documento assinado digitalmente
Odorico Machado Mendizabal
Data: 10/04/2026 08:51:05-0300
CPF: ***.941.170-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Odorico Machado Mendizabal, Dr.
Orientador

Florianópolis, 2026.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisors, Odorico Machado Mendizabal and Thais Idalino Bardini, for their guidance, patience, and support throughout this work. This research would not have been possible without them. I greatly admire them as professors and researchers, and I have learned much from both.

I also thank Matheus Antonio de Souza for the contributions around the Declarative API in Section 6.2.1, which helped concretize some of the theoretical concepts discussed in this work.

Special recognition is extended to the examining committee members, Allan Freitas, Fernando Dotti, and Frank Siqueira, for the helpful and insightful feedback provided.

Finally, I would like to thank everyone who, in some way, supported me during the development of this work by providing words of encouragement or motivation.

RESUMO

A Replicação Máquina de Estados (RME) é uma abordagem bem estabelecida para implementar serviços altamente disponíveis e tolerantes a falhas. No modelo tradicional de RME, todas as réplicas executam as mesmas requisições de forma determinística e na mesma ordem, garantindo um modelo de consistência forte. Nas últimas décadas, a RME ganhou popularidade, levando a uma extensa pesquisa que aprimorou sua resiliência, desempenho e escalabilidade. No entanto, um aspecto ainda não abordado na RME é a composição de serviços. Neste trabalho, damos os primeiros passos nessa direção, apresentando uma definição formal de replicação de máquina de estados e composicionalidade. Ao combinar RMEs por meio de composição, permitimos a reutilização de serviços existentes para construir sistemas mais complexos e confiáveis. Esta abordagem modular proporciona soluções flexíveis e fracamente acopladas, avançando a base teórica de RME e alinhando-se com o desenvolvimento de aplicações em larga escala baseadas em computação em nuvem e microsserviços. Demonstramos que o uso da composição em RME permite adicionar novas funcionalidades às especificações existentes de RME e estende a semântica da operação do serviço, permitindo que diferentes réplicas de máquinas de estados executem etapas complementares para a mesma operação. O *sharding* e a partição de estados também são facilitados pela composição de RME, pois variáveis de estado disjuntas podem ser atribuídas a RMEs separadas. Além disso, esse trabalho apresenta exemplos ilustrativos de composição em SMR e introduzimos uma arquitetura a alto nível, mostrando os componentes essenciais, e suas interações para suportar o processo de composição. Para demonstrar sua praticabilidade, apresentamos uma API para construir um SMR com composição que combina comunicação baseada em RPC com configuração declarativa em formato YAML.

Palavras-chave: Palavra-chave. Computação distribuída. Tolerância a falhas. Replicação máquina de estados. Composição.

ABSTRACT

State Machine Replication (SMR) is a well-established approach for implementing highly available and fault-tolerant services. In the traditional SMR model, all replicas execute the same requests deterministically and in the same order, ensuring strong consistency. Over the past decades, SMR has gained popularity, leading to extensive research that has enhanced its resilience, performance, and scalability. However, one aspect not yet addressed in SMR is service composition. In this work, we take the first steps in this direction, presenting a formal definition of state machine replication and compositionality. By combining SMRs through composition, we enable the reuse of existing services to build more complex and reliable systems. This modular approach provides loosely coupled and flexible solutions, advancing the theoretical background of SMR and aligning with large-scale application development based on cloud computing and microservices. We demonstrate that using composition in SMR allows adding new features to existing SMR specifications and extends the service operation's semantics by enabling different state machine replicas to perform complementary steps for the same operation. Sharding and state partitioning are also facilitated by SMR composition, as disjoint state variables can be assigned to separate SMRs. Furthermore, this work presents illustrative examples of composing SMR and introduces a high-level architecture, detailing its essential components and their interactions to support the composition process. To demonstrate its practicality, we present an API for constructing SMR with composition that combines RPC-based communication with a declarative YAML configuration format.

Keywords: Distributed computing. Fault tolerance. State machine replication. Composition.

RESUMO EXPANDIDO

Introdução

Disponibilidade e escalabilidade são requisitos essenciais em sistemas distribuídos. Disponibilidade é a capacidade de um sistema permanecer operacional mesmo na ocorrência de falhas, e escalabilidade é a capacidade de um sistema lidar com o crescimento da demanda por acessos e requisições sem comprometer seu desempenho. Apesar de serem requisitos cada vez mais desejados em sistemas atuais, atingir seus altos níveis é bastante desafiador. Uma técnica popular para atender a esses requisitos é a replicação. A Replicação Máquina de Estados (RME) é uma abordagem bem estabelecida que utiliza a replicação ativa para implementar sistemas altamente disponíveis e tolerantes a falhas.

No modelo tradicional de RME, todas as réplicas iniciam no mesmo estado, executam os mesmos comandos na mesma ordem, o que garante que todas evoluam para um estado idêntico. Geralmente, as implementações de RME se beneficiam de protocolos de comunicação como o Atomic Broadcast ou protocolos de consenso. Atualmente, a RME é uma abordagem muito popular, utilizada em serviços de coordenação como Apache Zookeeper (HUNT et al., 2010) e Google Chubby (BURROWS, 2006), bem como em sistemas de armazenamento distribuídos, como o Hadoop Distributed File System (HDFS) (SHVACHKO et al., 2010).

Existem também vários trabalhos que propuseram diferentes abordagens para a RME, tais como a tolerância a falhas arbitrárias e bizantinas (CASTRO; LISKOV et al., 1999; BESSANI; SOUSA; ALCHIERI, 2014). Do ponto de vista de desempenho, dado que a execução sequencial de comandos limita a vazão e não aproveita o poder de máquinas com múltiplos processadores, surgiram propostas de implementações paralelas para RME (KOTLA; DAHLIN, 2004; MARANDI; BEZERRA; PEDONE, 2014; MENDIZABAL et al., 2017; BATISTA et al., 2022; BURGOS et al., 2021). Nessas abordagens, comandos independentes podem ser executados de forma paralela, enquanto comandos dependentes são processados de maneira ordenada, sem comprometer a consistência.

Apesar dos avanços significativos na RME, esta dissertação busca abordar um aspecto ainda não explorado: a composição de serviços. A ideia geral é combinar instâncias separadas de RME, criando um modelo mais flexível, que não exige que todas as réplicas executem exatamente os mesmos comandos sem que a consistência seja comprometida.

Objetivos

O objetivo geral deste trabalho é propor uma extensão ao modelo de RME que assume todas as réplicas executando exatamente as mesmas operações. A proposta alinha-se às técnicas de sistemas distribuídos em que pequenos componentes especializados cooperam para construir um sistema altamente disponível e escalável. Dessa forma, busca-se um design de RME mais flexível, em que diferentes réplicas possam realizar operações distintas, mantendo, contudo, as propriedades fundamentais de consistência e linearizabilidade. Assim, visa-se um modelo com réplicas heterogêneas e especializadas, que garanta tolerância a falhas, alta disponibilidade e escalabilidade sem comprometer as garantias essenciais do modelo RME tradicional.

Como objetivos específicos, destacam-se: (i) a formalização da RME e de seus componentes básicos; (ii) a definição de casos de uso, por meio de exemplos práticos, para demonstrar a aplicabilidade do modelo de Composição em RME; (iii) a formalização da Composição em RME; (iv) a definição de estratégias de composição que aproveitem a modularidade do modelo para adicionar ou remover operações, estender a semântica de execução de operações ou realizar a partição de estado do serviço; (v) a proposta de uma arquitetura para a Composição em RME,

detalhando seus componentes e (vi) a proposta de uma API declarativa para a configuração inicial da composição.

Metodologia

Foi realizado uma revisão exploratória da literatura sobre o estado da arte em RME. Essa revisão permitiu constatar que, dentro do nosso conhecimento, nenhum trabalho abordava o uso de composição em RME. Diante dessa lacuna, o trabalho adota uma abordagem proativa e aplicável, visando fomentar novas discussões sobre uma nova aplicação do modelo de RME.
Para fornecer uma base teórica sólida e uma revisão da literatura, são detalhados os fundamentos matemáticos, bem como uma revisão das variações do modelo de RME e de composição. São abordados, em particular, os seguintes tópicos: (i) fundamentação matemática baseada na teoria de conjuntos para a formalização da RME e da composição em RME; (ii) base em RME apresentando os conceitos e propriedades fundamentais do modelo tradicional de RME; (iii) variações da RME revisando trabalhos que propõem extensões ao modelo e investiga propostas que incorporam composição e (iv) técnicas de composição verificando como a composição é aplicada em diferentes áreas da computação.
A formalização da Composição em Replicação de Máquinas de Estados (CSMR) está fundamentada na teoria de conjuntos e no formalismo de relações. Para validar o modelo e ilustrar o processo de composição, empregamos exemplos práticos que exploram diversos cenários de configuração, demonstrando tanto a aplicabilidade prática quanto a flexibilidade da abordagem proposta.
Enquanto a implementação completa e operacional desse serviço se mantém fora do escopo desse trabalho, nos propomos um arquitetura compreensiva para CSMR e detalhamos seus componentes. Isso inclui um serviço proxy para o gerenciamento da comunicação e uma API declarativa baseada em especificações YAML para composição em RME.

Resultados e Discussão

Foram definidos três casos de uso, na forma de aplicações simples, para servir de exemplo nas demonstrações da formalização da RME e da Composição em RME. Cada um deles tem sua funcionalidade e Interface de Programação de Aplicações (API) delimitadas:
Serviço de travas: Coordena o acesso a recursos compartilhados, assegurando exclusão mútua (um cliente por recurso por vez). Para uso seguro, a operação $acquire(lock)$ obtém a trava associada a um recurso. Após o uso, a operação $release(lock)$ libera o recurso.

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

Serviço de armazenamento chave-valor: Implementa um serviço de armazenamento persistente baseado em pares chave-valor. A operação $get(key)$ retorna o valor associado à chave ou um valor nulo se a chave não existir. A operação $put(key, value)$ insere um novo par ou atualiza o valor se a chave já estiver presente.

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

Serviço de log: Mantém um registro histórico e ordenado das operações executadas, preservando a ordem estrita de execução. Isso facilita a análise posterior de eventos do sistema ou atividades de usuários. A operação $append(entry)$ adiciona uma nova entrada ao final do log. A operação $retrieve(first, last)$ recupera uma sequência de entradas dentro de um intervalo. A operação $truncate(index)$ remove todas as entradas anteriores ao índice especificado.

```
void append(Object entry)
```

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

Para a formalização da RME, partimos das relações entre as réplicas e as operações que elas executam. Considera-se uma ordem global atômica (PACHECO; DOTTI; PEDONE, 2022), pela qual todas as réplicas recebem e executam uma mesma sequência totalmente ordenada de comandos.

Definição 1 (Serviço replicado)**.** Seja O um conjunto de operações e M um conjunto de réplicas, então $R = O \times M$ é uma relação de O para M que representa um serviço replicado.

Definição 2 (Operações com argumentos)**.** Seja O o conjunto de operações e A o conjunto de argumentos, então $\mathcal{O} = O \times A$ é um conjunto de operações com argumentos.

Definição 3 (Serviço replicado com argumentos)**.** Seja $\mathcal{O}$ um conjunto de operações com argumentos e M um conjunto de réplicas. Então $R = \mathcal{O} \times M$ é uma relação de $\mathcal{O}$ para M que representa um serviço replicado com argumentos.

Definição 4 (Execução)**.** Seja $\mathcal{O}$ um conjunto de operações com argumentos e U um conjunto de saídas. Então $E \subseteq \mathcal{O} \times U$ é uma relação de $\mathcal{O}$ para U que representa uma execução de operações com argumentos e suas saídas correspondentes.

Definição 5 (Saída)**.** Seja $S_n \in M$ uma réplica, req_i um identificador único associado com cada requisição do cliente, $op \in O$ uma operação e $u \in U$ um valor de retorno para cada requisição. Definimos $w = [s_n, req_i, (op, u)]$ como uma saída para a requisição do cliente.

Definição 6 (Histórico)**.** Seja um histórico H uma rodada de execução em um RME representada por uma sequência finita de requisições de execução E totalmente ordenadas entre as réplicas.

Definição 7 (Serviço replicado composto)**.** Seja O um conjunto não vazio de operações e M um conjunto não vazio de réplicas. Então $R \subseteq O \times M$ é uma relação de O para M que representa um serviço replicado composto, onde $|R| \geq f + 1$ para todo $x \in O$ e f um inteiro positivo.

Definição 8 (Conjunto de replicação)**.** Seja R uma relação de O para M. Então $R(x) = \{s_i \in M : (x, s_i) \in R\}$ é o conjunto de todas as réplicas que executam a operação $x \in O$.

Definição 9 (Conjunto de replicação com argumentos)**.** Seja R uma relação de $\mathcal{O}$ para M, com $\mathcal{O} = O \times A$. Então $R(x) = \{s_i \in M : ((x, a) \in R), \text{com } a \in A\}$ é o conjunto de todas as réplica sque executam a operação com argumento $x \in \mathcal{O}$.

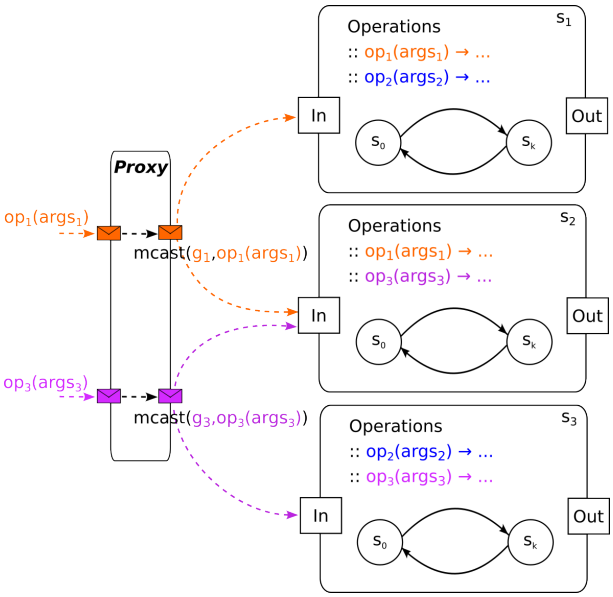
Definição 10 (Composição)**.** Seja O_1 e O_2 conjuntos de operações, M_1 e M_2 conjuntos de réplicas e seja $R_1 \subseteq O_1 \times M_1$ e $R_2 \subseteq O_2 \times M_2$ duas relações. Define-se uma composição como: $R = R_1 \cup R_2$. Como consequência, temos $O = O_1 \cup O_2$ e $M = M_1 \cup M_2$.

Definição 11 (Saídas coletivas)**.** Seja $w_i = [s_i, req, (op, u)]$ uma saída de uma réplica s_i para a requisição req. Nós definimos o conjunto de saídas coletivas como $D = \{w_1, \ldots, w_n\}$ representando a coleção de todas as saídas geradas por uma requisição.

Conforme estabelecido na Definição 10, o modelo de CSMR permite estender ou modificar o conjunto de operações e de réplicas disponíveis. Essa composição pode ser realizada de diferentes formas, como: adicionar ou remover operações de uma instância RME existente; atribuir uma semântica de execução distinta a uma operação já definida ou particionar o estado da aplicação entre as réplicas (e.g., via sharding).

Adicionando operações ao RME: Este tipo de composição estende um RME pré-existente com novas operações. A Figura 1 ilustra esse cenário: a réplica s_1 executa as operações $\{op_1, op_2\}$; s_2, as operações $\{op_1, op_3\}$; e s_3, as operações $\{op_2, op_3\}$.
Quando um cliente invoca $op_1(args_1)$, o proxy envia essa operação, via multicast, para o grupo g_1, que é formado pelas réplicas s_1 e s_2 e são capazes de executá-la. De modo análogo, uma requisição para $op_3(args_3)$ é enviada, também em multicast, para o grupo g_3 (isto é, s_2 e s_3).

Figura 1 – Combinando operações e réplicas de um RME



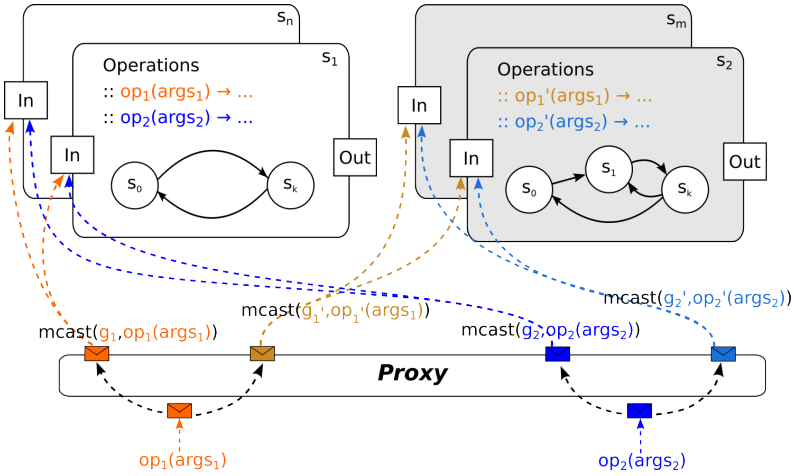
Fonte: A Autora (2025).

No Exemplo 1 é apresentada uma extensão do RME 2, mediante a incorporação das operações do conjunto O_1 que serão executadas pelas réplicas em M_1.

Exemplo 1. Definimos um sistema composto que é formado por um serviço de armazenamento chave-valor com um serviço de travas. Sejam $O_1 = \{acquire, release\}$ e $O_2 = \{get, put\}$ os conjuntos de operações, e $M_1 = \{s_1, s_2, s_3\}$ e $M_2 = \{s_4, s_5, s_6\}$ os conjuntos de réplicas. Determinam-se relações $R_1 \subseteq O_1 \times M_1$ e $R_2 \subseteq O_2 \times M_2$, em que, para cada $x_1 \in O_1$ e $x_2 \in O_2$, os conjuntos de replicação $R_1(x_1)$ e $R_2(x_2)$ devem ter tamanho $f+1$. A composição que resulta no serviço final é dada por $R = R_1 \cup R_2$.

