Revving Up Replication: Communication and State Management Support for State Machine Replication

Doctoral Dissertation submitted to the

Faculty of Informatics of the Università della Svizzera Italiana (USI)

in a *cotutelle de thèse* agreement with the

Polytechnic School of the Pontifical Catholic University of Rio Grande do Sul (PUCRS)

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

presented by

Eliã Rafael de Lima Batista

under the supervision of

Fernando Pedone (USI) and Fernando Luís Dotti (PUCRS)

February 2026

Dissertation Committee

Dissertation accepted on 11 February 2026

Research Advisor	Co-Advisor
Fernando Pedone (USI)	**Fernando Luís Dotti (PUCRS)**

PhD Program Director

Walter Binder

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Eliã Rafael de Lima Batista
Lugano, 11 February 2026

To my sweet daughter Ellie

Commit to the Lord whatever you do,
and he will establish your plans.

Proverbs 16:3

Abstract

State Machine Replication (SMR) is a fundamental approach for building fault-tolerant distributed systems, ensuring that multiple replicas maintain a consistent state despite failures. However, achieving both strong consistency and high performance remains challenging due to the overhead of replica coordination and state synchronization. This thesis addresses these challenges by proposing novel communication and state management mechanisms for SMR systems.

We present FlexCast, an overlay-based and genuine atomic multicast protocol that combines geographical locality with efficient message delivery through a complete DAG overlay. It reduces latency in geographically distributed deployments and eliminates communication overhead. Moreover, we introduce *quiescence*, our proposed novel atomic multicast property that complements *minimality* by ensuring that processes cease communication once they are no longer involved in any multicast, thereby reinforcing both efficiency and genuineness. Additionally, we introduce a reconfiguration mechanism to FlexCast, that dynamically adapts its overlay to workload locality, maintaining high performance under changing network conditions.

We also address efficient state transfer and verification in replicated systems. We propose two different self-validating clustered data structures that enable efficient, parallel, and independently verifiable state synchronization, reduce tail latencies, improve checkpointing and recovery, and remain robust under Byzantine faults. Integration into a practical SMR framework demonstrates their effectiveness across normal and adversarial scenarios in wide-area networks.

Overall, this thesis provides a comprehensive set of techniques to improve the performance, scalability, and fault tolerance of SMR systems, combining efficient communication, dynamic adaptation, and advanced state management to meet the demands of modern distributed applications.

Acknowledgements

First and foremost, I thank God for granting me life, health, and strength throughout this journey.

I express my deepest gratitude to my wife, Larissa. Your endless love, patience, and support have been my anchor through every challenge. Thank you for believing in me, for standing by my side, and for carrying so many responsibilities to allow me to pursue this dream. To my precious daughter, Eloise, who fills my days with joy and reminds me of what truly matters.

To my parents, Carlos and Mari, for their unconditional love, sacrifices, and prayers that shaped who I am today. Thank you for teaching me the value of dedication, faith, and perseverance from an early age. To my family and friends, who have supported and encouraged me in countless ways, especially my sister, Eliadi, and her husband, Emerson, for their support and affection. I also extend my gratitude to my parents-in-law Djalma and Hilza, and brothers-in-law, Thiago and Leticia, and their children Livia and Felipe, for their encouragement and prayers.

I am sincerely grateful to my advisors, Prof. Fernando Pedone and Prof. Fernando Dotti, for their guidance, expertise, and trust throughout my research. Their mentorship has been invaluable to my academic and personal development. I am further grateful to Prof. Eduardo Alchieri and Prof. Paulo Coelho for their collaborations, insightful discussions, and ideas, which contributed significantly to the development of the majority of the contributions presented in this thesis. I would also like to thank Michele Cattaneo for his collaboration and contributions, particularly for implementing the core of the B+AVL tree, which was essential to the development and evaluation of this work. Finally, I would like to thank the members of the examination committee for their time, careful review, and constructive feedback, which helped improve the quality and clarity of the final version of this thesis. Also, to all my colleagues from the Distributed Systems Laboratory (DSLab), led by Prof. Pedone, for the stimulating research environment, insightful discussions, and daily exchanges that enriched my PhD experience.

Finally, this achievement is not mine alone, but a testament to the love, support, and knowledge I have received from each of you. Thank you.

Preface

All the research efforts I carried out during my Ph.D., including the contributions presented in this thesis and collaborations with other members of our research group, have resulted in the following publications:

- E. Batista, P. Coelho, E. Alchieri, F. Dotti, and F. Pedone. *Robust and efficient replication with CAV data structures*. Submitted to the ACM Symposium on Cloud Computing (SOCC) 2026.
- L. Martignetti, E. Batista, G. Cugola, and F. Pedone. *TRAM: Tree-based atomic multicast on RDMA*. Submitted to the ACM Symposium on Cloud Computing (SOCC) 2026.
- M. Cattaneo, E. Batista, and F. Pedone. *B+AVL trees: towards data structures for robust and efficient blockchain state synchronization*. Proceedings of the 44th International Symposium on Reliable Distributed Systems, 2025.
- E. Batista, P. Coelho, E. Alchieri, F. Dotti, and F. Pedone. *Reconfiguring atomic multicast*. Submitted to IEEE Transactions on Dependable and Secure Computing, 2024.
- E. Batista, P. Coelho, E. Alchieri, F. Dotti, and F. Pedone. *FlexCast: Genuine Overlay-based Atomic Multicast*. Proceedings of the 24th International Middleware Conference, 2023, Pages 288-300.
- E. Batista, E. Alchieri, F. Dotti, and F. Pedone. *Early Scheduling on Steroids: Boosting Parallel State Machine Replication*. Journal of Parallel and Distributed Computing, Volume 163, 2022, Pages 269-282.

Contents

Chapter 1

Introduction

Modern distributed systems are designed to provide uninterrupted services even in the presence of failures. From global-scale cloud infrastructures to blockchain networks and replicated databases, applications today rely on replication to ensure availability, fault tolerance, and performance. Replication allows multiple servers to maintain copies of the same data or service, so that if one fails, others can continue operating. However, ensuring that these replicas behave consistently is far from trivial.

The challenge lies in the fact that distributed systems must operate under uncertainty: messages can be delayed, lost, or reordered; machines can crash or behave maliciously; and yet, clients expect the system to behave as if there were only a single, reliable server. This tension between fault tolerance and consistency has made replication one of the central and most challenging problems in distributed computing.

State Machine Replication (SMR) offers a principled solution to this problem. It allows a system to tolerate failures by replicating a deterministic state machine across multiple nodes, ensuring that all correct replicas execute the same operations in the same order, and thus reach the same state [65, 90]. SMR has been successfully deployed in numerous real-world systems, such as distributed databases [26, 92], large-scale coordination services [91, 93], and blockchain platforms [15, 94], demonstrating its versatility and importance as a foundation for dependable distributed computing.

Despite its broad adoption, implementing SMR efficiently remains a significant challenge. Coordinating replicas requires communication protocols that can ensure agreement on the order of operations even in the presence of failures. Moreover, as system states grow large or evolve dynamically, maintaining and synchronizing these states efficiently across replicas becomes increasingly complex. These challenges directly impact performance, scalability, and fault recovery, key aspects for modern deployments that span wide-area networks and geo-distributed infrastructures.

The work presented in this thesis addresses these challenges by proposing novel techniques for replica communication, state management, and synchronization in SMR systems. In particular, it investigates how to optimize communication overlays for scalability and locality, and how to design data structures that enable fast, verifiable state synchronization. Together, these contributions aim to improve the performance and resilience of SMR systems while preserving their strong correctness guarantees.

1.1 From replication to State Machine Replication

Having introduced the motivation for replication and the role of State Machine Replication (SMR) in dependable systems, we now provide a brief overview of the fundamental concepts behind SMR and the challenges it addresses.

1.1.1 Replication and agreement

Replication aims to improve both the availability and reliability of distributed systems by maintaining multiple copies of data or services. When replicas must all behave consistently, they need to agree on the sequence of operations to execute, a problem known as *consensus*. Achieving consensus is non-trivial because distributed environments are inherently unreliable: messages can be delayed, lost, or reordered, and processes can fail at any time. Classic results such as the FLP impossibility [37] demonstrate that even simple agreement cannot be guaranteed under full asynchrony, which has motivated decades of research on practical fault-tolerant protocols that make realistic timing assumptions.

1.1.2 State Machine Replication

State Machine Replication (SMR) provides a general approach to implementing fault-tolerant services using replication [65, 90]. The idea is to model the service as a deterministic state machine replicated across multiple nodes. Each replica starts in the same initial state and executes the same sequence of operations in the same order. As a result, all correct replicas remain consistent, even if some replicas fail or recover later.

The correctness of SMR relies on two fundamental properties: *safety* and *liveness* [90]. Safety ensures that all correct replicas execute the same operations in the same order, preventing divergent states. Liveness guarantees that every operation requested by a correct client is eventually executed by all correct replicas. A related property, *linearizability* [54], defines the system's observable behavior, requiring that every operation appear to take effect atomically between its invocation and response.

1.1.3 Communication and ordering

Coordinating replicas efficiently requires communication mechanisms that ensure a consistent total or partial order of operations. Protocols such as *atomic broadcast* and *atomic multicast* are fundamental building blocks of SMR. They guarantee that all replicas deliver the same set of messages in the same order, even in the presence of failures and network delays. Designing these protocols is challenging because they must simultaneously achieve high fault tolerance, low latency, and scalability. To address these challenges, researchers have proposed overlay-based communication, genuine multicast, and fault-tolerant multicast protocols (e.g., [22, 50, 51, 89]), each offering different trade-offs between performance and reliability. These mechanisms form the foundation for the communication optimizations explored in Chapter 3.

1.1.4 State management and recovery

Maintaining consistent replica state is equally crucial. When a replica fails and later recovers, or when new replicas join the system, they must synchronize their state with other replicas without significantly affecting performance. Efficient state management structures enable rapid synchronization and validation of large states, which is particularly important in systems with dynamic workloads or large data volumes. Data structures such as Merkle trees [76] and clustered variants (e.g., [24, 41, 46, 57]) enable compact and verifiable state representation. Chapter 4 introduces novel data structures that build on these principles to provide efficient, verifiable, and parallelizable state transfer.