Estendendo a semântica de operações do RME: Esta estratégia de composição permite associar múltiplas implementações a uma mesma operação. Dessa forma, é possível adicionar funcionalidades complementares ou oferecer diferentes variantes de uma mesma funcionalidade. A Figura 2 ilustra esse conceito. As réplicas de s_2 a s_m (servidores cinzas) fornecem outras implementações das operações op_1 e op_2, denotadas por op_1' e op_2', respectivamente. Quando um cliente solicita op_1, o proxy envia a requisição a todas as réplicas dos grupos g_1 e g_1', pois ambos participam do processamento de op_1 (na sua forma base op_1 ou na variante op_1'). Embora todos os servidores (brancos e cinzas) sejam acionados, as réplicas cinzas executam a implementação alternativa op_1'. O processo é análogo para a operação op_2.
Esse tipo de composição introduz n versões de uma mesma operação. Cada versão modifica apenas o estado local das réplicas que a executam. Consequentemente, as saídas produzidas pelas réplicas podem ser diferentes.
Conforme a Definição 11, D representa o conjunto de todas as saídas possíveis geradas pelas n versões de uma operação. Para lidar com essa multiplicidade de respostas, o proxy deve

Figura 2 – Estendendo a execução de operações de um RME.



Source: Author (2025).

implementar uma função determinística $f(D)$ que seleciona qual saída $u \in D$ será retornada ao cliente.

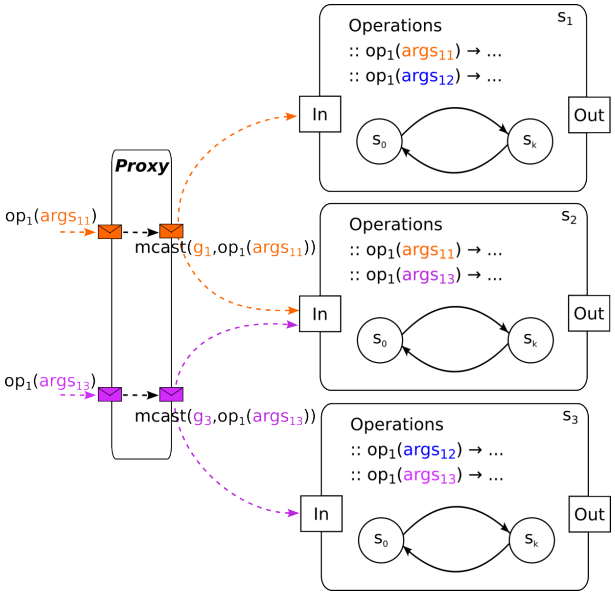
Exemplo 2. Definimos uma composição entre o RME 2 e o RME 3 para estender o serviço de armazenamento com a funcionalidade de log. As operações *put* e *get* do conjunto O_2 foram definidas no RME 2. Nessa composição, essas operações são sobrescritas no RME 3 para que se comportem como a operação *append*. Quando um cliente invoca $put(\text{“}k\text{”},\text{“}v\text{”})$, o proxy envia essa requisição, via multicast, às réplicas tanto do RME 2 quanto do RME 3. As réplicas do RME 2 processam a operação normalmente, associando o valor “v” à chave “k”. Simultaneamente, as réplicas do RME 3 executam a operação *append*, registrando a operação $put(\text{“}k\text{”},\text{“}v\text{”})$ em um log, com isso temos: $(append, (put(\text{“}k\text{”},\text{“}v\text{”}))) \in \mathcal{O}_{\text{append}}$.

Esse exemplo demonstra a necessidade de uma função de seleção de saída, que determina qual resposta é apropriada para retorno ao cliente. Por exemplo, suponha que a requisição $req = 10$ invoque a operação *get*. O proxy a encaminha ao RME 2 e ao RME 3, resultando no conjunto de saídas coletivas $D = \{[s_1, 10, (\text{“}get\text{”},\text{“}value\text{”})], [s_2, 10, (\text{“}get\text{”},\text{“}value\text{”})], [s_3, 10, (\text{“}get\text{”}, \text{“}value\text{”})], [s_4, 10, (\text{“}append\text{”}, true)], [s_5, 10, (\text{“}append\text{”}, true)], [s_6, 10, (\text{“}append\text{”}, true)]\}$. Para selecionar a saída correta da operação *get*, o proxy implementa uma função determinística $f : D \to D$. Uma possível implementação seria $f(D) = \{u \in D : s_n \in R(get), op = get\}$, em que as saídas são filtradas pela operação *get* e pelas réplicas pertencem a $R(get)$ (neste caso, $\{s_1, s_2, s_3\}$).

Particionando o estado por argumento: Esta estratégia de composição divide um grande conjunto de dados em partições menores, que podem ser armazenadas e gerenciadas de forma independente por diferentes instâncias de RME. As operações implementadas são as mesmas em todos os RMEs, porém cada uma acessa um subconjunto disjunto do estado global. A Figura 3 ilustra esse conceito, todas as réplicas executam a mesma operação op_1, mas cada uma é responsável por um subconjunto distinto dos seus argumentos. Quando uma invocação de op_1 contém os argumentos da partição $args_{11}$, o proxy envia a requisição ao grupo g_1, composto pelas réplicas s_1 e s_2, que a processam. De modo análogo, uma operação com os argumentos $args_{13}$ é direcionada ao grupo g_3 (s_2 e s_3).

Exemplo 3. Considere construir, a partir do RME 2, um CSMR para um serviço de armazenamento chave-valor com partição de estado. Seja $A = \{a, b, \ldots, z\}$ um alfabeto. Seja $A^n = A \times A \times \ldots \times A$ o conjunto de todas as palavras de comprimento n. Podemos particionar A^n

Figura 3 – Implementando RME com partição de estado.



Source: Author (2026).

em subconjuntos baseados na primeira letra, como a seguir: $A^n_\alpha = \{\alpha\} \times A \times \ldots \times A, \alpha \in A$, ou seja, o conjunto de palavras de tamanho n que começam com α. Temos, portanto, $A^n = A^n_a \cup A^n_b \cup \ldots \cup A^n_z$. Agora, considere $O = \{\text{get}, \text{put}\}$, $A_1 = A^n_a$, $A_2 = A^n_a \times A^*$. Para particionar o serviço por uma letra específica, podemos restringir os argumentos, por exemplo, $\mathcal{O}_{\text{get}} = \{get\} \times A_1$ e $\mathcal{O}_{\text{put}} = \{put\} \times A_2$ representam operações em O com argumentos restritos a palavras que iniciam com a letra a. Este esquema de partição pode ser estendido para qualquer letra do alfabeto e para palavras de qualquer comprimento.

Considerações Finais

Com isso, damos o primeiro passo para abordar a composição em RME. Apresentamos uma definição formal de RME e de composição, e introduzimos exemplos práticos para ilustrar diferentes estratégias. Essa abordagem promove maior flexibilidade, beneficiando-se de reuso e modularidade. Especificamente, mostramos a composição por meio de: (i) adição de novas operações a um RME existente; (ii) extensão da semântica de operações, permitindo que diferentes réplicas executem variantes da mesma operação; (iii) partição de estado, atribuindo diferentes variáveis de estado a instâncias RME distintas; e (iv) remoção de operações de uma composição. Por fim, apresentamos uma abordagem teórica para a arquitetura de CSMR junto com uma API declarativa, discutindo seus componentes e suas responsabilidades em um ambiente distribuido e sua organização. Essas discussões miram em prover uma fundamentação para futura implementações e extensões. Embora esse trabalho esteja limitado a uma abordagem teórica, essas discussões mostram o prontencial prático da arquitetura, sendo um primeiro passo para guiar a aplicabilidade do modelo em futuros desenvolvimentos utilizando composição em RME.

Palavras-chave: Computação distribuída. Tolerância a falhas. Replicação máquina de estados. Composição.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
CSMR	Composing State Machine Replication
DDD	Domain-Driven Design
DMT	Deterministic Multithreading
FIPS	Federal Information Processing Standards
GST	Global Stabilization Time
HDFS	Hadoop Distributed File System
IoT	Internet of Things
IP	Internet Protocol
KVS	Key-Value Store
ML-DSA	Module-Lattice-Based Digital Signature Algorithm
NIST	National Institute of Standards and Technology
OS	Operating System
PQC	Post-Quantum Cryptography
PRNG	Pseudo-Random Number Generator
PSMR	Parallel State Machine Replication
RPC	Remote Procedure Call
SSMR	Scalable State Machine Replication
SMR	State Machine Replication
SNN	Spiking Neural Network

LIST OF SYMBOLS

$\forall$	Universal quantifier
$\in$	Belongs to
$\notin$	Not belongs to
$\not R$	Not related by R
R	Related by R
$\emptyset$	Empty set
$\subseteq$	Subset of
$\supseteq$	Superset of
$\not\subseteq$	Not a subset of
$\cup$	Union
$\cap$	Intersection
$\setminus$	Difference
$\times$	Cartesian product
$\wedge$	And
$\vee$	Or
$\rightarrow$	Maps
:	Such that
$>$	Greater than

CONTENTS

1 INTRODUCTION

Availability and scalability became common requirements in distributed systems development. Availability is a system's ability to remain operational despite failures, while scalability is the ability to increase workload without a degradation in performance. Although these are nearly universal goals for modern systems, achieving high levels of both is very challenging. Therefore, many techniques aim to address these requirements.

A common method for making a system highly available, and thus fault-tolerant, is replication. This involves deploying multiple copies of a service across several replicas, so that if a bounded number of replica fail, a set of correct replicas can keep the service available to clients. Depending on the desired consistency levels, it also enables load balancing, distributing the workload and leveraging the power of each machine, thereby facilitating scalability. However, implementing replication is challenging because consistency between the replicas must be preserved. State Machine Replication (SMR), the technique we explore in this work, is a popular method used to achieve both availability and scalability.

The SMR (LAMPORT, 1978; SCHNEIDER, 1990) is a well-established approach for implementing highly available fault-tolerant services through active replication. In the traditional SMR model, all replicas start with the same state and execute the same inputs in order, ensuring they all traverse the same state transitions. By following these simple design rules and enjoying the ordering guarantees provided by underlying communication protocols (e.g., atomic broadcast (DÉFAGO; SCHIPER; URBÁN, 2004) or consensus protocols (LAMPSON, 1996; LAMPORT, 2001; HOWARD et al., 2015)), SMR has gained popularity and is widely adopted in practical systems. Prominent examples include coordination services such as Apache Zookeeper (HUNT et al., 2010) and Google Chubby (BURROWS, 2006), as well as distributed storage systems like Google File System (GHEMAWAT; GOBIOFF; LEUNG, 2003) and Hadoop Distributed File System (HDFS) (SHVACHKO et al., 2010), which rely on state machine replicas to keep metadata highly available and consistent. Other typical cases of SMR implementations are Key-Value Store (KVS) services. For instance, Kubernetes, a popular container orchestration platform, uses the etcd (ETCD, 2013) KVS to ensure the fault tolerance of the cluster manager and controllers. KVS are also used in many online services, including Twitter (MANHATTAN, 2014), Amazon (DECANDIA et al., 2007), and Facebook (MASTI, 2021).

There has been a large body of research in SMR over the past decades. Some advances have enhanced SMR with protocols capable of tolerating arbitrary and Byzantine faults (CASTRO; LISKOV et al., 1999; BESSANI; SOUSA; ALCHIERI, 2014). Other approaches have aimed to improve resilience by implementing recovery and reconfiguration protocols in SMR (CASTRO; LISKOV, 2002; LAMPORT; MALKHI; ZHOU, 2010; ALCHIERI et al., 2017). From a performance perspective, SMR sequential execution of requests limits throughput and makes poor use of multiprocessor computing machines. Based on the observation that some requests do not conflict with each other (e.g., operations like $write(x, v)$ and $read(y)$), some researchers have proposed the implementation of Parallel State Machine Replication

(KOTLA; DAHLIN, 2004; MARANDI; BEZERRA; PEDONE, 2014; MENDIZABAL et al., 2017; BATISTA et al., 2022; BURGOS et al., 2021). In such SMR extension, independent requests can be executed in parallel, while dependent requests are partially ordered without affecting replica consistency.

Despite significant advances in various research areas of SMR, this work addresses an aspect not yet explored: service composition. The general idea is to build services by combining separate instances of SMR. From a theoretical standpoint, reasoning about composition in SMR reveals some interesting adoptions and new aspects concerning operations, as well as some challenges regarding the representation of the service state. From a practitioner's view, service composition is gaining traction in deploying cloud services and microservice-based systems. The complexity and costs inherent to the design and implementation of monolithic applications have motivated the development of more flexible and loosely coupled solutions. Hence, an SMR model capable of combining previously existing SMR implementations supports the development of SMR services in a modular way.

1.1 OBJECTIVES

This work proposes an extension to the traditional SMR model, aligning with modular distributed system approaches that combine components for developing compounded systems. Thus, the objective is to design a more flexible SMR model, that allows different replicas to perform distinct operations while still maintaining fundamental properties such as consistency and linearizability. By doing so, the intention is to enable a model in which heterogeneous replicas, executing specialized tasks, ensure fault tolerance, high availability, and greater scalability without compromising the essential guarantees of the SMR paradigm.

1.1.1 Specific Objectives

(a) **SMR Formalization:** Revisit the SMR definition and adapt its concepts, illustrating the basic components using Set Theory and supporting the definitions with applicable examples;

(b) **Use Cases:** Define running examples to demonstrate the applicability of the proposed Composing SMR model and to illustrate its potential usage scenarios;

(c) **Composing SMR Formalization:** Extending our SMR formalization and delineate the new model's elements, their relations, and allowed behaviors to clearly establish its scope for future application;

(d) **Composition Strategies:** Present strategies for composing SMR with different configurations, leveraging its modular construction to add or remove SMR operations, extend the operations semantics, or partition state;

(e) **Composing SMR Architecture:** Propose an initial architecture for Composing SMR in a distributed environment. This includes describing the components involved from the phase of validating and deploying the composition to enabling communication among them for its execution;

(f) **Declarative API:** Propose an API for declaring the initial configuration of Composing SMR, using a standard notation to exemplify the arrangement of replicas and the operations comprising a composition.

1.2 METHODOLOGY

To address the goals of this work, we adopted the following methodology. An exploratory literature review was conducted on the state of the art in SMR. This review helped determine that, to the best of our knowledge, no existing work directly addresses SMR composition. Furthermore, this work adopts a proactive and applicable approach, aiming to generate new discussions on a novel application of SMR.

To provide a solid background and literature review, we drill down into the mathematical foundations as well as a review of SMR variations and composition. In particular, we addressed the following topics:

- **Mathematical Background:** Set Theory, forming the basis for the formalization of SMR and Composing SMR;
- **SMR Foundations:** Core concepts and key properties of traditional State Machine Replication;
- **Review of SMR Variants:** Identify other works that propose different approaches to SMR and search proposals that incorporate composition at the SMR level;
- **Review of Composition Techniques:** An examination of how composition is applied across some computing fields.

The formalization of the Composing State Machine Replication (CSMR) model is grounded in set theory and relational formalism. To validate the model and illustrate the composition process, we employ running examples that explore various configuration scenarios, demonstrating both the practical applicability and the flexibility of our approach.

While the implementation of a fully operational production-grade service remains outside the scope of this work, we propose a comprehensive CSMR architecture and detail its core components. These include a proxy service for communication management and a declarative, YAML-based specification for SMR composition.

1.3 CONTRIBUTIONS

This work brings the following contributions:

- We provide a formal definition of SMR and CSMR, establishing definitions from basic SMR elements up to the precise semantics of the new model. This formalization aims to avoid ambiguity by expressly defining the model's components, their relationships, and the allowed behaviors. Furthermore, it clearly delineates the scope of the new CSMR model, preventing undue generalization in its future application.
- We define use cases based on running examples that demonstrate how CSMR enables the addition of new operations, the extension of operation semantics, the partitioning of state, and the removal of operations. These use cases help ground the model concretely and comprehensively by instantiating it in the described scenarios. This also serves to highlight the model's flexibility and generality. Furthermore, even without presenting a concrete implementation, the use cases establish the model's applicability to real-world cases.
- We propose a CSMR architecture with an Remote Procedure Call (RPC) inspired API in declarative YAML configuration for CSMR. This configuration allows for the practical specification of replicas and their operations within a composition, thereby illustrating the model's adoption in distributed environments.
- We extend the insightful discussions from the articles in which the formal definition, the illustrative use cases, and the declarative API were presented (ALVES; IDALINO; MENDIZABAL, 2024; ALVES et al., 2025).

1.4 ROADMAP

The rest of this work is organized as follows. Chapter 2 introduces the foundational concepts of SMR. Chapter 3 presents the literature review on SMR and composition. Chapter 4 provides the formalization of SMR. Chapter 5 introduces Composing SMR along with its formalization, including composition strategies and propositions regarding composition and consistency. Chapter 6 presents the architectural components for composition and the declarative API. Finally, Chapter 7 concludes the work and discusses future work.

2 STATE MACHINE REPLICATION

State machine replication 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.