In summary, SMR offers a robust theoretical foundation for fault tolerance, but its practical realization in large-scale or geographically distributed environments remains challenging. This thesis explores how to overcome these challenges through optimized communication overlays and efficient state management mechanisms that improve performance and resilience without compromising correctness.

1.2 Contributions of this thesis

Despite advances in communication protocols and state management structures, SMR systems still face bottlenecks that limit throughput, increase latency, or complicate scaling to larger deployments. Efficient coordination, fault-tolerant multicast, and state synchronization remain active areas of research. This thesis addresses these gaps by proposing novel communication and state management techniques that improve performance while preserving safety (i.e., linearizability) and liveness.

The main contributions of this thesis include novel atomic multicast protocols and overlay-based communication mechanisms for efficient replica coordination; efficient state management structures with practical integration into SMR systems; and comprehensive evaluation demonstrating improved performance, fault tolerance, and scalability over existing approaches. In more detail, the four contributions are as follows.

FlexCast In our first contribution, we proposed FlexCast, an overlay-based and genuine atomic multicast protocol that combines reduced connectivity and geographical locality by using a complete DAG overlay, where each group makes local ordering decisions. We also introduced a new property, quiescence, which refines the minimality property of atomic multicast to prevent unnecessary communication and ensure the integrity of genuine atomic multicast protocols. Our evaluation shows that FlexCast is sensitive to overlay selection for latency but consistently outperforms a distributed genuine protocol and shows comparable or better performance than a hierarchical protocol in typical workloads. Unlike hierarchical protocols, FlexCast introduces no communication overhead while providing global acyclic ordering from local decisions via a history-based mechanism. Its design, implementation, and evaluation using the gTPC-C benchmark, which extends TPC-C with geographical distribution, demonstrate significant latency reductions in geographically distributed deployments.

FlexCast reconfiguration In our second contribution, we extended FlexCast with a reconfiguration mechanism. The proposed reconfiguration scheme allows FlexCast to dynamically adapt its communication overlay to shifts in workload locality, improving performance in changing conditions and making the system well-suited for dynamic environments. Experimental evaluation shows that the protocol effectively adjusts to current workloads, delivering better performance with new configurations.

B+AVL trees On the state management side, our third contribution addressed the challenge of efficient state transfer in blockchain systems, particularly for recovering out-of-sync peers. Existing solutions using snapshots and Merkle-based validation often face limitations in cluster efficiency and proof size. To overcome these issues, we proposed B+AVL trees, a novel data structure that combines the balancing and verification strengths of AVL trees with the space efficiency of B+Trees. Unlike previous approaches, B+AVL trees simplify cluster maintenance by preventing leaf movement across clusters and provide more compact validation proofs. Experimental results show that B+AVL trees maintain consistent performance across insertions and searches, optimize cache utilization through contiguous key storage, and produce smaller, more stable Merkle proofs. They also achieve superior space efficiency compared to most alternatives, particularly for smaller clusters. In state synchronization, B+AVL trees outperform baseline trees in both non-Byzantine and Byzantine settings, thanks to efficient serialization, compact memory layout, and self-verifiable clusters that enable rapid detection and recovery under attack.

SVCSkipList Finally, our fourth contribution introduced the concept of self-validating clustered data structures as a principled approach to improving state management in Byzantine fault-tolerant state machine replication (BFT SMR). This idea is instantiated in the SVCSkipList, a novel data structure that enables efficient, parallel, and independently verifiable state transfer. By tightly integrating state representation and validation into the replication framework, SVCSkipList addresses critical limitations of existing SMR libraries, particularly in the presence of Byzantine faults. Experimental evaluation, conducted in a wide-area network (WAN) environment with integration into the BFT-SMaRt [12] framework, demonstrates robust performance improvements across all execution phases. During normal execution, SVCSkipList achieves throughput comparable to or higher than the baseline, with up to 60% reduction in tail latencies for large state sizes. Checkpointing times are halved for large states (up to 4GB) due to efficient multiversioning. State synchronization is faster and more consistent, both under normal operation and Byzantine attack, with up to 64% lower synchronization times for large replica groups. Independent cluster validation and parallel processing enable rapid recovery under attack, avoiding the cascading retries that impact baseline implementations.

1.3 Thesis organization

The remainder of the thesis is structured as follows: Chapter 2 introduces the system model and background concepts. Chapter 3 focuses on communication in SMR systems, including atomic multicast protocols, FlexCast, and reconfiguration mechanisms. Chapter 4 presents state management in SMR systems, detailing B+AVL trees, SVCSkipLists, and their evaluation. Finally, Chapter 5 concludes the thesis with a summary of contributions and potential future directions.

Chapter 2

System Model and Background

This chapter introduces our system model and revisits fundamental definitions of State Machine Replication (SMR), with particular emphasis on the two core aspects central to our contributions: communication and state management.

State Machine Replication (SMR) is typically defined as a message-passing distributed system composed of an unbounded set of client processes $C = \{c_1, c_2, \ldots\}$ and a bounded set of server processes $S = \{p_1, p_2, \ldots, p_n\}$. A client process represents an external entity that interacts with the replicated service by submitting requests (e.g., application commands) and receiving responses. Clients do not participate in replication or fault-tolerance protocols; instead, they rely on servers to execute their requests consistently. Servers are responsible for coordinating among themselves to ensure the correct emulation of a fault-tolerant state machine. Clients issue requests to the system, while servers execute these requests deterministically and return responses, thereby emulating the behavior of a single fault-tolerant state machine [65, 90].

A process is said to be *correct* if it follows the protocol and never fails, and *faulty* otherwise. Faulty processes may exhibit two fundamental types of failures: *crash failures*, in which a process halts prematurely and takes no further steps, and *Byzantine failures*, in which a process behaves arbitrarily, potentially deviating from the prescribed protocol in unpredictable or even malicious ways. Crash failures are generally easier to tolerate, while Byzantine failures represent a strictly more general and challenging adversarial model.

We assume a message-passing communication model in which processes interact by invoking the primitives $\mathsf{send}(m, q)$ and $\mathsf{receive}(m)$. A process p invokes $\mathsf{send}(m, q)$ to transmit a message m to a destination process q, and a process q invokes $\mathsf{receive}(m)$ to obtain a message sent to it. Communication channels are point-to-point and asynchronous, meaning that message transmission delays are finite but unbounded and messages may be delivered out of order.

We assume reliable communication channels between correct processes. In particular, the system satisfies the following property: if a correct process p sends a message m to a correct process q, then q eventually receives m. Messages are neither created nor duplicated by the communication subsystem; that is, every received message was previously sent by some process, and each message is received at most once. This assumption is standard in SMR system models and allows the protocol to focus on handling process failures rather than communication omissions.

In the context of this thesis, we adopt both failure models depending on the nature of the contribution. Specifically, in our communication-related contributions, we focus on tolerating crash failures, reflecting the model commonly assumed in high-performance SMR systems. In contrast, in our state management contributions, we also consider Byzantine failures, as state transfer and recovery mechanisms must remain correct and secure even in adversarial environments. This dual perspective allows us to explore both efficiency and robustness in different components of SMR.

We assume the system is partially synchronous [35]: it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called the Global Stabilization Time (GST), and it is unknown to the processes. Before GST, there are no bounds on communication and processing delays; after GST, such bounds exist but are unknown.

In SMR, interactions between clients and server replicas must be coordinated to ensure that all correct replicas exhibit the same state evolution. A SMR system must satisfy the safety and liveness properties. The safety property enables the implementation of strongly consistent services, satisfying strong consistency criteria, while liveness guarantees system progress. Formally, these properties can be defined as follows:

- *Safety:* A distributed system satisfies safety if nothing bad ever happens. Formally, safety properties assert that all executions of the system avoid a set of "bad" states. *Linearizability* is the consistency criterion used to implement safety property that ensures each operation appears to take effect atomically at some instant between its invocation and its response, such that the real-time order of non-overlapping operations is preserved and the resulting history is consistent with the object's sequential specification [54].

- *Liveness:* A distributed system satisfies liveness if something good eventually happens. Formally, liveness properties assert that in every execution, progress is eventually made — that is, every request that should be handled will eventually be handled.

Implementing these properties requires the following [90]:

- *Initial state:* All correct replicas start in the same state.
- *Determinism:* All correct replicas that receive the same input in the same state produce identical outputs and resulting states.
- *Coordination:* All correct replicas process the same sequence of operations.

Replica coordination protocols implement on their core a *total order multicast* and/or a *consensus* algorithm [53]. Determinism can impact system performance as it restricts the use of multiple threads and cores [4]. To overcome this limitation, many approaches focus on Parallel State Machine Replication (PSMR), which aims to parallelize the execution of independent requests while maintaining correctness. The key idea is to identify non-conflicting operations that can be executed concurrently on replicas, thus exploiting multi-core architectures for improved throughput.

Several strategies have been proposed to achieve parallel execution in PSMR. In CBASE [62], for example, replicas are augmented with a deterministic scheduler. Based on application semantics, it serializes the execution of conflicting requests respecting delivery order and dispatches requests that do not conflict to parallel execution. Conflict detection is done by a dependency graph, which organizes requests to be processed as soon as possible (whenever dependencies have been executed).

In [74] the authors avoid a central scheduler by statically mapping requests to different multicast groups. Non-conflicting requests are multicast to different groups that partially order requests across replicas. Conflicting requests are multicast to the same group. At a replica, each thread is associated with a group and processes requests as they arrive. Requests delivered by different groups are executed concurrently. Eve [59] uses optimistic approaches that may lead to additional overhead in some cases, where replicas compare the results of optimistic execution using consensus and if they diverge, roll-back and conservatively re-execute requests.

In [5] the authors present an early-scheduling technique, where part of the scheduling decisions are made before requests are ordered. After requests are ordered, their scheduling must respect these restrictions. In [7] we present a resource utilization analysis of this early-scheduling technique, evaluating the impact of such restrictions, showing that threads may be idle while pending independent requests are available to be executed, leading to poor processor utilization. Hence, in [8] we extend the early-scheduling technique by incorporating work-stealing and other synchronization mechanisms to achieve a more balanced workload and improve execution performance.

Another approach to improving the performance of SMR is to weaken the total order requirement of requests at the replicas. In particular, only conflicting requests must be ordered consistently at the replicas. Generalized Paxos [68], Generic Broadcast

[83], Egalitarian Paxos [77], and Mencius [73] are examples of protocols that adopt this approach.

By exploiting these techniques, PSMR can significantly increase throughput while preserving the safety and linearizability guarantees of traditional SMR. However, the design must carefully balance parallelism and determinism, as excessive concurrency can lead to complex synchronization and rollback overheads. These optimizations are orthogonal to the approaches described in this thesis and can be combined with them to further improve performance.

2.1 Atomic multicast

Atomic multicast is a fundamental communication abstraction in reliable distributed systems, encapsulating the complexity of propagating and ordering messages consistently across subsets of processes. With atomic multicast, a client can multicast a message to different groups of servers, and the abstraction guarantees that all correct destinations deliver the message in a consistent order. In the following, we formalize these reliability and ordering guarantees.

To implement fault-tolerant atomic multicast, servers are organized into *groups of processes*. Each group behaves as a single logical process from the perspective of the multicast abstraction, masking individual server failures inside the group. Replication within a group ensures that the group as a whole can continue to participate in multicast even if some of its members fail. Formally, we define the set of server groups as $\Gamma = \{G_A, G_B, \ldots, G_N\}$, where for every $g \in \Gamma$, $g \subseteq S$. Additionally, groups are non-empty and disjoint [22, 50, 51, 89], ensuring that each server belongs to exactly one group.

A client atomically multicasts an application message m to a set of groups by invoking the primitive $\mathsf{multicast}(m)$, where $m.sender$ denotes the process that calls $\mathsf{multicast}(m)$, $m.id$ is the unique identifier of the message, and $m.dst$ is the set of destination groups to which m is multicast. A server delivers a message m by invoking the primitive $\mathsf{deliver}(m)$. If $|m.dst| = 1$, we say that m is a *local* message; if $|m.dst| > 1$, we say that m is a *global* message.

We define the relation $\prec$ on the set of messages server processes deliver as follows: $m \prec m'$ iff there exists a process that delivers m before m'. If $m \prec m'$ or $m' \prec m$, we say that there is a dependency between m and m'.

Atomic multicast satisfies the following properties [52]:

- *Validity*: If a correct process p multicasts a message m, then eventually all correct server processes $q \in g$, where $g \in m.dst$, deliver m.

- *Agreement*: If a process p delivers a message m, then eventually all correct server processes $q \in g$, where $g \in m.dst$, deliver m.

- *Integrity*: For any process p and any message m, p delivers m at most once, and only if $p \in g$, $g \in m.dst$, and m was previously multicast.

- *Prefix order*: For any two messages m and m' and any two server processes p and q such that $p \in g$, $q \in h$ and $\{g, h\} \subseteq m.dst \cap m'.dst$, if p delivers m and q delivers m', then either p delivers m' before m or q delivers m before m'.

- *Acyclic order*: The relation $\prec$ is acyclic.

In a genuine atomic multicast protocol, only the sender and the destinations of a message coordinate to order the message. A genuine atomic multicast protocol does not depend on a fixed group of processes and does not involve processes unnecessarily. More precisely, a genuine atomic multicast algorithm should guarantee the following property [51].

- *Minimality*: If a process p sends or receives a message in run R, then some message m is multicast in R, and p is $m.sender$ or in a group in $m.dst$.

2.2 State management

Satisfying initial state in a SMR system is not trivial in practical scenarios where replicas fail (or are turned off for maintenance) and later recover with a possibly outdated state. Consequently, replication state management is a vital component, ensuring consistent, reliable handling of state across replicas to support fault tolerance. State management also enables the implementation of durability [11] and reconfigurations [12] that incorporate new replicas into the system. Additionally, lagging replicas can utilize this component as a catch-up mechanism to update their state directly from other replicas, thereby eliminating the need to execute all missed operations.

State management requires methods like logging, checkpointing, and state synchronization. Replicas maintain a log of executed operations, allowing replicas to restore a consistent state by replaying the log obtained from other replicas. To avoid unbounded log growth, replicas periodically create a checkpoint and truncate the log, discarding operations that happened before the checkpoint. To update its state, a replica installs a valid checkpoint and processes logged commands obtained from other replicas after the checkpoint. These functionalities are exposed through an interface that allows applications to manage state by creating checkpoints (or snapshots) and/or restoring replica state. Usually, developers interact with these capabilities via SMR library APIs, which simplify these operations. For instance, in the BFT-SMaRt [11, 12] library, replication state management is part of its modular architecture. Applications must implement an interface for saving and loading state, which is used by the state management module to periodically generate state snapshots that capture the application's current state.

Ensuring consistent and correct state transfer in the presence of Byzantine failures is challenging, as some replicas may send incorrect, incomplete, or outdated state data. BFT-SMaRt addresses this issue using quorum-based validation to ensure that only valid state snapshots and logs are applied to replicas executing a state synchronization. In this process, a replica r executing a state synchronization requests the state from a quorum (typically $f + 1$ replicas for f Byzantine faults). One replica sends the state snapshot, while others provide portions of the log and cryptographic hashes of the same snapshot for validation [11, 12]. Replica r then verifies the snapshot hash against those sent by the quorum. If the hashes do not match, the state is rejected, preventing malicious tampering. This approach leverages the guarantee that at least $2f + 1$ replicas are non-faulty, ensuring that replica r only applies correct state data while ignoring malicious responses.

However, this approach has a key limitation in Byzantine scenarios: replica r can detect a corrupted state only after fully downloading it. This significantly impacts performance, as repeated downloads from malicious replicas waste bandwidth and prolong state synchronization time. Moreover, since the entire state is fetched from a single replica, the transfer speed is constrained by the bandwidth and responsiveness of that specific replica, further amplifying the cost of failed synchronization attempts.

A promising approach, commonly employed in blockchains (a specialized form of SMR), involves integrating the application state directly into the replication library and representing it with clustered, self-validating data structures. This method also relieves the application programmer from the burden of state management by providing a straightforward interface for storing and manipulating the data that constitutes the state. Moreover, such clustered data structures provide an efficient means to organize and validate data, ensuring that even in the presence of malicious or faulty replicas, the system can uphold a consistent state.

Chapter 3

Communication in SMR Systems

State Machine Replication (SMR) systems rely on communication primitives to ensure consistency among replicas in the presence of failures. These communication mechanisms underpin the fundamental requirement of SMR: that all correct replicas execute the same sequence of commands in the same order, despite asynchrony and faults in the system [66, 67, 90]. This chapter examines the main communication abstractions used in SMR, emphasizing their role in ensuring consistent and efficient message ordering and delivery, and builds upon these foundations to introduce novel communication protocols that address existing limitations.

We begin by discussing *Atomic Broadcast*, the classic abstraction that guarantees total order delivery of messages to all participants. Although atomic broadcast is sufficient for many replication scenarios, it imposes unnecessary overhead in systems where commands need to be delivered only to subsets of replicas. To address such scenarios, *Atomic Multicast* extends the broadcast abstraction by enabling messages to be sent to specific groups of processes while maintaining consistent ordering among overlapping groups. We describe fundamental algorithms implementing atomic multicast, including Skeen's algorithm [13] and ByzCast [22], and present a classification of existing atomic multicast protocols based on their design approaches and fault models.

Then we present FlexCast [9], our proposed protocol and the first genuine overlay-based atomic multicast for SMR systems. Genuineness captures the essence of atomic multicast in that only the sender of a message and the message's destinations coordinate to order the message, leading to efficient protocols [51]. Overlay-based protocols restrict how process groups can communicate. Limiting communication leads to simpler protocols and reduces the amount of information each process must keep about the rest of the system. FlexCast implements genuine atomic multicast using a complete directed acyclic graph (C-DAG) overlay. Morevoer, we introduced a new property, quiescence, which refines the minimality property to prevent unnecessary communication and ensure the integrity of genuine atomic multicast protocols. We experimentally evaluate FlexCast in a geographically distributed environment using our proposed gTPC-C, a variation of the TPC-C benchmark that takes into account geographical distribution and locality. We show that, by exploiting genuineness and workload locality, FlexCast outperforms well-established atomic multicast protocols without the inherent communication overhead of state-of-the-art non-genuine multicast protocols.

As a follow-up to FlexCast, we present its reconfiguration mechanisms, which further extend the benefits of genuine overlay-based atomic multicast. FlexCast's design not only ensures less communication overhead but also allows the system to adapt dynamically to changing workload localities through reconfiguration of the overlay. We model the reconfiguration process as an optimization problem and implement a protocol to adjust the overlay in response to workload shifts, enhancing the system's suitability for dynamic environments. Our experimental evaluation demonstrates that, by reconfiguring the overlay, our reconfiguration protocol can substantially reduce latency for destination groups representing the bulk of the workload.

The implementation of FlexCast, including its reconfiguration extension, is publicly available to support reproducibility and further experimentation. The complete codebase can be accessed through the author's GitHub repository[1].

3.1 Atomic broadcast

Atomic broadcast is best understood by first introducing simpler broadcast primitives that incrementally strengthen communication guarantees. This progression highlights how reliability and ordering properties culminate in the total order provided by atomic broadcast. We discuss how atomic broadcast is tightly linked to consensus, as the ability to totally order messages across processes enables agreement on a single decision value in distributed systems.

3.1.1 Baseline broadcast primitives

Broadcast communication abstractions are defined by two basic primitives:

- `broadcast(m)`: a process sends message m to all processes in the system.
- `deliver(m)`: a process delivers message m.