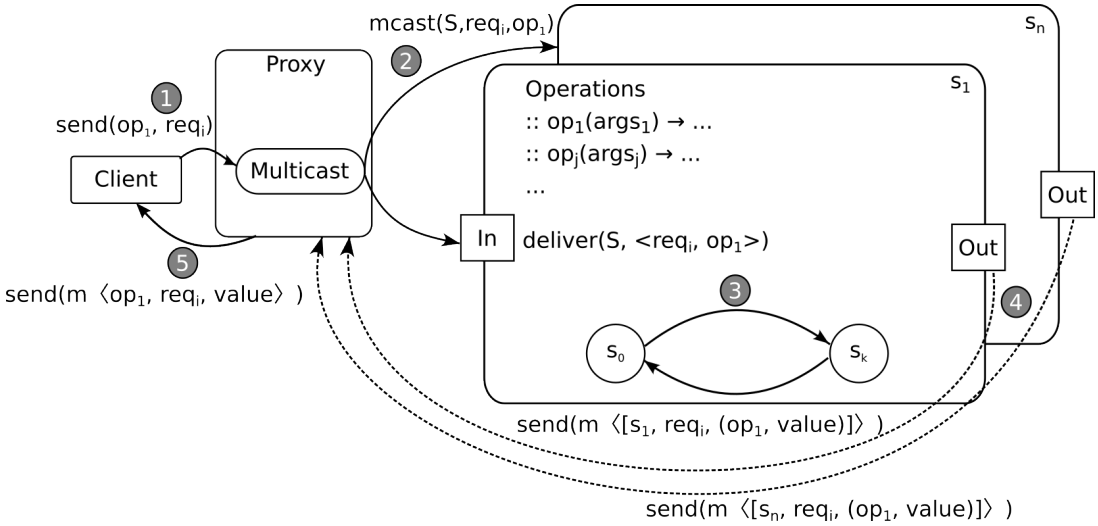
By starting from the same initial state and executing totally ordered commands deterministically, SMR ensures strong consistency. As widely noted in the literature (LYNCH, 1996; CASTRO; LISKOV et al., 1999; BERGER; REISER; BESSANI, 2021), this level of consistency aligns with linearizability, a well-established and widely used correctness criterion in concurrent and distributed systems (HERLIHY; WING, 1990; SELA; HERLIHY; PETRANK, 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; WELCH, 2004).

Figure 4 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 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 operation identifier and the respective result (step ④). The proxy is responsible for receiving the request's output. It waits for the first response from one replica, forwards the 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 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

Figure 4 – Graphical representation of State Machine Replication elements.



Source: Author (2026).

one-to-one or one-to-many communication. One-to-one communication is through primitives send(m) and 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 $mcast(g,m)$ and $deliver(g,m)$, where g refers to the multicast group. Regarding delivery, processes choose from which multicast groups they wish to deliver messages.

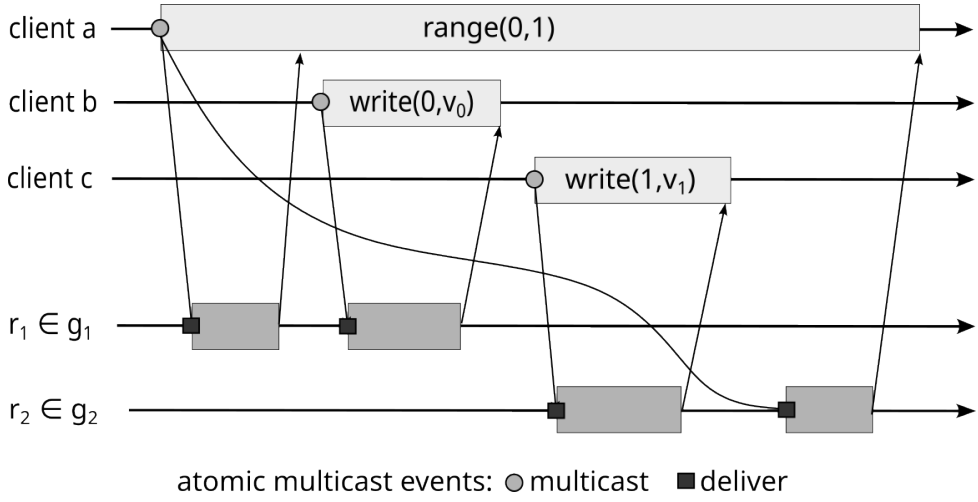
Atomic multicast ensures the following properties (PACHECO; DOTTI; PEDONE, 2022):

> *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, Dotti & Pedone (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 5, adapted from (PACHECO; DOTTI; PEDONE, 2022), the two commands $write(0, v_0)$ and $write(1, v_1)$ insert key-value pairs for *keys* 0 and 1, respectively. Moreover, $write(0, v_0)$ precedes $write(1, v_1)$ in real-time, that is, $write(0, v_0)$ finishes at client b before $write(1, v_1)$ starts at client c. Now consider a concurrent range query $range(0, 1)$ that tries to read previously inserted pairs for *keys* 0 and 1. The range query accesses both partitions, defined by groups g_0 and g_1, and is

delivered and finishes at $r_1 \in g_1$ before $write(0, v_0)$ is delivered, and is delivered at $r_1 \in g_1$ after $write(1, v_1)$ 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: $write(0, v_0)$ and $write(1, v_1)$ are not directly related since they are delivered at different groups, and thus, $range(0, 1) \prec write(0, v_0)$ at r_1 and $write(1, v_1) \prec range(0, 1)$ at r_2. Although the $\prec$ relation is acyclic, $write(1, v_1) \prec range(0, 1) \prec write(0, v_0)$, it does not account for the real time order in which $write(0, v_0)$ precedes $write(1, v_1)$. 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 5 – An execution that violates linearizability (*atomic global order* violation).



Source: Adapted from Pacheco, Dotti & Pedone (2022)

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 5 satisfies weaker atomic broadcast variations but does not satisfy the *atomic global order* property. The $write(1, v_1)$ is multicast by client c after $read(0, 1)$ is delivered by the replica at group g_0, it follows from *atomic global order* that $read(0, 1) \prec write(1, v_1)$; and from the delivery order of requests at g_1, $write(1, v_1) \prec read(0, 1)$, 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 partial synchrony (DWORK; LYNCH; STOCKMEYER, 1988) and quorum intersection (*e.g.*, $n \geq 2f + 1$). The *partially synchronous* model assumes that the system behaves asynchronously for an unknown period but eventually reaches a state of stability, the *Global Stabilization Time* (GST). After GST, there exists a fixed, but potentially unknown, upper bound on the time it takes for a message sent by a correct process to be received by another correct process. In consensus protocols, once GST is reached, the leader will be able to communicate with a quorum of replicas within a predictable timeframe, allowing the protocol to terminate and consensus be achieved.

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 LITERATURE REVIEW

Many researchers have proposed extensions to the traditional SMR model to meet the strict requirements for modern applications concerning scalability, performance, and usability. Advances over the traditional SMR model enabled it to tolerate more severe fault models (CASTRO; LISKOV et al., 1999; BESSANI; SOUSA; ALCHIERI, 2014) and improve resilience by incorporating recovery and reconfiguration protocols (CASTRO; LISKOV, 2002; LAMPORT; MALKHI; ZHOU, 2010; ALCHIERI et al., 2017), among other context-specific improvements. This chapter 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 chapter introduces the reader to different uses of composition, showcasing the versatility and potential developments of this approach.

3.1 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.

3.1.1 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 presented in Kotla & 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 & 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 & 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 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, Bezerra & Pedone (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 State Machine Replication (S-SMR) (BEZERRA; PEDONE; RENESSE, 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 Zookepper 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 Zookepper's 45.000 commands under the same workload.

Table 1 summarizes the researches that have proposed different approaches to SMR. The table presents the main strategies adopted, either parallel or scalable SMR, along with the key idea of each work and how each proposal was implemented using parallelism or scalability mechanisms. The last column highlights some advantages of each work. The proposal of variations in SMR constitutes a rich and extensively explored research area, which has yielded numerous interesting approaches. In line with these studies, we also seek to propose a distinct approach to SMR, specifically based on composition. This technique has not yet been explored in the context of SMR variations.

Table 1 – Comparison of approaches for SMR variations

Approach / Paper	**Main Strategy**	**Key Idea**	**Parallelism / Scalability Mechanism**	**Advantages**
High Throughput Byzantine Fault Tolerance (KOTLA; DAHLIN, 2004)	Parallel SMR	Deterministic scheduler ("parallelizer") dispatches independent commands to worker threads	Atomic broadcast + deterministic scheduling; parallel execution of independent requests	Preserves consistency; leverages multicore systems; first parallel SMR proposal
Efficient and Deterministic Scheduling for Parallel State Machine Replication (MENDIZABAL et al., 2017)	Parallel SMR (optimized)	Efficient dependency tracking with lightweight scheduling	Per-thread queues + bitmap-based fast dependency checks for conflict detection	Higher throughput; reduced contention; efficient dependency representation
Rethinking SMR for Parallelism (MARANDI; BEZERRA; PEDONE, 2014)	Parallel SMR (multicast-based)	Clients assign commands to multicast groups based on independence	Multiple multicast groups enabling partially ordered delivery; eliminates internal scheduler	Lower overhead; direct execution by worker threads; simpler replica design
Scalable State Machine Replication (BEZERRA; PEDONE; RENESSE, 2014)	Scalable SMR	State partitioning with atomic multicast across partitions	Partitioned state; commands span partitions; caching + execution atomicity	Scalable throughput; strong consistency; transparent implementation (Eyrie)

Source: Author (2026)

3.1.2 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, Griesemer & Redstone (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 & Amir (2008) 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 (ALTINBUKEN; SIRER, 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.

Table 2 compares existing approaches that aim to facilitate transparent replication. For each work, the table specifies the main strategy, such as practical development, system design, or a transparent replication mechanism, along with the key idea, such as building a fault-tolerant database, replicating multithreaded programs, or enabling applications to share a consensus layer. The mechanism/technique column describes the approach used to achieve the work's objectives, while the last column summarizes its advantages. Our work shares similarities with these efforts, because levering SMR we enable transparent replication. In our variation of SMR based on composition the property of transparent replication is preserved while offering the additional benefit of allowing replicas to implement different services.

3.2 COMPOSITIONAL METHODS

Now we explore how some works in the literature already introduced some notions of composition in different scenarios.

A proposition on building software architecture as state machine composition using Domain-Driven Design (DDD) is proposed in Perone & 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

Table 2 – Comparison of approaches to implement transparent replication

Approach / Paper	**Main Strategy**	**Key Idea**	**Mechanism / Technique**	**Advantages**
Paxos Made Live (CHANDRA; GRIESEMER; REDSTONE, 2007)	Practical SMR development	Building a fault-tolerant database using SMR	Real-world implementation experience highlighting system design challenges	Exposes practical difficulties; emphasizes need for flexible and maintainable implementations
Paxos for System Builders (KIRSCH; AMIR, 2008)	SMR system design	Engineering details impact liveness and performance	Analysis of implementation-level decisions in SMR systems	Highlights importance of low-level design choices; connects engineering with system correctness and performance
Paxos Made Transparent (CUI et al., 2015)	Transparent replication	Replicates multi-threaded programs transparently	Intercepts POSIX sockets and Pthreads; deterministic multithreading (DMT); consensus per socket call	No need to rewrite applications; ensures deterministic execution; strong consistency
Commodifying Replicated State Machines with OpenReplica (ALTINBUKEN; SIRER, 2012)	Transparent object replication	Clients interact with replicated objects as if local	Paxos-based replication; binary rewriting; dynamic reconfiguration with view change protocol	Transparency for developers; supports dynamic scaling; simplified client interaction
A Library for Services Transparent Replication (PEREIRA et al., 2019)	Shared SMR infrastructure	Multiple applications share the same consensus layer	Transparent replication without explicit communication handling; shared consensus protocol	Improved resource utilization; better performance in shared environments; reduced deployment overhead

Source: Author (2026)

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; IBARRA; SU, 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 $(A; A_1, ..., A_r)$ 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 providing dynamic composition as it provides 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 & Musco (2022) is focused on an algorithmic theory of brain networks, based on synchronous, stochastic Spiking Neural Network (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. 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, Pedone & Renesse (2014) introduces S-SMR (see Subsection 3.1.1 for details), which partitions the state-machine state, allowing each command to access different combinations of partitions. As we illustrate in Subsection 5.1.3, 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, Xavier & Mendizabal (2023) further extends 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 strategies in the context of active replication. The aforementioned works reinforce both the relevance and the potential of SMR composition.

Table 3 compares researches that relate to composition, either by explicitly implementing it in a specific area or by employing techniques that resemble it. The first three works implement composition in distinct areas: software architecture, automata theory, and spiking neural networks. The remaining approaches are in the SMR field and, although they do not implement composition directly, they exhibit similarities: Bezerra, Pedone & Renesse (2014) implement S-SMR, which can be interpreted as implicit composition via sharding; Xavier et

al. (2020) introduce decoupled logging in SMR, where the logging replica has a responsibility separate from the other replicas; and Scharf, Xavier & Mendizabal (2023) combine S-SMR and P-SMR to optimize SMR throughput. These works are relevant to our own for two main reasons. First, the initial three explicitly employ composition, which is the technique we seek to bring to SMR. Second, the last three present SMR variations where replicas exhibit flexibility in their responsibilities or in how commands are executed.

Table 3 – Comparison of approaches related to composition concepts

Approach / Paper	**Domain / Context**	**Key Idea**	**Composition Mechanism**
Crème de la Crem: Composable Representable Executable Machines (PERONE; KARACHALIAS, 2023)	Software architecture (DDD)	Modular architecture via composition of state machines	Compositional state machines using the Crem library; Haskell-based specification of transitions
Composability of Infinite-State Activity Automata (DANG; IBARRA; SU, 2004)	Automata theory / web services	Formalization of composability in language acceptors	Decomposition of accepted strings across automata; k-lookahead delegator for dynamic assignment
A Basic Compositional Model for Spiking Neural Networks (LYNCH; MUSCO, 2022)	Spiking Neural Networks (SNN)	Formal model for composing neural networks	Composition and hiding operators over directed graph-based SNNs; behavior depends on external interfaces
Scalable State Machine Replication (BEZERRA; PEDONE; RENESSE, 2014)	Distributed systems (S-SMR)	State partitioning across replicas	Partitioned state-machine with commands spanning partitions
Scalable and Decoupled Logging for State Machine Replication (XAVIER et al., 2020)	Distributed systems (SMR logging)	Decoupling logging from application replicas	Introduction of logger processes to handle command logs
Joining Parallel and Partitioned State Machine Replication Models for Enhanced Shared Logging Performance (SCHARF; XAVIER; MENDIZABAL, 2023)	Distributed systems (SMR optimization)	Combining parallelism and scalability techniques	Integration of P-SMR and S-SMR with parallel loggers

Source: Author (2026)

4 STATE MACHINE REPLICATION FORMALIZATION

In this chapter, we formalize the traditional SMR model using set theory, specifically relations, as our basis. Since our formalization relies on these discrete structures, we first provide a concise review of fundamental concepts (ROSEN, 2011) in Section 4.1. Readers familiar with these topics may proceed directly to Section 4.2, where we introduce some use cases that help elucidate the formal definitions presented in Section 4.3.

4.1 MATHEMATICAL BACKGROUND

The most elementary discrete structure, upon which all others are based, is the *set*. A set is an unordered collection of distinct objects, referred to as its elements or members. When an element a belongs to a set A, we denote $a \in A$. Conversely, $a \notin A$ denotes that a is not an element of A. It is standard convention to denote sets with uppercase letters (e.g., A, S) and their members with lowercase letters (e.g., a, x). One method for representing sets is the roster method, which lists all elements of the set between braces.

Example 1. The set M_{31} of months containing 31 days expressed by $M_{31} = \{January, March, May, July, August, October, December\}$, the set M_{28} of months containing 28 days expressed by $M_{28} = \{February\}$ and the set $M_{30} = \{April, June, September, November\}$.[1]

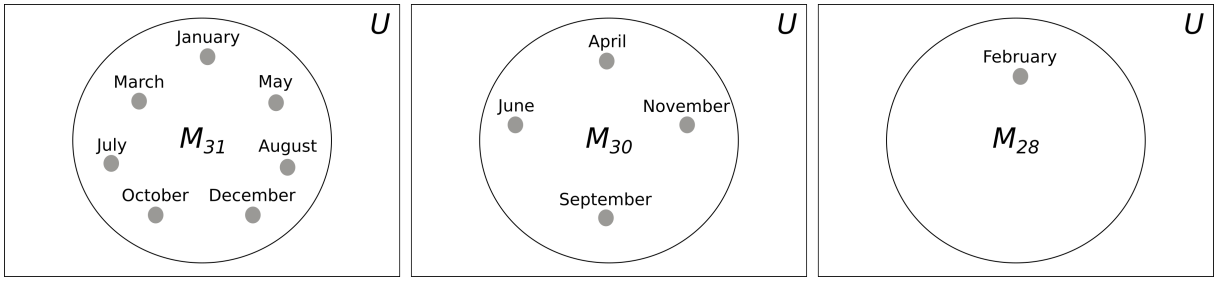
Another way to represent sets is using set builder notation. This notation specifies the elements of a set by describing the properties they must satisfy. To convert Example 1 to set-builder notation we define the universal set $U = \{January, February, March, April, May, June, July, August, September, October, November, December\}$ of all months, then $M_{31} = \{m \in U : m \text{ has 31 days}\}$ is the set of months containing 31 days, $M_{28} = \{m \in U : m \text{ has 28 days}\}$ is the set of months containing 28 days and $M_{30} = \{m \in U : m \text{ has 30 days}\}$ is the set of months containing 30 days.

Certain sets have special names due to their properties. The *empty set*, denoted by $\emptyset$ or $\{\}$, contains no elements. A *singleton set* contains exactly one element. Note that $\emptyset$ is distinct from $\{\emptyset\}$, where the former is a null set and the latter is a singleton set containing the empty set as its element.

Sets can be represented visually using Venn diagrams, introduced by John Venn in 1881 (VENN, 1881). In this model, a rectangle denotes the universal set U, joining all objects under consideration. Sets are typically represented by circles inside this rectangle, and individual elements can be indicated using dots. Figure 6 provides a Venn diagram representing the sets M_{31} (months with 31 days), M_{30} (months with 30 days), and M_{28} (months with 28 days). Each set is depicted as a circle containing the months that satisfy its respective condition. The universal set U in each case includes all twelve months and is represented by the rectangle that encompasses the circles.

[1] For the sake of simplicity in this example, let's ignore the occurrence of 29 days when February is a leap year.

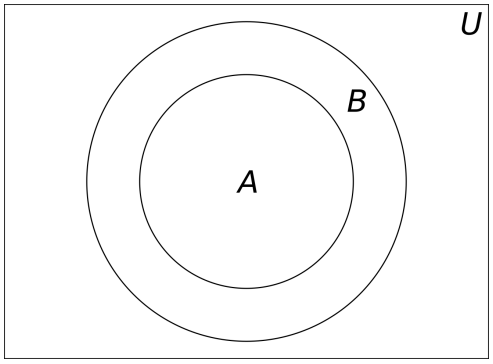
Figure 6 – Venn Diagram for the Sets M_{31}, M_{30} and M_{28}



Source: Author (2026).

A *subset* is a set whose elements are all contained in another set. So A is called a subset of a set B if and only if every element of A is also an element of B. This relationship, denoted $A \subseteq B$, is formally defined by the universal quantification: $\forall x(x \in A \rightarrow x \in B)$. If $A \subseteq B$, we may also say B is a *superset* of A, written $B \supseteq A$. Conversely, A is not a subset of B, denoted $A \nsubseteq B$, if there exists at least one element x such that $x \in A$ and $x \notin B$. Subsets can be visually illustrated, as shown in Figure 7, where set A is a subset of set B.

Figure 7 – Venn Diagram for A subset of B



Source: Adapted from Rosen (2011)

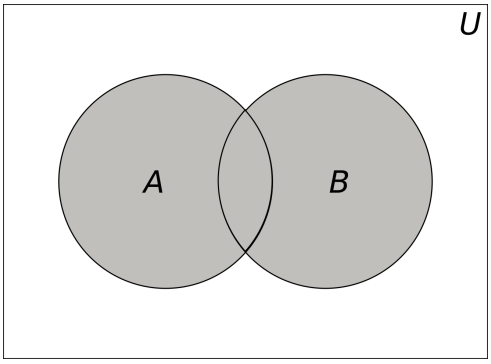
The size of a set is measured by its number of elements. Let S be a set. If there are exactly n distinct elements in S, where n is a nonnegative integer, then S is *finite* and n is the cardinality of S, denoted by $|S|$. A set is *infinite* when it is not finite.

Example 2. Let M_{31} be the set of all months with 31 days, M_{30} be the set of all months with 30 days and M_{28} be the set of all months with 28 days. Then $|M_{31}| = 7$, $|M_{30}| = 4$ and $|M_{28}| = 1$.

Several fundamental operations can be performed on sets. The *union* of two sets A and B, denoted $A \cup B$, is the set containing all elements that are in A, in B, or in both. Thus, an element x belongs to $A \cup B$ if and only if $x \in A$ or $x \in B$, expressed formally as $A \cup B = \{x : x \in A \vee x \in B\}$. Figure 8 illustrates the union of A and B as the highlighted region encompassing both circles.

The *intersection* of two sets consists of their common elements. Formally, for sets A and B, the intersection $A \cap B$ is the set containing all elements that are members of both A and B. Therefore, an element x belongs to $A \cap B$ if and only if $x \in A$ and $x \in B$, formalized as

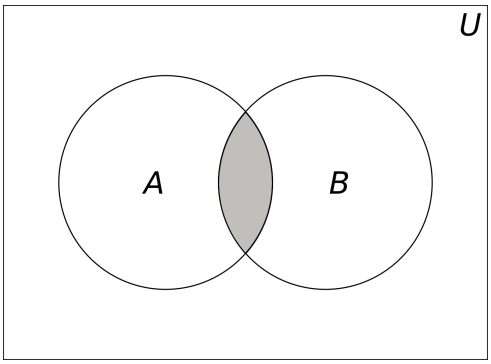
Figure 8 – Venn Diagram for the union of sets A and B



Source: Adapted from Rosen (2011)

$A \cap B = \{x : x \in A \wedge x \in B\}$. Figure 9 provides a graphical representation of the intersection of sets A and B, the highlighted region between the circles indicates the elements common to both sets.

Figure 9 – Venn Diagram for the intersection of sets A and B



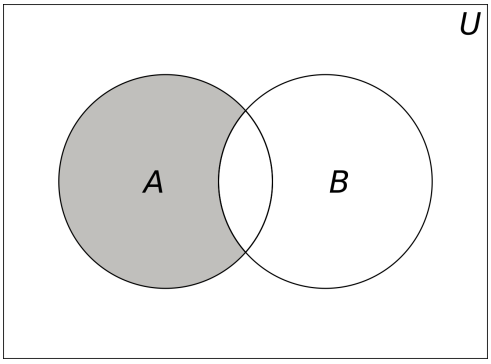
Source: Adapted from Rosen (2011)

The *difference* consists of all elements that belong to one set but not to another. Formally, for sets A and B, the difference of A and B is denoted $A \backslash B$ (or equivalently $A - B$).[2] So an element x belongs to the difference of A and B if and only if $x \in A$ and $x \notin B$, formalized by $A \backslash B = \{x : x \in A \wedge x \notin B\}$. Figure 10 depicts the set difference $A \backslash B$, the highlighted region corresponds to the part of set A that does not intersect with set B, representing the elements that are exclusively in A.

The *complement* is a fundamental set operation derived from the difference. The complement of a set A, denoted $\overline{A}$, is defined with respect to a universal set U. It is equivalent to the set difference $U \backslash A$, containing all elements in U that are not in A. Formally, the complement is defined by the set-builder notation as $\overline{A} = \{x \in U : x \notin A\}$. Thus, an element x belongs to $\overline{A}$ if and only if $x \notin A$. Figure 11 depicts the complement of A. The highlighted region represents $\overline{A}$, which is all the U set excluding the A set.

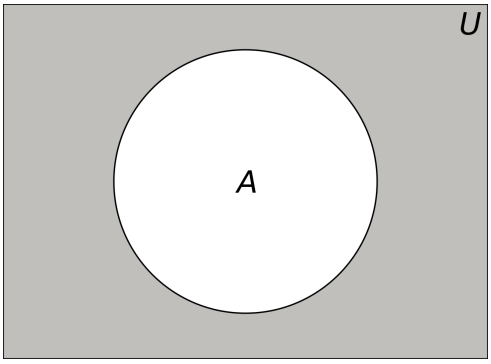
[2] In this work, we adopt the notation \ when referring to difference.

Figure 10 – Venn Diagram for $A \setminus B$



Source: Adapted from Rosen (2011)

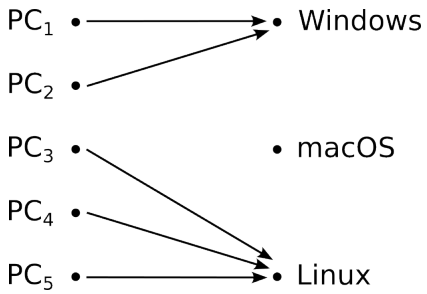
Figure 11 – Venn Diagram for $\overline{A}$



Source: Adapted from Rosen (2011)

The assignment of each element of a set to exactly one element of a second set is called a *function*. To illustrate this concept, consider a company with a set of computers $\{PC_1, PC_2, PC_3, PC_4, PC_5\}$, each running one Operating System (OS) from the set $\{Windows, Linux, macOS\}$. The assignment of an OS to each computer defines a function. For instance, suppose PC_1 and PC_2 run $Windows$, while PC_3, PC_4 and PC_5 runs $Linux$, and no computer runs $macOS$, Figure 12 represents this assignment of computers and OS.

Figure 12 – Assignment of operating systems and computers



Source: Adapted from Rosen (2011)

Function is a fundamental concept in both computer science and discrete mathematics. In computer science, for example, functions model the time required for a computer to solve

a problem of a given size. In discrete mathematics, functions are used to describe discrete structures such as sequences and strings.

Definition 1 (Function)**.** Let A and B be nonempty sets. A *function* f from A to B is an assignment of each element of A to exactly one element of B. The expression $f(a) = b$ is valid if and only if b is the unique element in B assigned by f to the element $a \in A$. If f is a function from A to B, we write $f : A \to B$.

In terms of representation, functions can be specified in several ways. One method is an explicit representation, as shown in Figure 12. Another common way is to use a mathematical formula, like the famous quadratic function $f(x) = x^2$. It is also possible to define a function using a computer program, which implements its logic algorithmically.

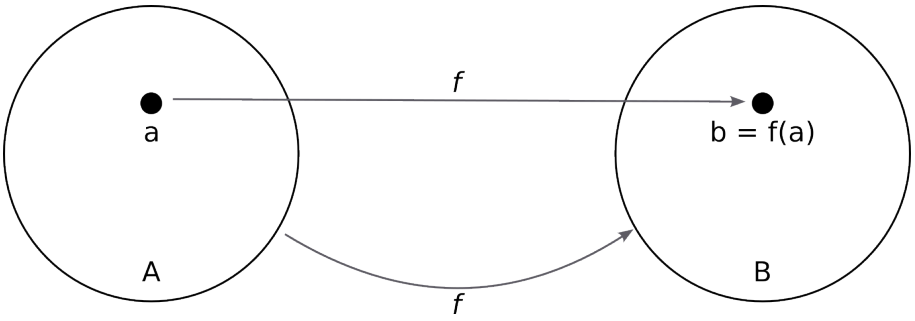
A function $f : A \to B$ can be defined as a specific type of relation from A to B. A function is a relation that contains one and only one ordered pair (a, b) for every element $a \in A$. This unique pair defines $f(a) = b$, where b is the element in B corresponding to a.

Definition 2 (Domain and Codomain)**.** If f is a function from A to B, then A is the *domain* of f and B is the *codomain* of f.

Definition 3 (Image and Preimage)**.** If $f(a) = b$, then b is the *image* of a and a is a *preimage* of b. The range or image of the function f is the set of all images of elements in its domain A.

Also if f is a function from A to B, we can say that f maps A to B. Figure 13 illustrates this scenario, showing sets A and B, where an element $a \in A$ is mapped to an element $b \in B$ by the function f, such that $b = f(a)$.

Figure 13 – Function f maps A to B



Source: Adapted from Rosen (2011)

To define a function, it is necessary to specify its domain, its codomain, and the rule that maps each element of the domain to an element in the codomain. Two functions are **equal** if and only if they have the same domain, the same codomain, and map every element of the domain to the same element in the codomain. Altering the domain, the codomain, or the mapping rule results in a different function.

Examples 3 and 6 provide examples of functions. In both cases, the domain, codomain, range, and the assignment of values for elements in the domain are described.

Example 3. Let O be the function that assigns an OS to a computer. Note that $O(PC_1) = Windows$, for instance. The domain of O is the set $\{Windows, macOS, Linux\}$, and the codomain is the set $\{PC_1, PC_2, PC_3, PC_4, PC_5\}$. The range of O is the set $\{Windows, macOS, Linux\}$, because each OS except $macOS$ is assigned to some PC.

Relationships can be represented in many contexts, for example, between computers and the programs executed on them, or between routers and Internet Protocol (IP) addresses. In set theory, these relationships are represented by a structure called *relation*. One way to represent a relationship between sets is by using ordered pairs. For this reason, a set of ordered pairs is referred to as a binary relation. A relation generalizes the concept of a function by allowing elements of the set A to be associated with multiple elements in B.

Definition 4 (Relation)**.** Let A and B be sets. A relation R is a binary relation from A to B is a subset of $A \times B = \{(a, b) : a \in A, b \in B\}$.

The notation $a\ R\ b$ is used to express that $(a, b) \in R$ and $a \not R\ b$ to express that $(a, b) \notin R$. Examples 4 and 5 illustrate the relation notion and Example 6 presents the use of both relations and functions.

Example 4. Let A be the set of computers and B be the set of programs. Let R be the relation consisting of pairs (a, b) where computer a executes program b. Thus, if both $Computer_1$ and $Computer_2$ execute $WebBrowser$, then the pairs $(Computer_1, WebBrowser)$ and $(Computer_2, WebBrowser)$ belong to R. If $Computer_1$ executes $Photoshop$ but $Computer_2$ does not, then $(Computer_1, Photoshop) \in R$ and $(Computer_2, Photoshop) \notin R$.

Example 5. Let A be the set of routers and B be the set of IP addresses. Define the relation R such that the pair (a, b) belongs to R if a router a is assigned the IP address b. For instance, $(Router_1, 10.0.0.1)$, $(Router_2, 127.0.0.1)$, and $(Router_3, 192.168.0.1)$ are elements of R.

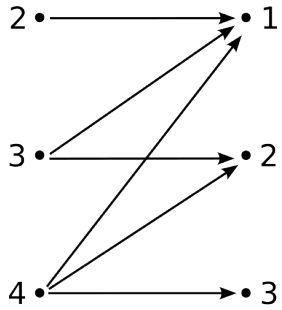
Example 6. Let R be the relation consisting of the ordered pairs $(C, 1972), (Python, 1991), (Java, 1995), (Go, 2009)$, where each pair contains a programming language and its year of creation. Let f be the function defined by this relation R, such that $f(C) = 1972$, $f(Python) = 1991$, $f(Java) = 1995$ and $f(Go) = 2009$. Here, $f(x)$ represents the creation year of the programming language x. The domain of f is the set $\{C, Python, Java, Go\}$. For the codomain, we choose a set that contains all possible creation years for programming languages. A suitable choice is the set of positive integers up to and including 2026 (the current year). This codomain also allows the function to be extended later by adding new programming languages and their creation years. The range (or image) of f is the set of distinct creation years present in the relation, which is $\{1972, 1991, 1995, 2009\}$.

When discussing relations, a special concept is relation on a set. This is a relation from a set A to itself, which is a relation from A to A, and is formally defined as a subset of the Cartesian product $A \times A$.

Example 7. Let A be the set $\{1,2,3,4\}$. The ordered pairs in the relation $R = \{(a,b) : a > b\}$ are: $R = \{(2,1),(3,1),(3,2),(4,1),(4,2),(4,3)\}$

Figure 14 provides two representation of the pairs from the relation in Example 7.In the tabular form, the row headers represent the first element of the ordered pair, and the column headers represent the second element. A mark (the x symbol) in a cell indicates that the corresponding ordered pair is present in the relation.

Figure 14 – Ordered pairs in the relation R from Example 7



R	1	2	3	4
1				
2	x			
3	x	x		
4	x	x	x	

Source: Adapted from Rosen (2011)

One useful property of relations is restriction by the first element. For a relation R, this operation yields the subset of pairs with a specified first coordinate. Definition 5 provides the formal definition.

Definition 5 (Restriction of a relation)**.** Let R be a relation from A to B, we define $R(x)$ to be: $R(x) = \{b \in B : (x,b) \in R\}$, which is the set of all elements in B related to x.

Example 8. Applying restriction to the relation in Example 4 for $Computer_1$ yields all programs executed by it: $R(Computer_1) = \{WebBrowser, Photoshop\}$.

We will now introduce some key properties used to classify relations on a set. The first is the case where every element is related to itself. For instance, consider the set of people $A = \{Emma, Sophia, John, Paul\}$ and define a relation R where $(x,y) \in R$ if and only if x and y are the same age. In this scenario, R would include the pairs $(Emma, Emma), (Sophia, Sophia)$, $(John, John), (Paul, Paul)$, because every person has the same age as themselves. Thus, xRx holds for every person x in A.

Definition 6 (Reflexive Relation)**.** For a relation R on a set A to be reflexive, it must contain every possible pair (a,a) for every element $a \in A$.

Example 9. Consider the set $A = \{1,2,3,4\}$ and the relation $R = \{(1,1),(1,2),(1,4),(2,1),(2,2),$ $(3,3),(4,1),(4,4)\}$. This relation is reflexive because it contains all pairs of the form (a,a) for every element $a \in A$, namely $(1,1)$, $(2,2)$, $(3,3)$, and $(4,4)$.

Definition 7 (Symmetric Relation)**.** A relation R on a set A is called *symmetric* if $(b,a) \in R$ whenever $(a,b) \in R$, for all $a,b \in A$.

Example 10. Consider the set of friends $A = \{Oliver, Charlotte, Noah, James\}$. The relation $R = \{(Oliver, Charlotte), (Charlotte, Oliver), (Noah, James), (James, Noah)\}$ on A is symmetric. This captures the idea that friendship is mutual, if a person is a friend of another, then the latter is also a friend of the former. Consequently, for every pair $(a,b) \in R$, the corresponding pair (b,a) is also in R.

Definition 8 (Antisymmetric Relation)**.** A relation R on a set A is called *antisymmetric* if, for all $a,b \in A$, whenever $(a,b) \in R$ and $(b,a) \in R$, it follows that $a = b$.

Example 11. Consider the set of persons $A = \{Sophia, Chloe, Lily, Grace\}$. The relation $R = \{(Lily, Chloe), (Grace, Sophia)\}$ represents pairs where the first person is older than the second. Note that it is not possible for both (a,b) and (b,a) to be in R for distinct elements a and b, because a person cannot be both older and younger than another simultaneously. The only way both pairs could be present is if $a = b$.

Definitions 7 and 8 can also be expressed using quantifiers. A relation R on a set A is symmetric if $\forall a \forall b((a,b) \in R \rightarrow (b,a) \in R)$. And it is antisymmetric if $\forall a \forall b(((a,b) \in R \wedge (b,a) \in R) \rightarrow (a = b))$.

This means that a relation is symmetric if and only if whenever a is related to b, it is necessarily true that b is related to a. Similarly, a relation is antisymmetric if and only if there are no pairs of distinct elements a and b such that both a is related to b and b is related to a. In other words, the only scenario where both (a,b) and (b,a) are in the relation is when $a = b$. An important point is that symmetric and antisymmetric are not opposites. A relation can have both properties, neither property, or exactly one of them. If a relation contains any pair (a,b) with $a \neq b$ that is symmetric (i.e., (b,a) is also present), then it cannot be antisymmetric.

Definition 9 (Transitive Relation)**.** A relation R over a set A is *transitive* if whenever a is related to b and b is related to c, then a is necessarily related to c, for all $a,b,c \in A$.

Example 12. Consider the set of relatives $A = \{Daughter, Mother, Grandmother\}$. The relation $R = \{(Grandmother, Mother), (Mother, Daughter), (Grandmother, Daughter)\}$ represents the ancestral relation between these relatives. Here, because $(Grandmother, Mother)$ and $(Mother, Daughter)$, transitivity requires $(Grandmother, Daughter)$, which represents that a grandmother is an ancestor of her granddaughter.