Different communication abstractions build upon these primitives by strengthening their guarantees [52].

[1] https://github.com/elbatista/flexcast

Reliable broadcast Reliable broadcast ensures that messages broadcast by processes are eventually delivered by all correct processes, without duplication or fabrication. It provides the following properties:

- **Validity:** If a correct process broadcasts a message m, then it eventually delivers m.
- **Agreement:** If a process delivers a message m, then all correct processes eventually deliver m.
- **Integrity:** For any message m, every process delivers m at most once, and only if m was previously broadcast by $m.\texttt{sender}$.

FIFO broadcast FIFO broadcast extends reliable broadcast by guaranteeing that messages sent by the same sender are delivered in the order they were sent. In addition to the reliable broadcast properties, it ensures:

- **FIFO Order:** If a process broadcasts message m before m', then no process delivers m' unless it has delivered m.

Causal broadcast Causal broadcast strengthens FIFO broadcast by preserving causality between messages, as defined by the happens-before relation [65]. An execution of a broadcast or deliver primitive by a process is called an *event*. We say that event e causally precedes event f, denoted $e \rightarrow f$, if: *(i)* A process executes e and f in that order; or *(ii)* e is the broadcast of message m and f is the delivery of m; or *(iii)* there exists an event h such that $e \rightarrow h$ and $h \rightarrow f$.

The causal broadcast abstraction ensures:

- **Causal Order:** If the broadcast of a message m causally precedes the broadcast of a message m', then no process delivers m' unless it has previously delivered m.

3.1.2 Atomic broadcast definition

Atomic broadcast extends reliable broadcast by providing a total ordering of messages, ensuring all processes deliver messages in the same order. Formally, atomic broadcast provides:

- **Total Order:** If processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

Atomic broadcast guarantees that all correct processes deliver messages in the same order, regardless of the order in which they were sent. This total order is crucial for ensuring consistency in distributed systems, especially in state machine replication. Atomic broadcast is typically implemented under the assumption of a partially synchronous system model, where the system may initially behave asynchronously—with unbounded message delays and process execution speeds—but after some unknown Global Stabilization Time (GST), bounds on communication and processing hold.

3.2 Atomic multicast

Atomic multicast is a key communication abstraction in reliable distributed systems, ensuring messages sent to multiple groups are delivered consistently. Atomic broadcast can be regarded as a special case of atomic multicast in which every message is multicast to all processes in the system. Therefore, atomic broadcast can be trivially used to implement atomic multicast by broadcasting every message system-wide and restricting its delivery to intended recipients. However, such a solution has no practical interest due to inefficiency.

Therefore, we now characterize atomic multicast protocols based on three aspects: genuineness and quiescence (§3.2.1), fault tolerance (§3.2.2), and process communication (§3.2.3). We finish this section by classifying atomic multicast protocols based on these aspects (§3.2.4).

3.2.1 Genuine and quiescent atomic multicast

Atomic multicast can also be implemented by delegating the responsibility of ordering messages to a distinguished group of processes. Every atomically multicast message m is first sent to the distinguished group, which orders m with respect to other multicast messages, and then propagates m to its destination groups. Although simple, this solution does not capture the spirit of atomic multicast.

Instead, we are interested in atomic multicast protocols that are "genuine". In a genuine atomic multicast protocol, only the sender and the destinations of a message coordinate to order the message. A genuine atomic multicast protocol does not depend on a fixed group of processes and does not involve processes unnecessarily. More precisely, a genuine atomic multicast algorithm should guarantee the following property [51].

- **Minimality**: If a process p sends or receives a message in run R, then some message m is multicast in R, and p is $m.\texttt{sender}$ or in a group in $m.\texttt{dst}$.

Minimality restricts communication to comprehend processes involved in the multicast of a request as the request's sender or destination. Once a process is the sender or destination of a request, however, the process can communicate freely with any other process without violating minimality. This behavior obviously contradicts the spirit of minimality and genuineness. We avoid such undesired behaviors by introducing the following additional property.

- **Quiescence**: If there is a time t_1 in run R after which no request r is multicast such that p is r.sender or a destination in r.dst, then there is a time $t_2 \geq t_1$ such that after t_2 p does not send or receive any messages.

Notice that minimality and quiescence are complementary: while minimality defines when processes can exchange messages, quiescence states when processes should stop communicating. Our proposed FlexCast protocol (§3.3) is a genuine atomic multicast protocol that satisfies both properties, as we demonstrate in §3.3.3.

3.2.2 Fault-tolerant atomic multicast

The first atomic multicast protocol is attributed to D. Skeen and dubbed "Skeen's protocol" [13]. The protocol is genuine but does not tolerate failures. A multicast message m is first propagated to all its destinations. Upon receiving the message, a destination assigns the message a local timestamp and sends the timestamp to the other message destinations. When a destination has received all local timestamps for the message, it computes the message's final timestamp as the maximum among all of the message's local timestamps. A message is only delivered after it has a final timestamp and messages are delivered in order of their final timestamps.

Several fault-tolerant atomic multicast protocols are based on Skeen's ordering technique of exchanging timestamps (e.g., [23, 39, 51, 86]). In these fault-tolerant atomic multicast protocols, messages are addressed to sets of groups of processes, instead of a set of processes, as in the original Skeen protocol. Although some processes in a group may fail, each group acts as a reliable entity, whose logic is replicated within the group using state machine replication [90].

3.2.3 Overlay-based atomic multicast

Most atomic multicast protocols assume that processes can communicate directly with each other. In other words, the *communication graph*, a directed graph where vertices represent groups of replicated server processes and an edge from group g to group h means that g can send messages to h, is complete. An alternative approach is to restrict communication by means of an overlay that determines how groups can communicate.

Restricting communication can lead to simpler atomic multicast algorithms, as in a tree overlay. Moreover, if communication needs to be authenticated, as in Byzantine fault-tolerant protocols, a partially connected communication graph requires fewer keys to be maintained and exchanged between processes. Finally, a complete communication graph is a reasonable assumption in systems that run within the same administrative domain (e.g., Google's Spanner [26]). In some contexts, however (e.g., decentralized systems), multiple entities from different administrative domains collaborate but do not wish to establish connections with all other domains.

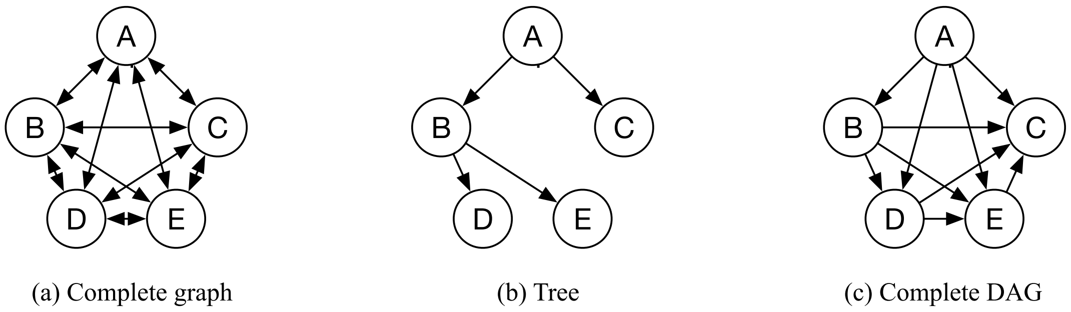


Figure 3.1. Three communication graphs used in atomic multicast algorithms involving groups $A, B, ..., E$: (a) a complete graph (the most common approach), (b) a tree, and (c) a complete directed acyclic graph or C-DAG (the approach we propose later). In the graphs, directed edge $g \rightarrow h$ means that group g can send messages to group h, and h can receive messages from g but not send messages to g.

Figure 3.1 shows three cases of interest. All genuine atomic multicast algorithms we are aware of assume a complete communication graph (Figure 3.1 (a)). A tree (Figure 3.1 (b)) is the minimum overlay needed by any atomic multicast protocol to support an arbitrary workload (i.e., messages can be multicast to any set of groups), as removing one edge from the overlay results in a partitioned graph. ByzCast [22] is an atomic multicast protocol that uses a tree overlay. To order a message m, m is first ordered within the lowest common ancestor group of groups in m.dst, in the worst case the root of the overlay tree. Then, m is successively ordered by the lower groups in the tree until it reaches all groups in m.dst. ByzCast's main invariant is that lower groups in the tree preserve the order induced by higher groups. Although simple, ByzCast is not genuine since a message may need to be ordered by a group that is not in the destination set of the message. For example, in Figure 3.1 (b), a message multicast to groups B and C will first be ordered at A, and then propagated and ordered by B and C.

C-DAG: a discussion on genuineness To motivate our approach presented later, we argue informally that a complete directed acyclic graph (C-DAG) is a necessary condition for genuineness in overlay-based partially synchronous systems. To understand why, notice that in a C-DAG, each vertex has either an ongoing or outgoing edge with any other vertex. If there is no edge between groups g and h in the graph, then a message addressed to g and h needs to involve one or more groups not part of the message's destination, which violates minimality. [2]

Similarly to a tree overlay, in a C-DAG a multicast message can be initially ordered at the lowest common ancestor (`lca`) group g among the message's destinations and then be propagated by g to the remaining destinations. In a C-DAG, however, the `lca` of a message can propagate the message directly to all remaining destinations of the message, respecting minimality. Differently from a tree overlay, in a C-DAG a group cannot deliver a message after it receives the message from its ancestor since groups can have more than one incoming edge and messages can arrive at a group from different ancestors. A group with multiple incoming edges must determine the correct order to deliver messages coming from its ancestors.

For example, in Figure 3.1 (c), group C can receive messages from A and B. In an execution in which message m_1 is addressed to groups A, B and C, and message m_2 is addressed to groups B and C, group C will receive m_1 from A and m_2 from B. If B delivers m_2 before m_1, then C must respect this order, even if it may receive m_1 before m_2. Otherwise, C would violate atomic multicast's acyclic order property.