Representing with quantifiers, a relation R on a set A is transitive if and only if the following condition holds: $\forall a \forall b \forall c(((a,b) \in R \wedge (b,c) \in R) \rightarrow (a,c) \in R)$.

Another important aspect of relations is their combination. Because a relation from A to B is defined as a subset of $A \times B$, any two such relations can be combined using standard set operations, such as union, intersection, and difference. Next, in Example 13 we illustrate this case.

Example 13. Consider the sets $A = \{x, y, z\}$ and $B = \{2, 4, 6\}$. The relations $R_1 = \{(x, 2), (y, 4), (z, 2)\}$ and $R_2 = \{(x, 2), (y, 6), (z, 4)\}$ can be combined as follows:

$$R_1 \cup R_2 = \{(x, 2), (y, 4), (y, 6), (z, 2), (z, 4)\}$$

$$R_1 \cap R_2 = \{(x, 2)\}$$

$$R_1 \backslash R_2 = \{(y, 4), (z, 2)\}$$

$$R_2 \backslash R_1 = \{(y, 6), (z, 4)\}$$

4.2 USE CASES

Now 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 will return to these use cases in Chapter 5 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 (KVS): 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. To read a value, clients may execute the *get* operation that returns the value associated with the *key*, if any, or a *null* value. 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. In this example, both key and $value$ are text strings. The key-value service operations are described next.

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

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 internal events or users' activity in a log for further analysis (e.g., forensics, diagnosis, or *post morten* 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, possible 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.

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

4.3 FOUNDATIONAL 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 10 (Replicated service)**.** Let O be a set of operations and M be a set of replicas. 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 redefine this definition in Chapter 5, 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: An SMR in which all replicas implement the operations $O_2 = \{get, put\}$. 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: An SMR in which all replicas implement the operations $O_3 = \{append, retrieve, truncate\}$. 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 n replicas in the system, where $n = f + 1$. In the previous use cases, $n = |M| = 2$, 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 $\mathcal{O}$ to incorporate these arguments.

Definition 11 (Operations with arguments)**.** Let O be a set of operations and let A be a set of arguments, we define $\mathcal{O} = O \times A$ 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 11 can be defined to take as many arguments as necessary.

Example 14. When performing operations in SMR 1, we define an alphabet $A = \{a, b, \ldots, z\}$, and A^* as the set of all strings over A. Then, possible pairs of elements of $\mathcal{O} = O_1 \times A^*$ are: $(acquire, \text{“}lock\text{”}), (release, \text{“}lock\text{”})$, etc.

Example 15. When performing operations in SMR 2, we define an alphabet $A = \{a, b, \ldots, z\}$, 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 $\mathcal{O}_{get} = \{get\} \times A_1$ and the second as $\mathcal{O}_{put} = \{put\} \times A_2$. Possible pairs of elements of $\mathcal{O}_{get}$ are $(get, \text{“}firstkey\text{”}), (get, \text{“}secondkey\text{”})$, etc.; and of $\mathcal{O}_{put}$ are $(put, (\text{“}firstkey\text{”}, \text{“}firstval\text{”}))$, $(put, (\text{“}secondkey\text{”}, \text{“}secondval\text{”}))$, etc.

Example 16. When performing simple 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: $\mathcal{O}_{append} = \{append\} \times A_1$; $\mathcal{O}_{retrieve} = \{retrieve\} \times A_2$; $\mathcal{O}_{truncate} = \{truncate\} \times A_3$. Possible elements of $\mathcal{O}_{append}, \mathcal{O}_{retrieve}$, and $\mathcal{O}_{truncate}$ are: $(append, (put, (\text{“}key\text{”}, \text{“}val\text{”})))$, $(retrieve, (10, 200))$, and $(truncate, 10)$, respectively.

Finally, the replicated service from Definition 10 can be updated to consider operations with arguments.

Definition 12 (Replicated service with arguments)**.** Let $\mathcal{O}$ be a set of operations with arguments and M be a set of replicas. Then $R = \mathcal{O} \times M$ is a relation from $\mathcal{O}$ 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 the execution of operations as a relation between operations and outputs.

Definition 13 (Execution)**.** Let $\mathcal{O}$ be a set of operations with arguments and U be a set of outputs. Then $E \subseteq \mathcal{O} \times U$ is a relation from $\mathcal{O}$ 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 key-value store 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 17. Consider the SMR 3 with operations with arguments as defined in Example 16. When $\mathcal{O}_{append}$ or $\mathcal{O}_{truncate}$ are invoked, it could produce the relations $E \subseteq \mathcal{O}_{append} \times \{true, false\}$ and $E \subseteq \mathcal{O}_{truncate} \times \{true, false\}$, respectively. On the other hand the $\mathcal{O}_{retrive}$ could produce $E \subseteq \mathcal{O}_{retrieve} \times \{[\text{“}firstobject\text{”}, \text{“}secondobject\text{”}, \text{“}thirdobject\text{”}]\}$. Thus, these are possible representations of executions over *append*, *truncate* and *retrieve* operations in a logging service.

Definition 14 (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 18. When a client sends a request $get(\text{“}firstkey\text{”})$ 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: $w_1 = [s_1, 1234, (get, \text{“}foobar\text{”})]$ and $w_2 = [s_2, 1234, (get, \text{“}foobar\text{”})]$.

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 15 (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 $deliver(S, op_i)$ and $send(m\langle op_1, output\rangle)$ illustrated in Figure 4). The arguments represent the multicast group γ, the operation o_i, and a message m containing o_i and its output u_i.

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; 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; DAHLIN, 2004; MARANDI; BEZERRA; PEDONE, 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. Some authors argue that linearizability is overly restrictive for these SMR models and that other consistency criteria, such as *interval-linearizability*, could enable higher concurrency on service state updates while preserving SMR requirements (CASTAÑEDA; RAJSBAUM; RAYNAL, 2015). 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 work, as we consider SMR services as building blocks.

In this chapter, 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.

5 COMPOSING STATE MACHINE REPLICATION

As observed in the literature, composition can be a powerful strategy in software development. For example, it can facilitate reuse (DANG; IBARRA; SU, 2004; LYNCH; 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 chapter, 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 composition in SMR, 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*.

Definition 16 (Composable replicated service)**.** Let O be a nonempty set of operations and M be a nonempty set of replicas. Then $R \subseteq O \times M$ is a relation from O to M that represents a replicated service, where $|R(x)| \geq f + 1$ for all $x \in O$ and f a positive integer.

Note that we can have a similar generalization for Definition 12 to define *composable replicated services with arguments*. It simply requires replacing O with $\mathcal{O}$ in the relation in Definition 16.

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 17 (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.

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 considering the Definition 17. 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 $x \in O$.

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 18 (Replication set with arguments)**.** Let R be a relation from $\mathcal{O}$ to M, with $\mathcal{O} = O \times A$. 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 $x \in O$.

5.1 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; 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; PEDONE; RENESSE, 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.

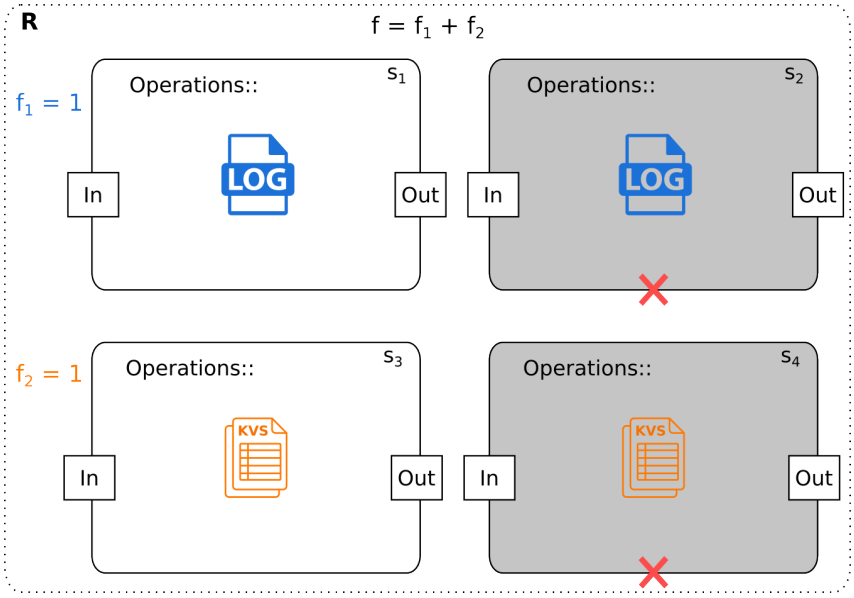
Definition 19 (Composition)**.** Let O_1 and O_2 be sets of operations, M_1 and M_2 be sets of replicas, and let $R_1 \subseteq O_1 \times M_1$ and $R_2 \subseteq O_2 \times M_2$ be two relations. We define the composition to be: $R = R_1 \cup R_2$. As a consequence, we have $O = O_1 \cup O_2$ and $M = M_1 \cup M_2$.

In Definition 19, 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. Once we compose $R = R_1 \cup R_2$, the overall upper-bound of R is now $f = min\{f_1, f_2\}$. 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.

Figure 15 illustrates a scenario in which "well-behaved" failures may occur. The composition R consists of four replicas, where s_1 and s_2 implement the logging service, while s_3 and s_4 implement the KVS. Because each replica implements exactly one service, either logging

or KVS, the failure of one replica from each service, *i.e.*, s_2 and s_4, results in the composition tolerating $f_1 + f_2$ failures. In this case, therefore, R tolerates two failures.

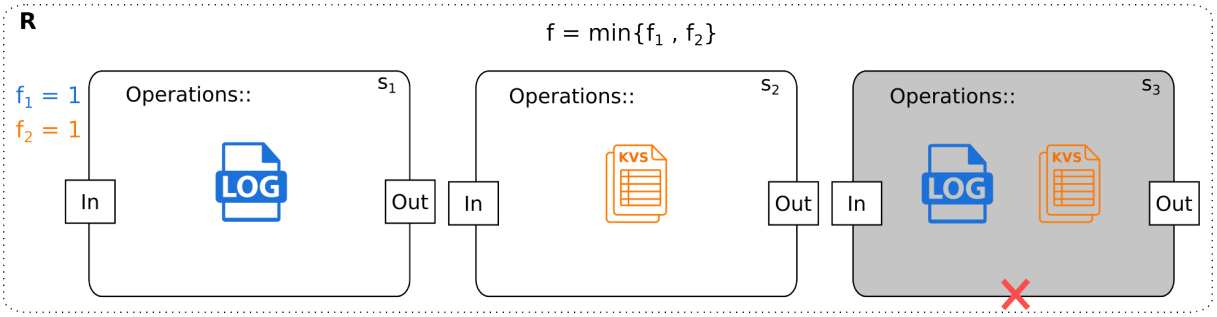
Figure 15 – "Well-behaved" failures



Source: Author (2026).

Figure 16 illustrates the opposite scenario, in which the composition R tolerates $f = min\{f_1, f_2\}$ failures. In this configuration, s_1 implements the logging service, s_2 implements the KVS, and s_3 implements both services. As shown in the figure, s_3 fails, causing both services to fail simultaneously. Once this occurs, each service has reached its failure tolerance limit, and the composition R can support no additional failures.

Figure 16 – Minimum f failures



Source: Author (2026).

Definition 20 (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 $D = \{w_1, ..., w_n\}$ 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 5.1.2 on composition types, where we present a concrete strategy for implementing the selection mechanism in our illustrative use cases.

Given the Definition 19, 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.

5.1.1 Adding SMR operations

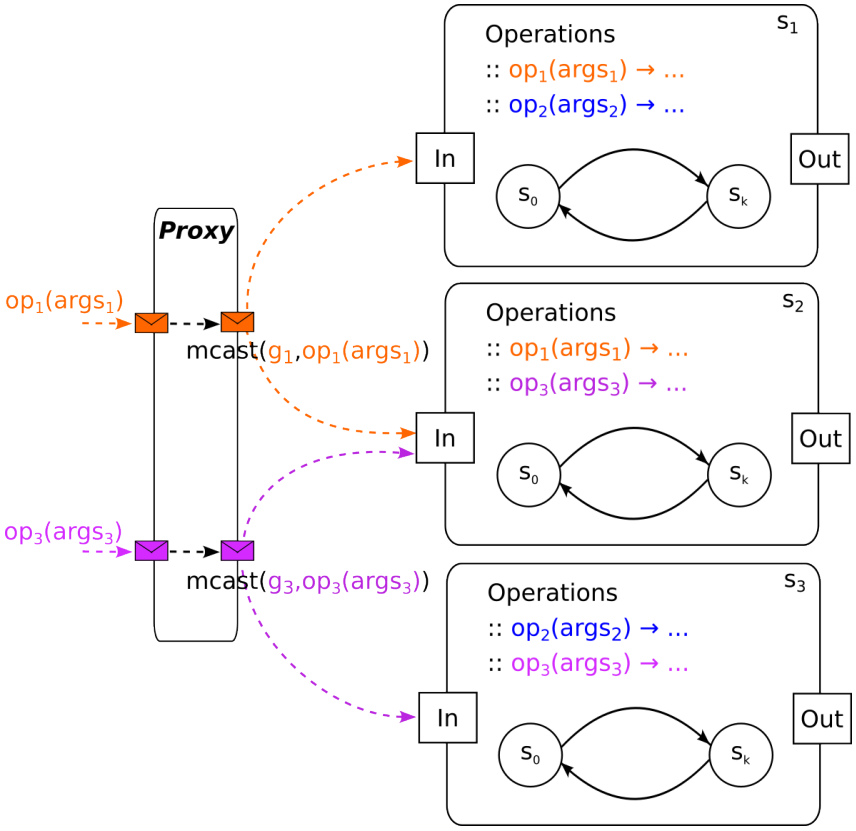
This type of composition allows the addition of operations to a previously defined SMR. Figure 17 illustrates how extending operations from an SMR may occur. s_1 implements only operations op_1 and op_2, s_2 implements op_1 and op_3, and s_3 implements op_2 and op_3. When a client issues operation $op_1(args_1)$, the proxy multicasts op_1 to 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.*, 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. If s_1 crashes, there is still at least one correct server implementing op_1, op_2, and op_3.

Next, in Example 19 we extend the key-value store presented in SMR 2. Note that operations from the O_2 set were added, which will now be executed by a set of servers available in M_2.

Example 19. Now we define a new 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 19, 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.

Figure 17 – Combining SMR operations and replicas.



Source: Author (2026).

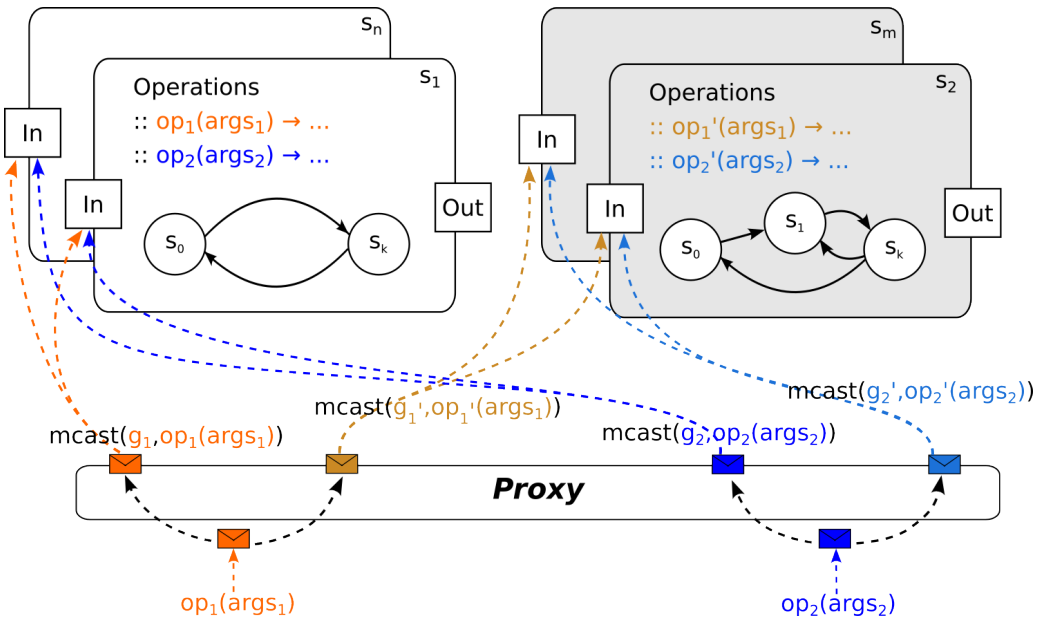
Remark (Joint operations and joint replicas)**.** From Definition 19, if $O_1 \cap O_2 \neq \emptyset$ or $M_1 \cap M_2 \neq \emptyset$, then the composition R is not fully partitioned. There are at least two practical cases to illustrate here. First, different sets of replicas may implement the same 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.

5.1.2 Extending SMR operations' execution

This composition strategy allows different implementations for the same operation. The idea is to add complementary functionalities for a given operation or to implement the same functionality in diverse ways. The latter case is commonly used in design diversity (AVIZIENIS; KELLY, 1984) for increasing fault tolerance. Figure 18 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 18 – Extending SMR operations' execution.



Source: Author (2026).

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 20, 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 (HOHPE; WOOLF, 2004). 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, e.g., $f+1$ output values.

Example 20. 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 presented 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". Simultaneously, the

replicas in SMR 3 invoke the $append$ method, passing the object $put(\text{“}k\text{”},\text{“}v\text{”})$ as an argument, thereby appending the operation to a log of operations, *i.e.*, $(append,(put,(\text{“}k\text{”},\text{“}v\text{”}))) \in \mathcal{O}_{append}$.