3.2.4 A classification of atomic multicast protocols

As previously noted, many atomic multicast protocols build upon Skeen's ordering technique, extending it to provide fault tolerance [23], [39], [50], [69], [86]. Since in all these protocols processes communicate directly with one another, we refer to this class of protocols as *distributed* atomic multicast protocols (see Table 3.1).

Another class of protocols is based on the assumption that processes communicate through a distinguished group of processes, which orders messages and propagates them to their destinations (e.g., [22, 42]). In this class, an important invariant is that lower groups in the tree preserve the order induced by higher groups. Hereafter, we refer to this class of protocols as *hierarchical* atomic multicast protocols.

Finally, our proposed approach FlexCast is a genuine atomic multicast protocol that uses a complete directed acyclic graph (C-DAG) overlay, instead of a tree as in all other known protocols in the *hierarchical* class. In FlexCast, a multicast message is

[2]This observation assumes that to order a message addressed to two ore more groups, these groups need to communicate. It is easy to see that this holds in a partially synchronous system: if g and h do not communicate, then they cannot agree on the order of messages addressed to both g and h.

Class	**Type**	**Examples**
Distributed	genuine	[13, 23, 30, 39, 50, 69, 86]
Hierarchical	non-genuine	[22, 42, 63]
C-DAG overlay	genuine	FlexCast [9] (our approach)

Table 3.1. Different classes of atomic multicast protocols.

initially ordered at the lowest common ancestor group of groups in m.dst, and then propagated by the lca to the remaining destinations of m. Differently from other hierarchical protocols, in FlexCast the lca of a message can propagate the message directly to all remaining destinations of the message, respecting minimality. Moreover, in FlexCast, a group with multiple incoming edges must determine the correct order to deliver messages coming from its ancestors, which requires sophisticated mechanisms. In the next section, we present such mechanisms in more details.

3.3 FlexCast: genuine overlay-based atomic multicast

As discussed in the previous sections, a genuine atomic multicast protocol ensures that only the message sender and destinations communicate to order a multicast message [51]. In geographically distributed settings, a genuine atomic multicast protocol can better exploit locality than a non-genuine protocol since messages addressed to nearby groups do not introduce communication with remote groups. Moreover, because a group only receives messages that are addressed to the group, in a genuine atomic multicast protocol groups do not incur communication overhead from relaying messages to the destinations. This is important in geographically distributed environments where communication across wide-area links represents an important cost (e.g., Amazon Web Services).

Most atomic multicast protocols assume that processes can communicate directly with one another. Alternatively, processes communicate following an *overlay*, which determines which processes can exchange messages with which other processes. Imposing limits on communication has advantages. For example, overlays can represent the structure of administrative domains, simplify the design of protocols, and reduce the amount of information each process must keep about the rest of the system (e.g., key management in Byzantine fault tolerant protocols [22]).

However, combining genuineness and overlays is challenging. Existing atomic multicast protocols focus on one aspect or the other but not both. For example, all existing genuine atomic multicast protocols assume a fully connected overlay. Hierarchical protocols, which structure communication between groups as a tree, are not genuine. For example, in ByzCast [22], a multicast message is first sent to the lowest common ancestor of the message destinations, and then proceeds down the tree until it reaches all destinations. ByzCast's logic is simple and processes in a group only need to keep information about their parent and children. However, it is not genuine since a message addressed to the children of group g, but not to g, are first sent to g and then propagated to g's children, violating genuineness.

Figure 3.2 quantifies ByzCast's communication overhead, computed as one minus the ratio between the number of messages that a group delivers (i.e., messages addressed to the group) and the number of messages the group receives as part of communication imposed by the tree overlay, and expressed as a percentage. Data for this experiment was collected while executing the gTPC-C benchmark with tree T_1 and 90% of locality (more details in Section 3.3.5). On average, groups incur on almost 10% of communication overhead. Some groups, however, are more penalized than others, depending on their position in the tree. In particular, about 23% and 36% of the communication of groups 5 and 9, respectively, is overhead. This is in contrast to genuine atomic multicast protocols, which have no communication overhead.

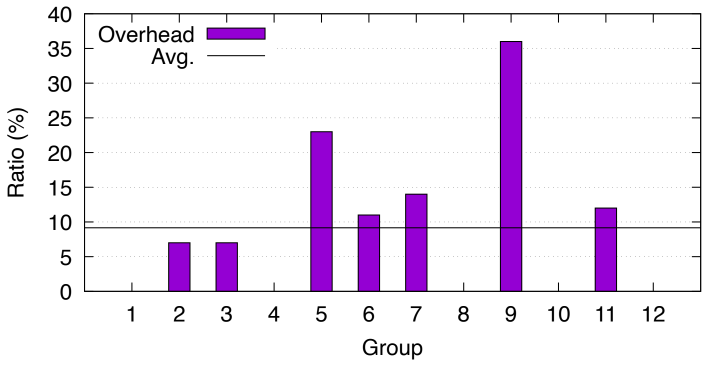


Figure 3.2. Communication overhead in a hierarchical protocol, expressed as a percentage, computed for each group as 1 minus the ratio between number of messages delivered and number of messages received by the group.

Therefore, we propose FlexCast, the first genuine overlay-based atomic multicast protocol. FlexCast assumes a complete directed acyclic graph (C-DAG) overlay. Multicast messages are sent to the lowest common ancestor (`lca`) of the message destinations. The `lca` then propagates the message to all other destinations in one communication step, without involving any groups that are not a message's destination. FlexCast uses a sophisticated history-based protocol to order messages. First, each process builds a history with all messages the process has delivered. This history is propagated to other processes in the C-DAG, so that processes can ensure consistency (e.g., no two processes order two messages differently).

However, simply following other processes' histories is not enough to ensure consistent order due to indirect dependencies. Indirect dependencies happen for a few reasons. For example, if process x orders message m_1 before message m_2 and process y orders m_2 before message m_3, then process z must order m_1 before m_3 as a consequence of dependencies created by processes x and y involving m_2, a message not addressed to z. FlexCast is well-suited to equip geographically replicated systems as it exploits locality.

In the remainder of this section, we describe the core idea behind FlexCast, provide detailed algorithms, a correctness proof, practical implementation considerations, and a comprehensive evaluation of our protocol in comparison with state-of-the-art approaches.

3.3.1 General idea

Groups in FlexCast are structured as a complete directed acyclic graph (C-DAG), as the example in Figure 3.1 (c). We assume there is a total order among groups. Each group is assigned a unique rank in $0..(n-1)$, where n is the number of groups. The C-DAG topology is such that there is a directed edge from each group with rank i to each group with rank j if $i < j$. In this graph, i's *ancestors* have lower rank than i and i's *descendants* have higher rank than i.[3] Figure 3.1 (c) shows a C-DAG with nodes ordered from lowest to highest as: A, B, D, E, C.

A *client process* is an external entity that interacts with the system by submitting requests to be multicast to one or more groups. Clients do not belong to any group in the C-DAG; instead, they use the group structure to disseminate their messages consistently. A client atomically multicasts a message m by sending m to m's lowest common ancestor (`lca`). The `lca` of a multicast message is the group with the lowest rank among the destinations of the message. At its `lca`, m is directly delivered and propagated to m's other destination groups (by definition the `lca` has direct edges with each other destination group in m.`dst`). Similarly to a tree-base atomic multicast, in a C-DAG, a group must respect the dependencies created by its ancestors and propagate dependencies to its descendants. In a C-DAG, however, a group may have multiple ancestors and dependencies can be created by any of them. An important challenge is to ensure that dependencies are properly communicated down the C-DAG without violating the minimality property of genuine atomic multicast. FlexCast uses three strategies to accomplish this, as explained next.

[3]We use the terms "lower" and "higher" groups to denote relative positions of groups in this rank, and "lowest" and "highest" group of a subset of groups, also referring to this rank. "Ancestors" of a group g denote the set of groups lower than g, while "descendants" respectively higher.

Strategy (a): First, every group keeps track of a *history*, a graph where messages are vertexes and their relative order are edges. A vertex contains a message's id and destinations. Messages delivered at a group are recorded in its history and build a total order within the graph. When a group propagates a message to another one, its history is included. The destination group extends its history with the histories that it receives from other groups and messages it delivers. The history then becomes a graph. More specifically, since ordering is respected (discussed next), the history is a DAG. Destination groups use the history to ensure that messages are delivered consistently across the system.

To understand the need for exchanging histories, consider the scenario depicted in Figure 3.3 (a), where group A is the `lca` of messages m_1 (multicast to A and C) and m_2 (multicast to A and B), and group B is the `lca` of m_3 (multicast to B and C). Since A delivers m_1 before m_2 (i.e., $m_1 \prec m_2$) and B delivers m_2 before m_3 (i.e., $m_2 \prec m_3$), C must deliver m_1 before m_3 to avoid a cycle among delivered messages. But C receives m_3 from B before it receives m_1 from A. By receiving B's history, C knows that it should deliver m_1 and then m_3 to avoid cycles.

Unfortunately, including histories in forwarded messages is not enough to ensure consistent order. Intuitively, this happens because not all dependencies are captured in the communication of application messages between groups. There are two cases to consider, depending on whether the group that creates the dependency is aware that it must propagate the dependency to its descendants or not.

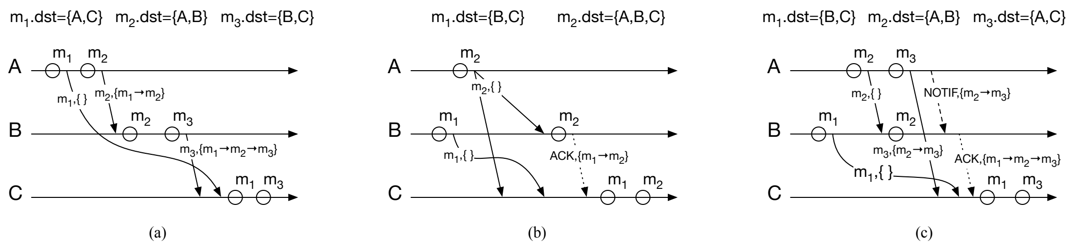


Figure 3.3. Executions of FlexCast illustrating the use of (a) histories, (b) ack messages, and (c) notif messages in an overlay where $A \rightarrow B, A \rightarrow C$ and $B \rightarrow C$. (Legend: a full arrow is the propagation of an application message, a circle is the delivery of a message, a dotted arrow is an ack message, and a dashed arrow is a notif message).