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 $D = \{[s_1, 10, (\text{“}get\text{”},\text{“}value\text{”})], [s_2, 10, (\text{“}get\text{”},\text{“}value\text{”})], [s_3, 10, (\text{“}get\text{”},\text{“}value\text{”})], [s_4, 10, (\text{“}append\text{”}, true)], [s_5, 10, (\text{“}append\text{”}, true)], [s_6, 10, (\text{“}append\text{”}, true)]\}$. 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 in Xavier et al. (2020), Scharf, Xavier & Mendizabal (2023), where the authors proposed decoupled logging services for SMR. Although the authors had not intentionally composed SMRs at that time, their proposal fit well in this context.

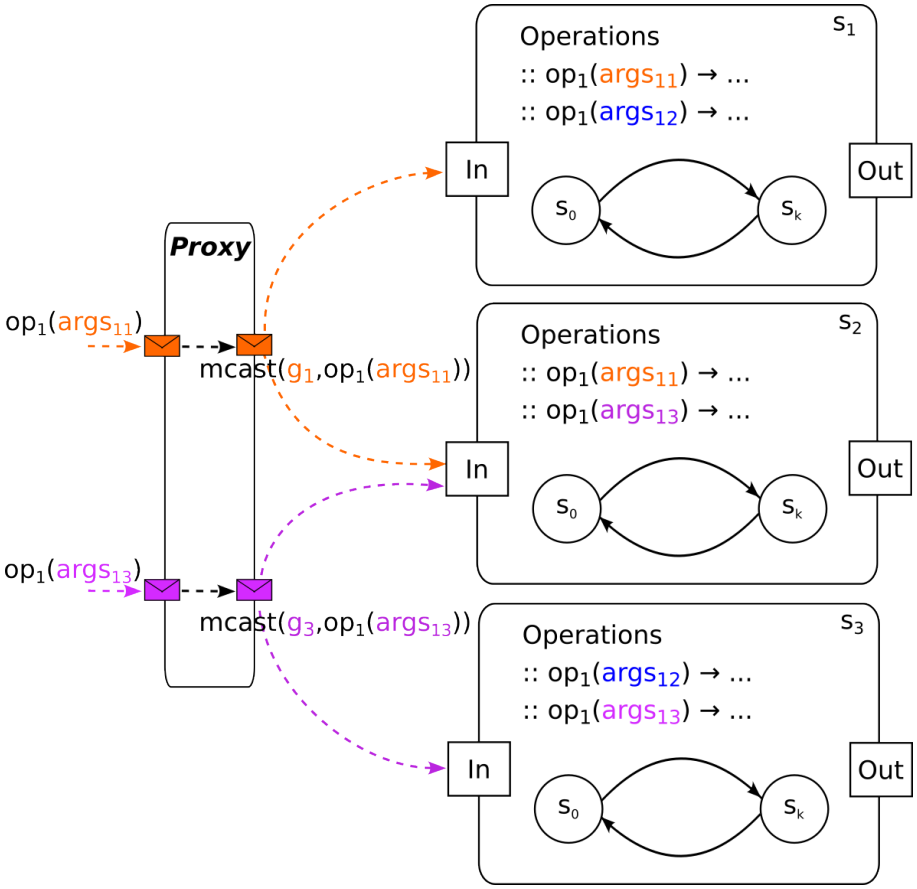
5.1.3 Argument Partition

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 19 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 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 s_2 and s_3 handling it.

Example 21. Here we recall SMR 2 to compose SMRs and implement a key-value store service with partitioned state. Let $A = \{a, b, \ldots, z\}$ be an ordinary alphabet. We define $A^n = A \times A \times \ldots \times A$ 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: $A^n_\alpha = \{\alpha\} \times A \times \ldots \times A$, $\alpha \in A$. Therefore, we have: $A^n = A^n_a \cup A^n_b \cup \ldots \cup A^n_z$. Consider $O = \{get, put\}$, $A_1 = A^n_a, A_2 = A^n_a \times A^*$. We can now define operations with restricted arguments. For instance, $\mathcal{O}_{get} = \{get\} \times A_1$ and $\mathcal{O}_{put} = \{put\} \times A_2$ 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

Figure 19 – Implementing SMR with state partitioning.



Source: Author (2026).

in Bezerra, Pedone & Renesse (2014). Again, although the authors did not intend to apply composition strategies to SMR when defining S-SMR, it also fits well with our approach.

Other sharding-based approaches could benefit from our CSMR. For example, DynaTree (ESLAHI-KELORAZI; LE; PEDONE, 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; BOSTIC; SELTZER, 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; BACIK; MASON, 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.

5.1.4 Removing operations

Composition is not limited to union operation as presented in Definition 19, which focuses on adding more complexity to the system. In some cases, it may be necessary to remove a specific operation due to vulnerabilities, bugs, or other concerns. Therefore, we present a case scenario in which it is required to eliminate unwanted operations from the composition. Definition 21 demonstrates how to build a new composable replicated service while excluding particular operations.

Definition 21. (Operation Removal) Let R be a composable replicated service and op be an operation to be excluded. Then $R' = R \backslash \{(o,m) : o = op\}$ is a new service without the op operation.

For illustration, consider a subset of replicas that perform traditional cryptographic algorithms, and the remaining replicas implement Post-Quantum Cryptography (PQC) algorithms. We have replicas M_1 executing the PQC operations $O_4 = \{sign\_ml\_dsa, verify\_ml\_dsa\}$, which gives the relation $R_1 \subseteq O_4 \times M_1$. Similarly, replicas M_2 execute the traditional cryptography operations $O_5 = \{sign\_rsa, verify\_rsa\}$, resulting in the relation $R_2 \subseteq O_5 \times M_2$.

Therefore, to create a cryptographic SMR composition incorporating both traditional and PQC operations, it would be necessary to take the union of relations R_1 and R_2, thus, $R = R_1 \cup R_2$.

In 2024, the National Institute of Standards and Technology (NIST) issued an internal report (NIST, 2024a) identifying digital signature algorithms vulnerable to quantum attacks. As shown in Table 4, ECDSA and RSA (with 112-bit security) will be deprecated after 2030 and after 2035 ECDSA, EdDSA, and RSA will be fully disallowed.

Table 4 – Quantum-vulnerable digital signature algorithms

Digital Signature Algorithm Family	**Parameters**	**Transition**
ECDSA [FIPS186]	112 bits of security strength	***Deprecated*** after 2030 ***Disallowed*** after 2035
	128 bits of security strength	***Disallowed*** after 2035
EdDSA [FIPS186]	128 bits of security strength	***Disallowed*** after 2035
RSA [FIPS186]	112 bits of security strength	***Deprecated*** after 2030 ***Disallowed*** after 2035
	128 bits of security strength	***Disallowed*** after 2035

Source: Adapted from NIST (2024a)

In order to protect cybersecurity, NIST approved three Federal Information Processing Standards (FIPS) for quantum cryptography, replacing traditional encryption. One of the selected algorithms is Module-Lattice-Based Digital Signature Standard (ML-DSA) (NIST, 2024b).

Given this timeline, vulnerable cryptographic algorithms must be removed from the cryptography SMR composition. This necessitates the elimination of the sign operation with RSA algorithm, as its use will be prohibited after 2035. Following Definition 21, Example 22 demonstrates this mechanism for removing operations from the composition.

Example 22. Consider R the cryptography SMR, where the operation $sign\_rsa$ must be removed. Given $R' = R \backslash \{(o,m) : o = sign\_rsa\}$ the new composition will be $R' = \{(sign\_ml\_dsa, s_1), (sign\_ml\_dsa, s_2), (sign\_ml\_dsa, s_3), (verify\_ml\_dsa, s_1), (verify\_ml\_dsa, s_2), (verify\_ml\_dsa, s_3), (verify\_rsa, s_4), (verify\_rsa, s_5), (verify\_rsa, s_6)\}$.

Figure 20 provides a visual representation of the new configuration for the cryptographic CSMR. The top part of the figure represents the initial relation R, with replicas s_1, s_2, and s_3 executing $sign\_ml\_dsa$ and $verify\_ml\_dsa$, and replicas s_4, s_5, and s_6 executing $sign\_rsa$ and $verify\_rsa$. Applying the operation removal $R' = R \backslash \{(o,m) : o = sign\_rsa\}$ results in the new relation R', where the replicas s_1, s_2, and s_3 remain unchanged, while replicas s_4, s_5, and s_6 now execute only the $verify\_rsa$ operation. This allows the system to continue verifying files signed with the deprecated RSA algorithm, thereby maintaining backward compatibility.

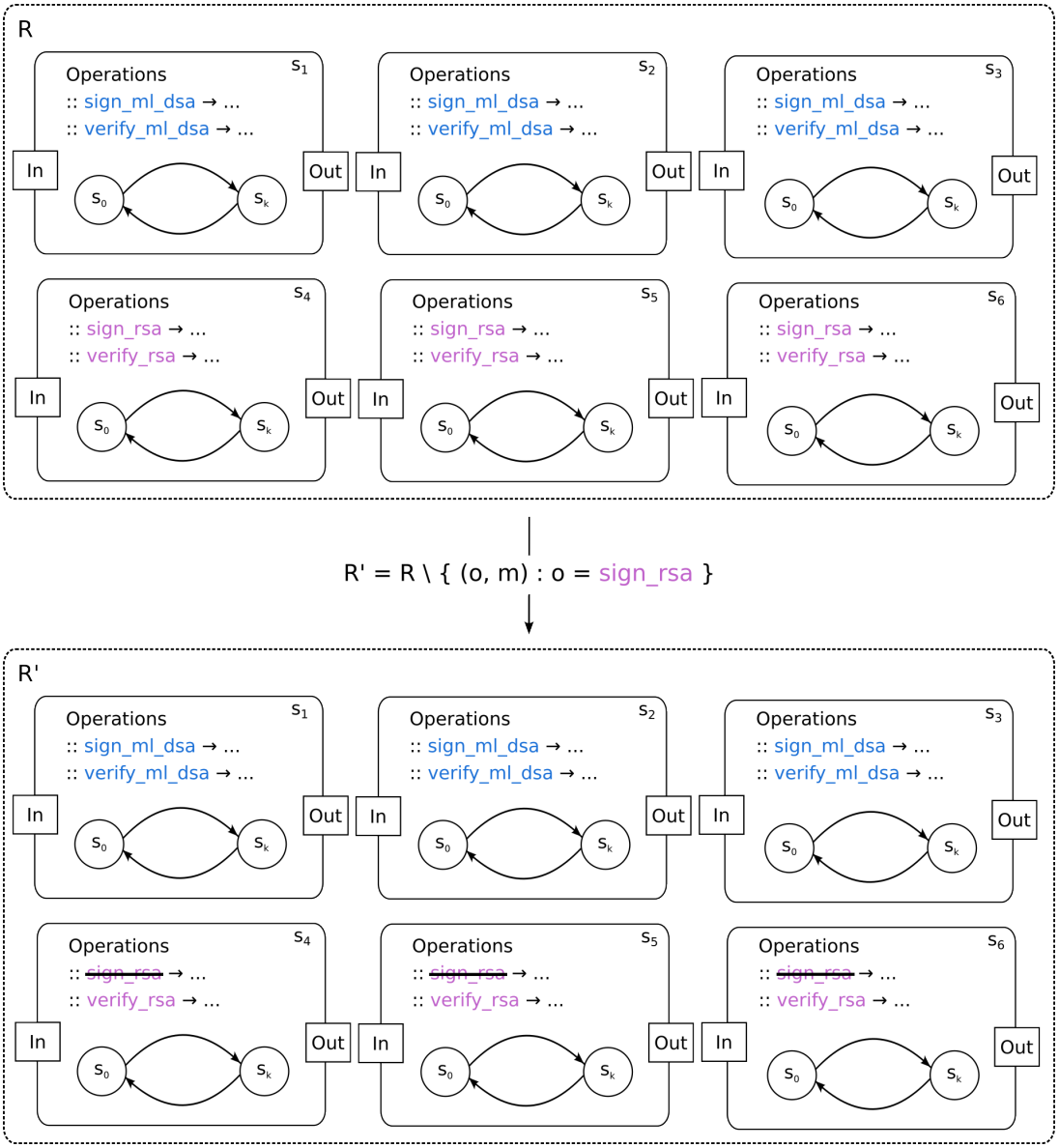
5.1.5 Other examples

The applications of the model described in this work are not restricted 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. Dependable services that provide atomicity, security and other related functionality for distributed systems would also suit well for composing SMR. Instead of building complex, low-level primitives from scratch, developers can leverage existing solutions that handle the complexities of concurrency, fault tolerance, and consistency.

For instance, many applications in Internet of Things (IoT), cryptography, and gaming require randomness. Pseudo-Random Number Generators (PRNG) address this by producing complex, seemingly unpredictable numbers from a given seed value, that is, providing the same seed and initial parameters will always produce the same sequence, ensuring reproducibility.

Similarly, in distributed environments where concurrent participants may access shared data, atomic counters ensure that operations on an integer (e.g., increment/decrement) are executed as a single, indivisible step, preventing race conditions by ensuring safe modifications to shared values. Some commercial examples such as Apache ZooKeeper (HUNT et al., 2010), Redis (SANFILIPPO, 2009) and Hazelcast (HAZELCAST, 2012) provide robust, atomic primitives that developers can use. Developers use Zookeeper for distributed locks, leader election, and managing configuration data. Redis, although is primarily a caching system, implements atomic counters and distributed locks. Hazelcast is an in-memory data grid that

Figure 20 – Removing operations from cryptography SMR composition



Source: Author (2026).

provides a wide range of distributed data structures, including atomic counters, maps, and queues.

Through composition, PRNGs and distributed atomic counters, among other distributed data structures, could be combined to create a versatile framework for clients and programmers adopt as needed. A resulting service API can be defined as follows:

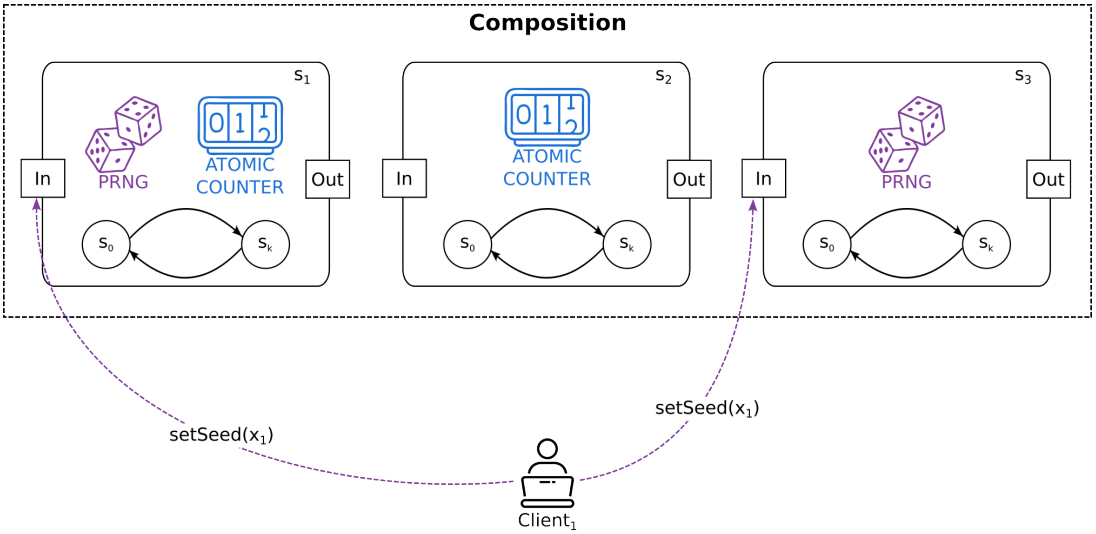
```
void setSeed(int seed)
int random()
int getAndIncrement()
int getAndDecrement()
int get()
void set(int newValue)
```

The operations from the PRNG are *setSeed()*, allowing clients to set an initial seed, and *random()* to generate numbers. The operations from the atomic counter are *getAndIncrement()*, enabling clients to atomically increment the current value by one and retrieve the updated value; *getAndDecrement()*, to atomically decrement the current value by one and retrieve the updated value; *get()*, to retrieve the current value without modification; and *set(int newValue)*, to assign a new value.

Figures 21 through 24 illustrate the operation of this service. All figures depict a composition in which $Client_1$ sends requests interacting with the service. For the sake of legibility, in this design, the proxy is omitted. Thus, in Figures 22 and 24 where replicas send multiple responses to the client, note that a proxy would typically prevent this by implementing a function $f(D)$ over the replica responses (Definition 20) to select a single response to forward. The composition structure consists of three replicas, the replica s_1 which implements both the PRNG and Atomic Counter services, the replica s_2, which implements only the Atomic Counter service and replica s_3, which hosts only the PRNG service.

In Figure 21, $Client_1$ sends a request to initially set the seed x_1 in the PRNG. Note that because this operation does not return a value, replicas s_1 and s_3 do not send a response. Note that if another client sets the same initial seed, it will obtain the same sequence of pseudo-random numbers from the replicas when calling the operation $random()$, which will ensure reproducibility and deterministic execution.

Figure 21 – $Client_1$ sending the request to set an initial seed

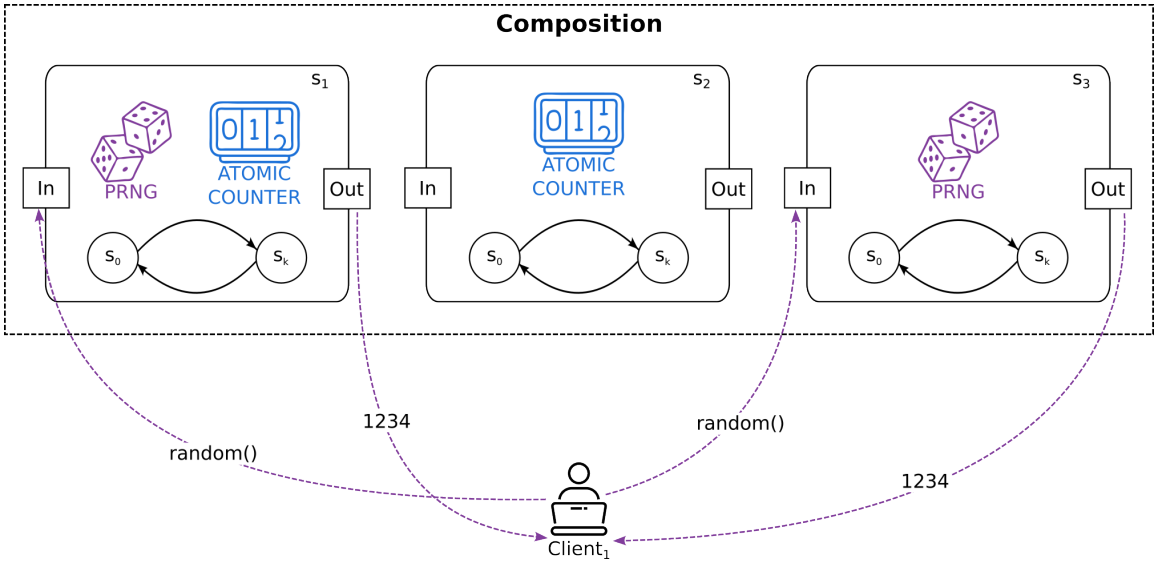


Source: Author (2026).