Strategy (b): To motivate the case where a group is aware that it should send dependencies to its descendants, consider the execution in Figure 3.3 (b). In this case, B delivers m_1 before m_2, and C receives m_2 from A (with an empty history) and then m_1 from B (with an empty history since B did not know about m_2 when it sent m_1 to C). Yet, C must deliver m_1 before m_2. FlexCast ensures proper order in such cases as follows. If group g and its descendant h are in the destination of a message m and

g is not m's `lca`, then g sends an ACK message to h with g's history. Conversely, if h receives a message m and h has an ancestor that is in m's destination, but is not m's `lca`, h waits for g's ACK message.

Strategy (c): To motivate the case where a group is not aware that it should send dependencies to its descendants, consider the execution in Figure 3.3 (c). In this case, group A sends m_3 and its history (i.e., m_2 precedes m_3) to C, and B sends m_1 and an empty history to C (i.e., because the dependency between m_1 and m_2 happens in B after B communicates with C). B does not send C the information that m_1 precedes m_2 since m_2 is not addressed to C. Yet, C must deliver m_1 before m_3. To handle this case, when a group determines that a descendant d must forward its history down the C-DAG, it sends a NOTIF message to d so that d can communicate its dependencies to other groups.

More precisely, when a group g, the `lca` of a message (or another destination in m.`dst`) is about to forward message m (respectively, an ACK message regarding m) and there is a group h such that: (i) h is not in m.`dst`; (ii) h is a descendant of g and an ancestor of group r in m.`dst`; and (iii) there is a message in g's history addressed to h, then g sends a NOTIF message regarding m to h. If group h receives a NOTIF message regarding m, it sends ACK messages to all its descendants $k \in m$.`dst`. Moreover, inductively, if there is a message addressed to group h' in h's history with the same restrictions above, h notifies h'. This induction naturally finishes since there is a total order on groups.

Why it is genuine To argue that FlexCast is genuine, first notice the following aspects discussed about *Strategies (a)* and *(b)*:

- when m is multicast, it enters the overlay at m.`lca()` (see Algorithm 1), which is by definition a destination of m;
- m.`lca()` propagates m to its further destinations in m.`dst`; and
- each destination d (other than m.`lca()`) sends ACK messages to groups in m.`dst` higher than d.

From the above, it follows that the communication described involves exclusively groups in m.`dst`.

Now, consider the *Strategy (c)* and notice that:

- a group $g \in m$.`dst` can send a NOTIF message to a group $h \notin m$.`dst` provided that g previously sent a message to h, i.e. some message was multicast to h in run R; and

- inductively, h notifies h' only if some message was multicast from h to h' in run R.

From the above, it follows that groups not in $m.\texttt{dst}$ exchange messages only if they communicated in run R, keeping minimality (see definition in Section 3.2.1).

3.3.2 Detailed protocol

Algorithm 1 presents the basic data structures used in FlexCast. Each group knows the C-DAG topology and has a communication channel to each descendant group (i.e., a FIFO reliable point-to-point link). As a consequence, each process has an input queue for each input channel from ancestor groups. Each queue contains not-yet-delivered messages sent by the respective ancestors.

Algorithm 1 Types and data structures, for each group g

```
 1: Type $Message$: every message $m$ has:
 2:     $m.id$                                          {$m$'s global unique id}
 3:     $m.dst$                                         {$m$'s destinations, a subset of groups}
 4:     $m.payload$                                     {provided by the application}
 5:     $m.acks \leftarrow \emptyset$                   {a set of received acks}
 6:     $m.notifList \leftarrow \emptyset$              {a set of notified groups}
 7:     $m.lca() : func$                                {returns the lca in $m.dst$}
 8: Type (history) $H$:                                 {a history is }
 9:     $H = (M, D, lastDlvd)$                          {messages, dependencies, last one}
10:     $M$ : set of $Message$                          {a pair $(m_1, m_2) \in D$ means ...}
11:     $D : M \times M$                                {$m_1$ ordered before $m_2$: $m_2$ depends of $m_1$}
12:     $lastDlvd : M \cup \{\perp\}$                   {the last message delivered}
13: Group $g$ variables:
14:     $queues \leftarrow [\emptyset, \ldots, \emptyset]$      {an empty queue per ancestor}
15:     $hst \leftarrow H(\emptyset, \emptyset, \perp)$         {the initial history of group $g$}
16:     $deliveredInG \subseteq hst.M$                  {the messages in hst delivered in $g$}
17:     $pendNotif \leftarrow \emptyset$                {a set of pending notifications}
18:     $\forall\ h$ higher than $g, hst(h) \leftarrow H(\emptyset, \emptyset, \perp)$   {the history of $g$ informed to each $h$ so far}
```

A message has a unique id, a set of destination groups, and an arbitrary payload, provided by the application. The protocol stores pending messages along with a set of respective ACK messages and a set of notified groups, both detailed later. Function $m.\texttt{lca}()$ returns the lowest group in $m.\texttt{dst}$.

A group g has the history it learns from each of its ancestors and the messages it delivers. The set of messages delivered in g is a subset of messages in the history. The history builds a DAG with dependencies in $hst.D$. As notification messages may not be immediately delivered according to criteria to be detailed later, a group also has a set of pending notification messages.

When group g communicates with a descendent group h, g informs only the difference in g's history with respect to the last message g sent to h. Therefore, for each descendent h, g keeps track of what part of its history it has already sent to h.

To atomic multicast message m, a client sends m to $m.\texttt{lca}()$. Algorithm 2 presents the events triggered at a group when receiving each one of the three types of messages in our protocol: (i) MSG is a client message; (ii) ACK is an acknowledge message; and (iii) NOTIF is a notification message.

Algorithm 2 Events, for each group g

```
1:  upon receiving [MSG, $m$, $history$] $\wedge$ $g = m.lca()$
2:      a-deliver(m)                                        {the lca can immediately deliver $m$}
3:  upon receiving [MSG, $m$, $history$] $\wedge$ $g \neq m.lca()$
4:      update-hst(history)                                 {update local history with the received history}
5:      queues[m.lca()].enqueue(m)                          {enqueue $m$ in the lca's queue}
6:      reprocess-queues()                                  {reprocess all ancestor queues}
7:  upon receiving [ACK, $m$, $history$] from ancestor $a$  {when receiving an ACK message}
8:      update-hst(history)                                 {update local history}
9:      queues[m.lca()].get(m.id).acks.add([ACK from $a$])  {add $a$'s ACK to $m$}
10:     queues[m.lca()].get(m.id).notifList.merge(m.notifList)   {update $m$'s notification list}
11:     reprocess-queues()                                  {reprocess all ancestor queues}
12: upon receiving [NOTIF, $m$, $history$]                  {when receiving a NOTIF message}
13:     update-hst(history)                                 {update local history}
14:     deps $\leftarrow$ open-dependencies()               {calculate possible dependencies}
15:     if $deps \neq \emptyset$ then
16:         pendNotif.add([NOTIF, $m$, $deps$])             {if dependencies found, add NOTIF to pending set}
17:     else
18:         send-descendants($m$, ACK)                      {otherwise, send ACK to all descendants in $m.dst$}
```

In FlexCast, the lca delivers a multicast message as soon as it receives the message. Thus, the lca imposes its delivery order on all its descendant groups through information disseminated in histories and auxiliary messages. In Algorithm 2, upon receiving a MSG m, if g is the lca, it can deliver m immediately. When non-lca groups receive a MSG first they update their local history with the history received together with m, enqueue m in the corresponding ancestor's queue, and reprocess all ancestors' queues, since this message may carry the information needed to deliver other messages.

When receiving an ACK message, g updates its local history, and associates the ACK to the corresponding MSG m in the lca's queue that originated the ACK. Since an ACK may identify further groups to be notified, the message's list of notified groups is updated accordingly. Group g then reprocesses all queues.

When receiving a NOTIF message, g updates its local history, sends the necessary ACK messages, and possibly sends notification messages to its descendants as well, as detailed later. However, if the local history contains a message m' addressed to g that was not delivered yet, then g waits until it delivers m' before sending the ACK messages, and appends the NOTIF in the pending notifications set, avoiding propagating incomplete dependencies.

Algorithm 3 Main functions, for each group g

```
 1: a-deliver ($m$ : Message)                         {when a message $m$ can be delivered}
 2:   hst-add(m)                                      {add $m$ to local history}
 3:   if $g = m.lca()$ then                           {if $g$ is the lca of $m$}
 4:     send-descendants($m$, MSG)                    {send $m$ to all descendants in $m.dst$}
 5:   else
 6:     queues[m.lca()].dequeue()                     {remove $m$ from the lca's queue}
 7:     send-descendants($m$, ACK)                    {possibly send ACK to all descendants in $m.dst$}
 8:   if $\exists$[NOTIF, $n$, deps] $\in$ pendNotif | $m \in$ deps then   {check if it unblocks any pending NOTIF}
 9:     deps $\leftarrow$ deps $\setminus m$          {remove $m$ from dependencies}
10:     if deps $= \varnothing$ then                  {if no more dependencies, send ACK}
11:       pendNotif $\leftarrow$ pendNotif $\setminus$[NOTIF, $n$, deps]   {remove NOTIF from pending set}
12:       send-descendants($n$, ACK)                  {send ACK to corresponding descendants}
13: update-hst ($ah : H$)                             {ancestor's history $ah$}
14:   hst.M $\leftarrow$ hst.M $\cup$ $ah.M$          {messages and dependencies are}
15:   hst.D $\leftarrow$ hst.M $\cup$ $ah.D$          {integrated to the group's $hst$}
16: reprocess-queues ()
17:   do:                                             {while messages were delivered}
18:     delivered $\leftarrow$ false
19:     for all $q \in$ queues do                     {iterates over all queues $q$}
20:       if can-deliver(q.head()) then               {check conditions to deliver the head of $q$}
21:         a-deliver(q.head())                       {call deliver function for the head of $q$}
22:         delivered $\leftarrow$ true
23:   while delivered
24: open-dependencies (): set of Messages             {calculates dependencies}
25:   return {$\forall\ m \in hst.M \mid g \in m.dst \land m \notin$ deliveredInG }   {messages in $hst$ not delivered yet}
26: send-descendants ($m$ : Message, mType $\in$ {MSG, ACK})
27:   send-notifs(m)                                  {send possible NOTIF messages to descendants}
28:   for all descendant $d \in m.dst$ do
29:     send [mType, $m$, diff-hst($d$)] to $d$        {send MSG or ACK to descendants with diff history}
```

In Algorithms 3 and 4, we present the logic used for each group to deliver messages. The total order of delivered messages is built having the new message depend on the last message delivered. We use set *deliveredInG* to identify messages delivered in g, which is a subset of *hst.M* and is used to identify possible open dependencies in the history. An open dependency happens when a message addressed to g is included in

g's history but not yet delivered. Operation *diff-hst* is an optimization: only the new parts of a history are sent to each descendent. Operation *depend* computes m's possible transitive dependency on m' in hst.

Algorithm 4 Auxiliary functions, for each group g

```
1: hst-add ($m$ : Message)
2:    $hst.M \leftarrow hst.M \cup \{m\}$                              {add m, if not yet in hst}
3:    $hst.D \leftarrow hst.D \cup \{(hst.lastDlvd, m)\}$                {build total order in}
4:    $hst.lastDlvd \leftarrow m$                                     {msgs delivered at this group}
5:    $deliverdInG \leftarrow deliverdInG \cup \{m\}$
6: diff-hst($h$ : a higher group) : $H$                          {g's history not informed to h so far}
7:    let $hstTmp.M \leftarrow hst.M \setminus hst(h).M$
8:    let $hstTmp.D \leftarrow hst.D \setminus hst(h).D$
9:    let $hstTmp.lastDlvd \leftarrow hst.lastDlvd$
10:   $hst(h) \leftarrow hst$                       {history sent to h is updated to current history of g}
11:   return hstTmp
12: depend ($m, m'$ : Message): boolean                         {computes transitive dependencies}
13:   return $(m', m) \in hst.D \lor \exists m'' \mid (m', m'') \in hst.D \land$ depend$(m, m'')$
14: send-notifs ($m$ : Message)                                       {send NOTIF to groups}
15:   for all descendant $d \mid d \notin m.dst$ do
16:      if $\exists d' \in m.dst \mid d$ is ancestor of $d'$ and hst.containsMsgTo(d) then
17:         send [NOTIF, $m$, diff-hst(d)] to $d$
18:         m.notifList.append(d)                              {m carries the notified groups}
19: can-deliver ($m$ : Message)
20:   if ancestors-to-ack(m) $\not\subseteq$ ancestors-that-acked(m) then      {check if all ancestors acked}
21:      return false
22:   if $\exists\ m' \in hst.M \mid g \in m'.dst \land m' \notin deliveredInG \land$
                     depend(m,m') then                              {check for dependencies}
23:      return false
24:   return true
25: ancestors-to-ack ($m$ : Message): set of Groups
26:   return (ancestors of $g$ in $m.dst \setminus m.lca()$) $\cup$
            queues[m.lca()].get(m.id).notifList               {ancestors in m.dst and notified groups}
27: ancestors-that-acked ($m$ : Message): set of Groups
28:   return queues[m.lca()].get(m.id).acks
```

When a message can be delivered, the group adds the message to its local history. An `lca` group sends the message to its descendants, while non-`lca` groups remove the message from the ancestor's queue and send the corresponding ACK messages to their descendants. All groups verify whether delivering this message may unblock pending notifications. Function *send-descendants()* is part of *Strategies (a)* and *(b)* discussed in Section 3.3.1. To send MSG m (or ACK m), the `lca` (or a descendant), first sends possible notification messages to its descendants that are not in m.dst. Function *send-notifs()* implements *Strategy (c)*: it searches past messages and evaluates if notifications are needed, including the notified groups in m's notification list. Then, m is sent to all other destinations in m.dst, carrying the list of notified groups along with the history with information needed by each destination (*diff-hst*).

Function *reprocess-queues()* is called upon receiving MSG and ACK messages (see Algorithm 2). In both cases, it iterates through ancestor's queues and tries to deliver messages. It keeps iterating while messages can be delivered due to updated dependency information. The delivery of messages in non-lca groups is defined in function *can-deliver(m)*. The first condition checks whether g received ACK from all needed ancestors: (i) all ancestors (except the lca) in m.dst; (ii) all ancestors (not in m.dst) notified about message m, which were informed to g either through MSG or ACK. Recall that a notified group, besides sending ACK can further notify other groups. In Algorithm 2, *notifList* accumulates all notified ancestors that have to ACK m. The list of ancestors that have acked is kept in *ancestors-that-acked*. Having the complete information on m, the second condition ensures that any message m' that precedes m and is addressed to g has already been delivered before m's delivery.

3.3.3 Correctness proof

To ensure that FlexCast operates correctly under all conditions defined by our system model, we provide a detailed correctness proof. We first outline the key assumptions underlying the protocol, followed by formal arguments demonstrating how it satisfies the atomic multicast properties.

FlexCast assumes that:

1. processes are organized in disjoint groups and each group is fault-tolerant;
2. groups are totally ordered and the communication topology has FIFO channels from each group to all higher groups.
3. when clients send a multicast message m to destination groups in m.dst, m is sent to the lowest group in m.dst, called the lowest common ancestor (lca) group. We use m.lca() to denote the lowest group in m.dst.

In the following discussion, communication is considered at the level of groups rather than individual processes. Accordingly, when we refer to a group as sending, receiving, or delivering a message, this denotes that a majority of its processes have executed the corresponding action.

Definition 1 *Message Order: for any pair of messages $m \neq m'$, we say that $m < m'$ iff:*

- *m and m' are delivered at a common group, and m is delivered before m';*
- *or by transitivity: $m < m'' \land m'' < m' \implies m < m'$.*

Lemma 1 *For any message m atomically multicast to multiple groups, m is received at all and only destination groups $d \in m.dst$.*

PROOF: By Assumption 3, the client sends m to $m.\texttt{lca}()$. By Assumption 2, any subset $m.\texttt{dst}$ of destinations is directly reached by $m.\texttt{lca}()$. According to Algorithm 3, lines 3-4, when $m.\texttt{lca}()$ receives m, it inconditionally $send\text{-}descendants(m)$ to all other destinations in $m.\texttt{dst}$ and only those. As groups and channels are fault-tolerant, eventually every destination group receives m, and no other group receives it. □

Lemma 2 *Let m and m' be messages such that $m.dst \cap m'.dst \neq \emptyset$. There is a unique group that assigns a relative order to m and m', to be followed by all higher groups.*

PROOF: By Assumption 3, m and m' ingress the overlay through their respective `lca`s and by Lemma 1 both are received at their respective destinations. Since groups are totally ordered (Assumption 2) and $m.dst \cap m'.dst \neq \emptyset$, in the intersection there is a unique lowest group that handles both m and m'. We call this group the lowest common destination of these messages, $lcd(m, m')$. Since message channels are directed towards higher groups only, the relative order of m and m' is assigned at $lcd(m, m')$ and followed at higher groups. □

Lemma 3 *For any atomically multicast message m, the complete dependency information to deliver m is eventually received at each group in $m.\texttt{dst}$. The complete dependency information to deliver m at a group g is the information about any message m' delivered before m, i.e. $m' < m$ at each group lower than g.*

PROOF: By the Algorithms 2, 3, and 4:

1. each group g keeps a history recording the order of messages it delivered and, for each message m delivered, the previous messages m' delivered at groups lower than g, such that $m' < m$;
2. every message carries the history of the sending group, which enriches the history of each receiving group upon reception;
3. each group g in $m.\texttt{dst} \setminus m.\texttt{lca}()$ sends ACKS to higher groups in $m.\texttt{dst}$; and

4. whenever any group g in m.dst has previously sent messages to a group h lower than others in m.dst, g sends NOTIF to h. Each notified group h reacts sending ACKS to higher groups in m.dst and inductively behaves as g to notify further groups. Since groups have a total order, this induction finishes.

From Lemma 1 and facts above, it follows that each group in m.dst is provided with the history of each lower group that is involved in messages ordered before m. □

Lemma 4 *For any atomically multicast message m, any destination group in $m.dst$ knows when the complete dependency information has been received.*

PROOF: By Lemma 1 each group in m.dst receives m, by Assumption 2 it knows which are the lower groups in m.dst and awaits for their respective ACKS. Each ACK informs also if the sending group has notified other groups, from which further ACKS are awaited (see Lemma 3, facts 3 and 4). Thus, from the messages received, any destination of m is able to detect if it has received ACKS from all groups with messages ordered before m. □

Proposition 1 *(Genuineness) A multicast protocol is genuine if, in a run R, only the message sender and destinations communicate to propagate and order a multicast message.*

PROOF: From Algorithm 2, when m is multicast, there are three kinds of messages possible in the overlay: MSG, ACK and NOTIF. MSG and ACK messages are exchanged exclusively among groups in m.dst, i.e. m's destinations. A NOTIF message can only be sent from a group $g \in m.dst$ to h if there exists a previous message m' in run R and $\{g, h\} \in m'.dst$. It follows thus that only destinations of messages in R communicate to propagate and order their messages. □

Proposition 2 *(Quiescence) A multicast protocol is quiescent if a process that has been but is no longer the sender or destination of requests in run R, eventually stops communicating in R.*

PROOF: FlexCast follows a reactive design: a group only sends a message in reaction to some message being received. Consider that after time t_1, a group g is no longer sender or destination of requests. After time t_1, g may still receive NOTIF messages from lower groups and react by sending ACK or other NOTIF messages to higher groups. We argue that after t_1, g eventually stops receiving NOTIF messages and thus no further reacts to generate ACK and NOTIF messages.