After the seed is set, Figure 22 shows the invocation of the $random()$ operation to obtain a pseudo-random number. Subsequently, replicas s_1 and s_3 return this number 1234 to the client. The returned number 1234 is a fictional output used solely to simplify the execution trace presented later.

Next, as shown in Figure 23, the $Client_1$ calls the Atomic Counter's *set* operation, using the pseudo-random number obtained previously from the PRNG as the new initial value

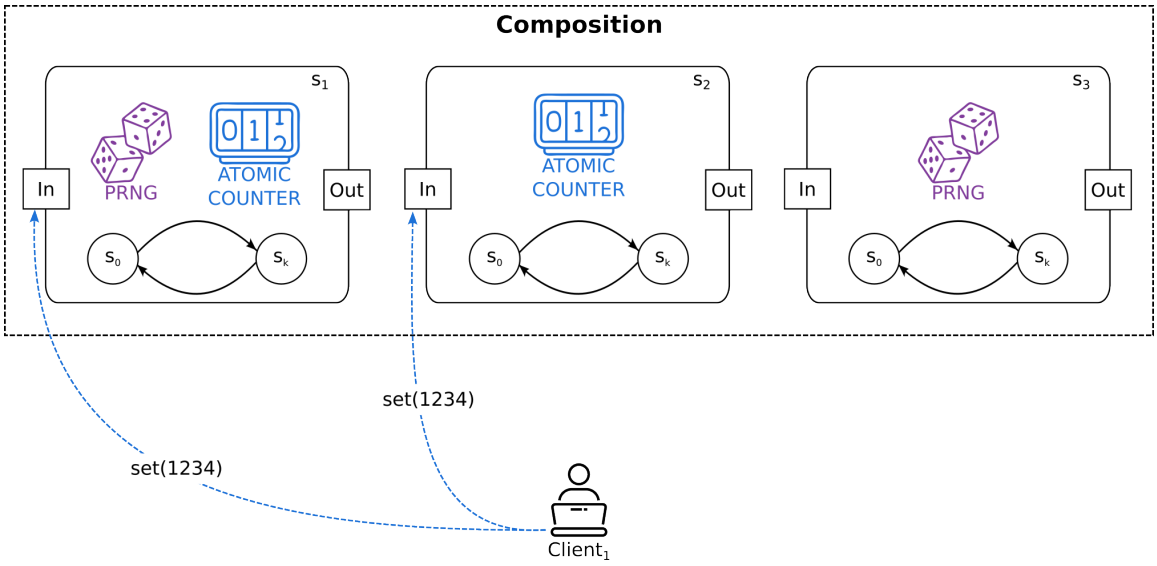
Figure 22 – $Client_1$ sending the request to return a pseudo random number



Source: Author (2026).

for the counter. Consequently, replicas s_1 and s_2 set their Atomic Counter values accordingly (e.g., to 1234). Since this operation also does not return a value, replicas s_1 and s_2 do not send a response to the client.

Figure 23 – $Client_1$ sending the request to set a initial value in the Atomic Counter

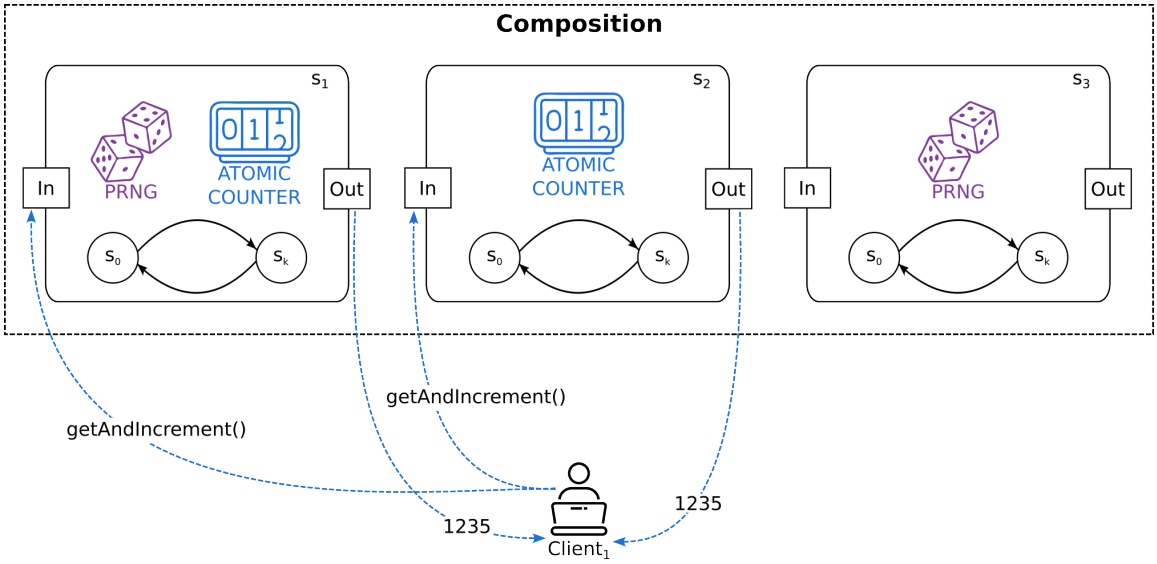


Source: Author (2026).

Finally, in Figure 24, the client sends a request to the Atomic Counter to increment its current value. The incremented value is then returned to the client. Since the counter's initial value was 1234, after the increment operation the returned value will be 1235.

Here, we have presented a unified solution combining both the Atomic Counter and PRNG operations within a single execution. However, these operations could also be used separately or in distinct executions.

Figure 24 – $Client_1$ sending the request to increment the counter and return the current value



Source: Author (2026).

A different perspective is adopting CSMR to integrate diverse implementations for solving a single problem. For example, consider optimization and operations research related problems, such as the shortest path problem, which can be solved using multiple algorithms and heuristics, such as Dijkstra's, Bellman-Ford, A* search, or Floyd-Warshall. One might devise SMR composition combining different algorithms for the same class of problem or setting SMRs with different initialization settings. In this approach, clients' requests delivered to different SMRs would lead to different outcomes. A proxy reduction/selection function would decide which is the best result, *e.g.*, the one with the shortest path, considering the shortest path problem.

Observe that this strategy can be extended to other well-known problems. By integrating multiple approaches into a unified service, we can handle several scenarios — as long as the inner state of the replicas remains unchanged, preventing inconsistencies in the global state. This implies that replicas may produce different outputs for the same input. Such divergence is allowed only if subsequent execution steps do not depend on the specific output value, as a dependency would cause replicas executing different implementations to diverge in state.

The purpose of presenting other examples is to initiate discussion and provide intuition for potential extensions enabled by the flexibility of the composition. A detailed analysis of this class of problems is beyond the scope of the present work.

5.2 COMPOSITION AND CONSISTENCY

In order to avoid violating linearizability or facing design issues, it is important to take certain precautions regarding the construction of the composition and how the services interact. Accordingly, we state that the composition must:

- **Preserve horizontality:** Each service must expose a self-contained interface that does not rely on nested calls between services during any request.
- **Preserve operations affinity:** When operating over the same variable, services must be organized so that operations on the same variable are grouped together, running on the same set of replicas to prevent lack of update.

Example 23 (Invalid Composition)**.** Let $O_1 = \{get\}$ and $O_2 = \{put\}$. Consider the relations $R_1 \subseteq O_1 \times M_1$ and $R_2 \subseteq O_2 \times M_2$. Thus, the composition $R = R_1 \cup R_2$ will produce an invalid composition if there is a replica s such that $s \in M_1$ and $s \notin M_2$.

Example 23 illustrates a scenario in which operations affinity is not preserved, thus producing a lack of update. In this example, there is at least one replica s in R_1 that execute only get operations. Because it does not run any put operations, s yields different results than those that execute both get and put operations. In other words, s does not update values, just retrieves the initial value for a given key. The important observation here is that operations over shared stated variables must be executed by the same set of replicas in the composition.

We state that an SMR constructed using the composition strategies presented in this work ensures strong consistency.

Definition 22 (Subhistory)**.** Given a history H, as defined in Definition 15, a subhistory of a history H, H_γ, is the subsequence of all events in H for a multicast group γ.

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

Proof Sketch. For every $deliver(\gamma_i, o_k)$, there is an immediate reply $send(m\langle o_k, u_k\rangle)$ in the history M_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 dependent on each other.

Proof Sketch. 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 5.1.1, 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 5.1.2 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 5.1.3, 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.

Proof Sketch. 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. Regarding composition, authors in Vale, Shao & Chen (2024) have demonstrated that linearizable concurrent objects can be horizontally composed while preserving linearizability, a property Herlihy & 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.

6 CSMR ARCHITECTURE

This chapter discusses the architectural components of the CSMR and a workflow for specifying and building a CSMR. First, we introduce the deployment phase, detailing the components required to establish the initial CSMR structure. Next, we present an algorithm that provides a possible implementation of the checker tool, which is responsible for verifying that the CSMR specifications do not violate the fault tolerance requirements. We then describe the CSMR architecture itself, illustrating how its components are organized using the extending SMR operations' execution as an example. Finally, we present the declarative API, written in YAML, which specifies how clients use the services.

Since we do not propose a complete service implementation, our approach remains agnostic regarding programming languages, platforms, and libraries. The sole exception is the declarative specification language used to define CSMR. We employ a standard declarative format to demonstrate the framework's expressiveness and intended usage paradigms. Specifically, we selected YAML due to its widespread adoption in microservice orchestration.

6.1 DEPLOYMENT PHASE

The deployment phase involves analyzing the YAML file containing the CSMR specifications and then building the composition based on those specifications. As shown in Figure 25, this process begins with a YAML file containing the base configuration. This configuration is sent to a checker component, which verifies the consistency of the specified composition. The possible outcomes in this step may be (i) the composition is invalid, and the checker generates a report detailing the inconsistencies found and returns it, or (ii) the composition is valid, and it will proceed to the next step, where an orchestrator component consumes the code base to build the replicas with the specified services implemented.

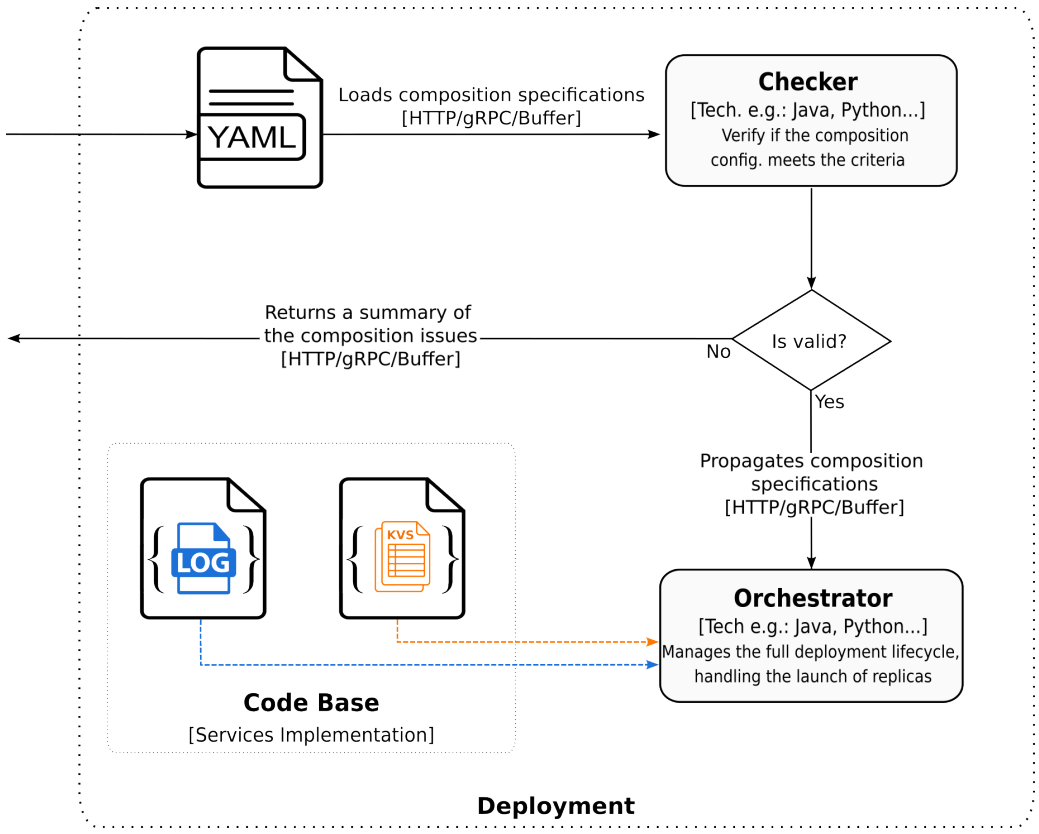
The components of the deployment phase are detailed next.

CSMR Specification (YAML file): Contains the complete specification for the composition. It defines the participating services, including their operation signatures, the replicas with their addresses and ports, and the mapping of which replicas implement which services. A concrete example of this file's structure and more details are provided in Section 6.2.1.

Checker: It is responsible for verifying that a sufficient number of replicas are allocated to execute the operations of the composition. In a fault-tolerant composition, "sufficient" means a minimum of $f + 1$ replicas per service. The checker can be implemented in any language (e.g., Java, Python, Go). Beyond validating the minimum replica count per service, it can also perform other custom validations as required by the system architect. In Section 6.1.1 we discuss a possible approach to implementing a basic checker.

Orchestrator: Builds the replicas according to the specifications in the YAML file. It manages the full lifecycle of the deployment, including the pipeline and other components necessary to bring the composition live. It also consults the code base to obtain the source code

Figure 25 – Deployment CSMR architecture



Source: Author (2026).

for the services.

Code Base: Stores the implementation code for all services. In our illustration, we represent only two source codes, one for the Logging Service and one for the KVS. However, for a composition involving more services, the code base must contain the source code for every service so that the Orchestrator can find the correct code to perform the build.

6.1.1 Checker

As briefly discussed earlier, the checker is responsible for validating the composition before proceeding with the building of replicas and their services. Many validations could be performed within the checker, but one is essential. Recall that the CSMR makes the SMR more flexible, meaning that not all replicas need to execute all operations. While this flexibility is beneficial, it also introduces a critical requirement: the architect must ensure that at least $f + 1$ replicas execute every operation in the composition. This fault-tolerance guarantee is the checker's primary validation.

Beyond this core requirement, the checker can be extended to include additional validations as needed. For instance, it could perform validations based on the application's semantics, verifying that each service correctly executes its operations as required. Of course, this is not a dynamic validation, and it could be very challenging to implement. The difficulty arises because

the checker would then be responsible for knowing the application semantics for every service in the composition and the expected behavior of those services.

Another possible validation could relate to replica load balancing. Based on the number of available replicas, services, and the configured composition, the checker could identify if some replicas are overloaded while others are underutilized. It could then suggest adjustments in the composition configuration in order to optimize their usage.

Next, we present Algorithm 1, a pseudo-code implementation of the checker. It focuses essentially on ensuring the failure tolerance of each operation specified in the composition.

Algorithm 1: Composition Checker

```
Data: Composition (C) and the number of tolerated failures (f)
Result: boolean, set
1  for each operation, replica ∈ C do
2  |  OR[operation] ⟵ OR[operation] ∪ {replica}
3  end
4  for each operation ∈ C do
5  |  if |OR[operation]| = 0 then
6  |  |  return false, OR
7  |  end
8  |  for replicas ∈ OR[operation] do
9  |  |  if |replicas| < f + 1 then
10 |  |  |  return false, OR
11 |  |  end
12 |  end
13 end
14 return true, OR
```

The Algorithm 1 takes as input a composition C and the number of tolerated failures f. This Algorithm returns a boolean value indicating whether the composition is valid ($true$) or invalid ($false$), along with a set object. In this object, each operation maps a set of replicas that execute it. Thus, beyond a simple validity check, the architect can verify both the number of replicas executing each operation and precisely which replicas are responsible for each one.

Recall from Definition 19 that a composition C is a set of ordered pairs (o,m), where o is an operation and m is a replica that executes it. In line 1, each pair $(o,m) \in C$ is destructured and the value o is stored in the variable $operation$, and the value m is stored in the variable $replica$. In line 2, this replica is added to the set object $OR[operation]$ by the union operation. Consequently, all replicas that execute the same operation are grouped under the same key, *i.e.*, the $operation$.

Lines 5-7 perform a basic validation to prevent an empty composition from being considered valid. This is done by checking if the $OR[operation]$ set has a length of zero, which indicates that no replicas were associated with operation key $operation$ in the previous

step (lines 1–3). This condition implies that an empty composition C was passed as input. Alternatively, this could be implemented as an early return by checking $|C| = 0$ first of all, but this is a matter of programmer preference.

In lines 8-12, the algorithm iterates over each set of replicas in the OR map. For each $operation$, in line 9, it checks whether the size of its associated replica set is less than $f + 1$. For instance, if $f = 1$, this condition would be true if only one or zero replicas execute the operation, thereby violating the composition's fault-tolerance requirement. If this condition is met for any operation, the algorithm returns $false$, indicating an invalid composition, along with the OR map for diagnostic purposes.

After the loop in lines 8-12 completes, the algorithm returns true along with the OR set, indicating a valid composition. This return is possible since no operation in the map violates the condition in line 9. Considering the $f = 1$ example, this means every operation is executed by at least two replicas, confirming that the composition satisfies the fault-tolerance condition.

Thus, we have a simple checker to verify the core replication requirement. However, as previously indicated, the implementation could be extended into a more sophisticated verifier to ensure the composition meets additional requirements.

6.2 ARCHITECTURE OVERVIEW

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.

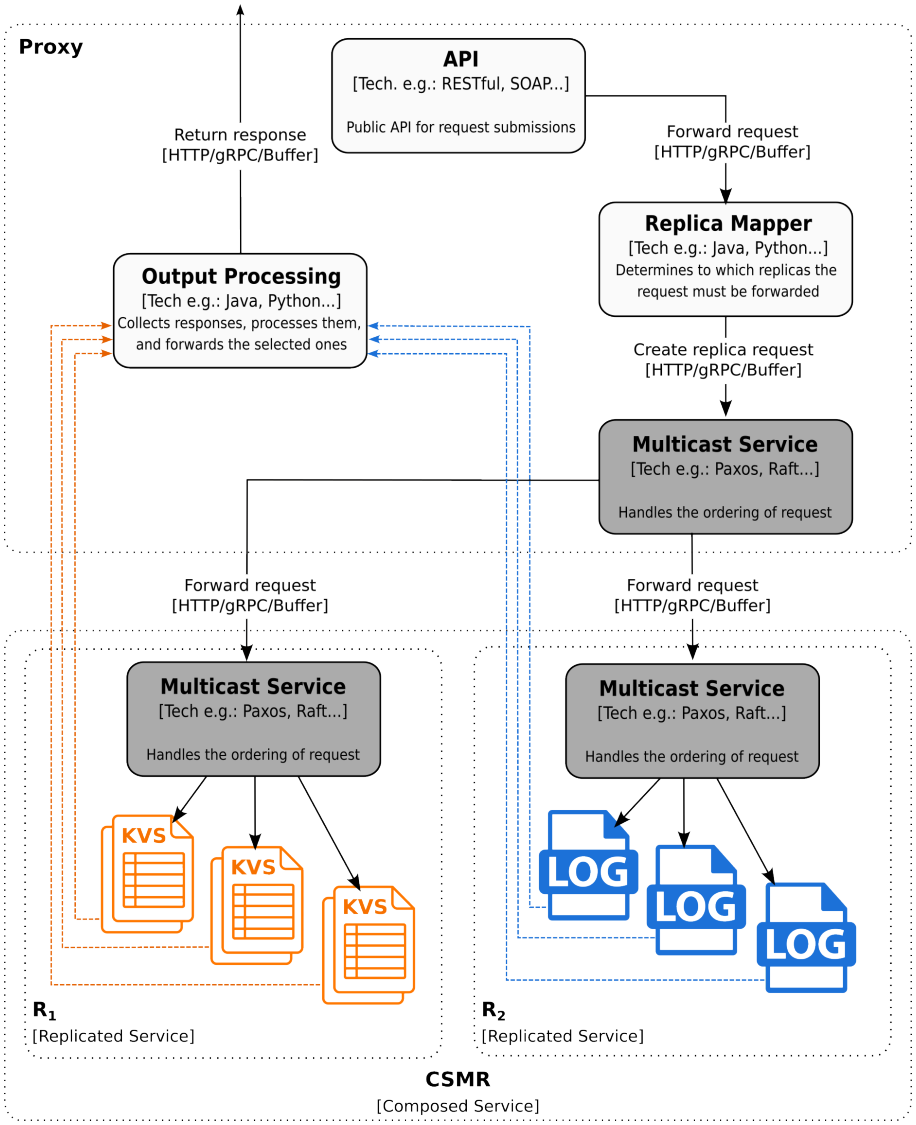
The Composing State Machine Replication (CSMR) architecture is based on two essential modules, as illustrated in Figure 26. The first is the *proxy*, which manages client requests and forwards them to the second module, the *CSMR layer*. The CSMR layer hosts the set of replicated services. These services will effectively execute client requests as if they were independent SMRs. For illustration purposes, Figure 26 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.

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 16). When receiving a request, it identifies the replicas responsible for executing the

Figure 26 – CSMR Architecture



Source: Author (2026).

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 19), 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 20). An application-dependent function must be implemented to determine the correct response to 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.

These components organized in the *Proxy* and the *CSMR* modules constitute the complete architecture of a CSMR. An important caveat concerns the proxy, to prevent it from becoming a single point of failure, the proxy must be implemented using a fault-tolerant design, such as SMR or primary-backup scheme. In doing so, if one proxy replica fails, the remaining replicas can sustain CSMR operation without downtime.

6.2.1 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)` *rightarrow* `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.

7 CONCLUSION

With this work, we took the first steps in addressing compositionality in SMR. We present an initial formal definition of state machine replication and compositionality and introduce some running examples to illustrate different composition strategies. Combining SMRs allows one to use the already implemented services to build more complex and reliable services through composition. This approach provides greater flexibility, benefiting from reuse and modularity. Specifically, we expressed composition by: (i) adding new operations to an existing SMR (Section 5.1.1); (ii) extending the operation's semantics by allowing different state machine replicas to perform complementary steps for the same operation (Section 5.1.2); (iii) providing state partitioning, assigning disjoint state variables to separate SMR instances (Section 5.1.3); and (iv) removing operations (Section 5.1.4) enabling the elimination of an operation from an existing composition.

The running examples highlight the potential for composing SMRs and reveal interesting observations. For instance, the proposed composition strategies preserve consistency. This means once SMR implementations are verified individually against their linearized specification, they can be adopted in a composition without violating linearizability. Furthermore, the SMR composition strategies discussed in the work resemble other strategies from the literature that aim to improve SMR sharing and performance (XAVIER et al., 2020; SCHARF; XAVIER; MENDIZABAL, 2023) as well as scalability (BEZERRA; PEDONE; RENESSE, 2014). Although these approaches were not explicitly framed as composition, they align with our formalized specifications of SMR composition. Now that we have formalized SMR composition and presented it as a method, we expect that innovative applications and strategies for composition will emerge in the SMR field.

Although other composition schemes can be envisioned, they might affect service transparency and consistency, thus requiring careful consideration. This is the case for composition strategies where operations update more than one SMR component simultaneously or share variables. Firstly, this behavior requires the designer to know the specifications of distinct SMRs in advance, reducing transparency and modularity. Secondly, the atomicity of updating independent SMR instances would be broken, potentially undermining consistency.

Finally, we present a theoretical approach for the CSMR architecture along with a declarative API, discussing its components, their responsibilities in a distributed environment, and their organization. We describe each component, propose a generic implementation for the Checker and illustrate the base configuration file for composition in YAML language. This discussion aims to provide a foundation for future implementations and extensions. Although this work is limited to a theoretical approach, it envisions the architecture's practical potential. This proposal is a first step to guide the applicability of the model in future developments using composition in SMR.

7.1 FUTURE WORK

As this work provides a formalization of CSMR, the natural next step is to implement the model and experimentally assess its performance and potential overhead under different workloads. The results could then be compared against other SMR variants in the literature to demonstrate the model's relative advantages and trade-offs.

Regarding the formalization itself, one avenue for exploration is chaining SMRs, where the output of one SMR serves as the input to another, similar to a feedback loop. This would involve an SMR processing an input to produce an output, which in turn becomes a new input for subsequent processing.

Another direction involves extending the checker component with additional validation types. These could include pre-deployment checks such as load balancing validation for operations and replicas, or semantic validation to ensure an operation correctly executes its intended function, thereby preventing faults from propagating through the system.

BIBLIOGRAPHY

ALCHIERI, E. et al. Reconfiguring parallel state machine replication. In: **2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)**. China: IEEE, 2017. p. 104–113.

ALTINBUKEN, D.; SIRER, E. G. **Commodifying Replicated State Machines with OpenReplica**. [S.l.], 2012. Disponível em: https://hdl.handle.net/1813/29009.

ALVES, C. M.; IDALINO, T. B.; MENDIZABAL, O. Extending state machine replication through composition. In: **Proceedings of the 13th Latin-American Symposium on Dependable and Secure Computing**. New York, NY, USA: Association for Computing Machinery, 2024. (LADC '24), p. 231–240. ISBN 9798400717406. Disponível em: https://doi.org/10.1145/3697090.3697106.

ALVES, C. M. et al. Composing state machine replication. **Journal of Internet Services and Applications**, v. 16, n. 1, p. 629–644, Dec. 2025. Disponível em: https://journals-sol.sbc.org.br/index.php/jisa/article/view/5914.

ATTIYA, H.; WELCH, J. **Distributed Computing: Fundamentals, Simulations, and Advanced Topics**. [S.l.]: Wiley-Interscience, 2004.

AVIZIENIS, A.; KELLY, J. P. Fault tolerance by design diversity: Concepts and experiments. **Computer**, v. 17, p. 67–80, 1984.

BATISTA, E. et al. Early scheduling on steroids: Boosting parallel state machine replication. **Journal of Parallel and Distributed Computing**, v. 163, p. 269–282, 2022.

BERGER, C.; REISER, H. P.; BESSANI, A. Making reads in bft state machine replication fast, linearizable, and live. In: IEEE. **2021 40th International Symposium on Reliable Distributed Systems (SRDS)**. [S.l.], 2021. p. 1–12.

BESSANI, A.; SOUSA, J.; ALCHIERI, E. E. State machine replication for the masses with bft-smart. In: **2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks**. Atlanta, GA, USA: IEEE, 2014. p. 355–362.

BEZERRA, C. E.; PEDONE, F.; RENESSE, R. V. Scalable state-machine replication. In: **2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks**. USA: IEEE, 2014. p. 331–342.

BURGOS, A. et al. Exploiting concurrency in sharded parallel state machine replication. **IEEE Transactions on Parallel and Distributed Systems**, v. 33, p. 2133–2147, 2021.

BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In: **Proceedings of the 7th symposium on Operating systems design and implementation**. USA: USENIX Association, 2006. p. 335–350.

CASTAÑEDA, A.; RAJSBAUM, S.; RAYNAL, M. Specifying concurrent problems: beyond linearizability and up to tasks. In: SPRINGER. **Distributed Computing: 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings 29**. [S.l.], 2015. p. 420–435.

CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance and proactive recovery. **ACM Transactions on Computer Systems (TOCS)**, v. 20, p. 398–461, 2002.

CASTRO, M.; LISKOV, B. et al. Practical byzantine fault tolerance. In: **OsDI**. USA: USENIX Association, 1999. p. 173–186.

CHANDRA, T. D.; GRIESEMER, R.; REDSTONE, J. Paxos made live: an engineering perspective. In: **Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing**. USA: ACM, 2007. p. 398–407.

CORBETT, J. C. et al. Spanner: Google's globally distributed database. **ACM Transactions on Computer Systems (TOCS)**, ACM New York, NY, USA, v. 31, n. 3, p. 1–22, 2013.

CUI, H. et al. Paxos made transparent. In: **Proceedings of the 25th Symposium on Operating Systems Principles**. USA: ACM, 2015. p. 105–120.

DANG, Z.; IBARRA, O. H.; SU, J. Composability of infinite-state activity automata. In: **International Symposium on Algorithms and Computation**. Berlim: Springer, 2004. p. 377–388.

DECANDIA, G. et al. Dynamo: Amazon's highly available key-value store. **ACM SIGOPS operating systems review**, v. 41, p. 205–220, 2007.

DÉFAGO, X.; SCHIPER, A.; URBÁN, P. Total order broadcast and multicast algorithms: Taxonomy and survey. **ACM Computing Surveys (CSUR)**, v. 36, p. 372–421, 2004.

DWORK, C.; LYNCH, N.; STOCKMEYER, L. Consensus in the presence of partial synchrony. **Journal of the ACM (JACM)**, ACM New York, NY, USA, v. 35, n. 2, p. 288–323, 1988.

ESLAHI-KELORAZI, M.; LE, L. H.; PEDONE, F. Developing complex data structures over partitioned state machine replication. In: **2020 16th European Dependable Computing Conference (EDCC)**. [S.l.: s.n.], 2020. p. 9–16.

ETCD. **etcd: A distributed, reliable key-value store for the most critical data of a distributed system**. 2013. https://etcd.io/.

GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S.-T. The google file system. In: **SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles**. USA: ACM, 2003. p. 29–43.

HAZELCAST. **Hazelcast | Unified Real–Time Data Platform for Instant Action**. 2012. https://hazelcast.com/.

HERLIHY, M. P.; WING, J. M. Linearizability: A correctness condition for concurrent objects. **ACM Transactions on Programming Languages and Systems**, v. 12, p. 463–492, 1990.

HOHPE, G.; WOOLF, B. **Enterprise integration patterns: Designing, building, and deploying messaging solutions**. [S.l.]: Addison-Wesley Professional, 2004.

HOWARD, H. et al. Raft refloated: Do we have consensus? **ACM SIGOPS Operating Systems Review**, v. 49, p. 12–21, 2015.

HUNT, P. et al. {ZooKeeper}: Wait-free coordination for internet-scale systems. In: **2010 USENIX Annual Technical Conference (USENIX ATC 10)**. USA: USENIX Association, 2010. p. 11.

KIRSCH, J.; AMIR, Y. Paxos for system builders: An overview. In: **Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware**. USA: ACM, 2008. p. 1–6.

KOTLA, R.; DAHLIN, M. High throughput byzantine fault tolerance. In: **DSN**. Florence, Italy: IEEE, 2004. p. 575–584.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. **Communications of the ACM**, v. 21, p. 558–565, 1978.

LAMPORT, L. Paxos made simple. **ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)**, v. 32, p. 51–58, 2001.

LAMPORT, L. Lower bounds for asynchronous consensus. **Distributed Computing**, Springer, v. 19, n. 2, p. 104–125, 2006.

LAMPORT, L.; MALKHI, D.; ZHOU, L. Reconfiguring a state machine. **ACM SIGACT News**, v. 41, p. 63–73, 2010.

LAMPSON, B. W. How to build a highly available system using consensus. In: **International Workshop on Distributed Algorithms**. Berlin, Heidelberg: Springer, 1996. p. 1–17.

LYNCH, N.; MUSCO, C. A basic compositional model for spiking neural networks. In: ____. Cham: Springer Nature Switzerland, 2022. p. 403–449.

LYNCH, N. A. **Distributed Algorithms**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996. ISBN 9780080504704.

MANHATTAN, T. **Manhattan, our real-time, multi-tenant distributed database for Twitter scale**. 2014. 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.; PEDONE, F. Rethinking state-machine replication for parallelism. In: **ICDCS**. Madrid, Spain: IEEE, 2014. p. 368–377.

MASTI, S. **How we built a general purpose key value store for Facebook with ZippyDB**. 2021. https://engineering.fb.com/2021/08/06/core-infra/zippydb/ [Accessed: Jun. 2024].

MENDIZABAL, O. M. et al. Efficient and deterministic scheduling for parallel state machine replication. In: **2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. USA: IEEE, 2017. p. 748–757.

NIST. **International Standards and Recommendations for Post-Quantum Cryptography: Initial Public Draft**. [S.l.], 2024. Initial Public Draft. Disponível em: https://doi.org/10.6028/NIST.IR.8547.ipd.

NIST. **Module-Lattice-Based Digital Signature Standard**. [S.l.], 2024. Initial public draft. Disponível em: https://doi.org/10.6028/NIST.FIPS.204.ipd.

OLSON, M. A.; BOSTIC, K.; SELTZER, M. I. Berkeley db. In: **USENIX Annual Technical Conference, FREENIX Track**. [S.l.: s.n.], 1999. p. 183–191.

PACHECO, L.; DOTTI, F.; PEDONE, F. Strengthening atomic multicast for partitioned state machine replication. In: **Proceedings of the 11th Latin-American Symposium on Dependable Computing**. [S.l.: s.n.], 2022. p. 51–60.

PEREIRA, P. M. et al. A library for services transparent replication. In: **Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing**. USA: ACM, 2019. p. 268–275.

PERONE, M.; KARACHALIAS, G. Crème de la crem: Composable representable executable machines. In: **Proceedings of the 1st ACM SIGPLAN International Workshop on Functional Software Architecture**. USA: ACM, 2023. p. 11–19.

RODEH, O.; BACIK, J.; MASON, C. Btrfs: The linux b-tree filesystem. **ACM Transactions on Storage (TOS)**, ACM New York, NY, USA, v. 9, n. 3, p. 1–32, 2013.

ROSEN, K. **Discrete Mathematics and Its Applications**. 7th. ed. [S.l.]: McGraw-Hill, 2011. ISBN 9780077418939.

SANFILIPPO, S. **Redis – REmote DIctionary Server**. 2009. Blog / projeto de código aberto. Disponível em: https://github.com/redis/redis ou no blog do autor.

SCHARF, J. a. L.; XAVIER, L. G. C.; MENDIZABAL, O. M. 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**. USA: ACM, 2023. p. 90–99.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. **ACM CSUR 1990**, v. 22, p. 299–319, 1990.

SELA, G.; HERLIHY, M.; PETRANK, E. Brief announcement: Linearizability: A typo. In: **Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing**. [S.l.: s.n.], 2021. p. 561–564.

SHVACHKO, K. et al. The hadoop distributed file system. In: **MSST**. USA: IEEE, 2010. p. 1–10.

VALE, A. O.; SHAO, Z.; CHEN, Y. A compositional theory of linearizability. **Journal of the ACM**, ACM New York, NY, v. 71, n. 2, p. 1–107, 2024.

VENN, J. Symbolic logic. **Nature**, Nature Publishing Group UK London, v. 24, n. 613, p. 284–285, 1881.

XAVIER, L. G. et al. Scalable and decoupled logging for state machine replication. In: **Proceedings of the 38th Brazilian Symposium on Computer Networks and Distributed Systems**. Brasil: SBC, 2020. p. 267–280.