Consider groups l and h, respectively lower and higher than g. Assume that l sends a request r_1 to h after time t_1. In this case, l sends both a data message to

h and a NOTIF message to g. These messages include l's history, and l records the history prefix forwarded to each higher group. Since, by assumption, g is no longer a destination, in l's subsequent request r_2 involving h, l detects that no messages with g appear beyond the history prefix already sent in r_1. This means there is no message involving g to decide order and therefore no NOTIF message to g is needed. Supposing all lower groups periodically multicast requests to higher ones, excluding g, t_2 takes place (i.e. g is quiescent) after the first request from each lower to each higher group has been multicast $(t_2 > t_1)$. If no such messages are sent, as FlexCast is reactive g is quiescent from t_1 $(t_2 = t_1)$. □

Proposition 3 *(Validity and Agreement)*

- **Validity**: If a correct process p multicasts a message m, then eventually all correct server processes $q \in g$, where $g \in m.\texttt{dst}$, deliver m.
- **Agreement**: If a process p delivers a message m, then eventually all correct server processes $q \in g$, where $g \in m.\texttt{dst}$, deliver m.

PROOF: Due to Assumption 1, Lemmas 1, 3 and 4, and by Algorithms 3 and 4, we have that all groups in $m.\texttt{dst}$ eventually have m and are able to pass the evaluation of the first condition of function *can-deliver(m)*. It remains to check if there is any message m' that should be deliverd before m. If no m' exists, then the group can deliver m. If there exists such m' it has to be first delivered. Assuming acyclic order, which is further discussed, the arguments above and by induction on message dependencies, there will allways be a message with no pending dependencies to deliver that will then enable further ones to be delivered, such that m can be delivered. Therefore, validity holds. By the same arguments, agreement holds. □

Proposition 4 *(Integrity)*

- **Integrity**: For any process p and any message m, p delivers m at most once, and only if $p \in g$, $g \in m.\texttt{dst}$, and m was previously multicast.

PROOF: By Lemma 1 a multicast message m reaches all and only its destination groups. Any other possible message (ACKS or NOTIFS) do not convey messages to be delivered. So, a group g delivers m only if $g \in m.dst$ and m has been multicast first. □

Proposition 5 *(Prefix Order)*

- **Prefix order**: For any two messages m and m' and any two server processes p and q such that $p \in g$, $q \in h$ and $\{g, h\} \subseteq m.\texttt{dst} \cap m'.\texttt{dst}$, if p delivers m and q delivers m', then either p delivers m' before m or q delivers m before m'.

Proof: From Lemma 2 there is a unique group, $lcd(m, m')$, that assigns the relative order among m and m'. From Lemmas 3 and 4 any further group in $h \in m.dst \cap m'.dst$ receives and preserves the order assigned by $lcd(m, m')$. Thus prefix order holds. □

Proposition 6 *(Acyclic Order)*

We already defined the relation $\prec$ on the set of messages server processes deliver: $m \prec m'$ iff there exists a process that delivers m before m'.

- **Acyclic order**: The relation $\prec$ is acyclic.

We argue that FlexCast ensures acyclic order by contradiction. Assume cycle C exists: $m_1 < m_2 < ... < m_k < m_1$. Let C be such that $m_k < m_1$ happens at group h (i.e., h delivers m_k and then m_1), where h is the highest group in the overlay. This is possible because the overlay induces a total order on groups.

Let q be the lcd group that delivers messages m_1 and m_2. We consider all `lca` combinatios for m_1 and m_2 (in Figure 3.4, cases a, b, c and d). We claim that there is a causal path P from the delivery of m_2 at q to the reception of message m_k at process p. Since processes deliver messages following their causal dependencies, showing that causal path P exists means that before p delivers m_k, it knows that m_1 precedes m_k, which leads to a contradiction since p will not deliver m_k before delivering m_1.

Proof: The proof of the claim is by induction on the size k of cycle C.

Base step ($k = 2$): This case corresponds to the four patterns involving messages m_1 and m_2 (see Figure 3.4), having $r = p$. For patterns (a) and (b), the claim follows directly. For patterns (c) and (d): Since m_2 is addressed to q and p, and p is below q in the overlay, upon delivering m_2, according to Algorithm 3, q sends an ack message to p (with all q's dependencies) and thus there is a causal path.

Inductive step: Assume there is a causal path between $m_2 < m_3 < ... < m_k$. We show that there is a causal message path from m_1 to m_k, where q delivers messages m_1 and m_2, and r is one of the destinations of m_2 (together with q and possibly other processes).

There are five possibilities for how q creates a dependency between m_1 and m_2, and where r is placed with respect to q in the communication overlay (see Figure 3.4).

- Cases (a) and (b). In these cases, r is necessarily below q in the overlay, since q multicasts m_2 and otherwise r would not be a destination of m_2. In these cases, m_2 multicast by q to r creates a causal path from m_1 to m_2 at r. From the induction hypothesis, this leads to a causal path until m_k.

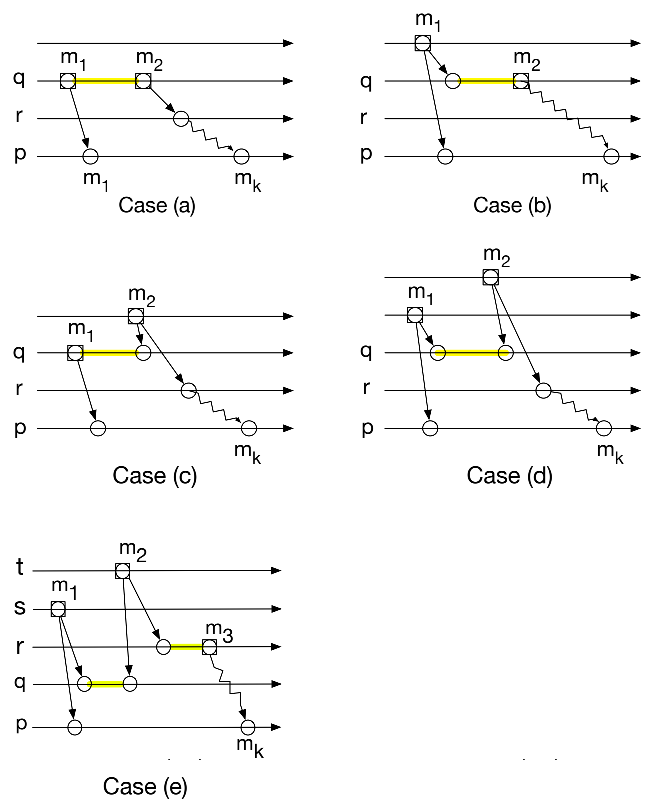


Figure 3.4. Five ways a process q can create a dependency between messages m_1 and m_2. Squares denote send events, circles denote delivery events, and arrows denote message destinations. Yellow highlights show dependencies between m_1 and m_2, while zig-zag arrows indicate causal paths.

- Cases (c) and (d). In these cases, we consider that r is below q in the overlay. Since both q and r are destinations of m_2 and r is below q, from Algorithms 3 and 4, q sends an ACK message to r and r waits for the ACK message before delivering m_2. This creates a causal path between the delivery of m_1 and m_2 at q and the delivery of m_2 at r. From the induction hypothesis, it follows that there is a causal path all the way to the delivery of m_k at p.

- Case (e). r is positioned above q in the communication overlay. Since there is a causal path P between the delivery of m_2 at r and the receive of m_k at p, it is the case that r sent a message in P, say m_3. Regarding the generation of m_3, it could also be that $r = t$. Regarding the generation of m_1, it could be that $s = q$.

  Since r knows that it was involved in m_2 with q, below r in the overlay, r sends a NOTIF message to q, and as a response, q sends an ACK message in path P to groups in $m_3.dst$ below q (completing the information that m_1 is in the past of m_3). Since groups can only deliver m_3 once these ACKS arrived, further messages after m_3 build a path P to m_k in p starting from m_2 in r. From the induction hypothesis that there is a path from m_1 to m_k. □

3.3.4 Practical considerations

Garbage collection The protocol as described so far does not include garbage collection. In our FlexCast prototype, however, we prune local histories associated with each ancestor group. A distinguished process periodically multicasts a *flush* message to all groups. Once a group delivers this message, it knows that all messages that precede *flush* can be garbage collected. The intuition behind this mechanism is that to deliver a message m from a specific ancestor, all dependencies before m must be resolved and do not need to be re-evaluated in the future. To further reduce communication, histories sent with messages do not enclose the ever-growing system history. FlexCast sends only a *diff* of the history for each descendant group. The idea is implemented by keeping track of the last message of the local history sent to each descendant d and, in subsequent messages to d, sending a history that contains only the newest messages added since the last communication to d.

Tolerating failures FlexCast assumes the same approach used in other atomic multicast protocols to tolerate failures (e.g., [23], [39], [50], [69], [86], [22]), that is, processes within a group are kept consistent using state machine replication [67, 90]. This means that processes in a group can fail as long as enough processes remain operational within the group. Consequently, groups do not fail as a whole and must remain connected (i.e., no network partitions). Tolerating the failure of a group requires additional system assumptions [89].

The implications of this approach on the number of correct processes per group and process communication depend of the particular consensus protocol used to implement state machine replication within a group. For example, Paxos [67] requires a majority of correct processes within each group and can tolerate message losses. In our prototype, we simply rely on TCP connections to ensure reliable communication.

3.3.5 FlexCast evaluation

In this section we present the evaluation of our protocol. We start by explaining the evaluation rationale, then we describe the environment and the benchmarks used, and present the results.

Evaluation rationale We compare FlexCast to a distributed atomic multicast protocol and a classical hierarchical atomic multicast protocol using single-process groups (i.e., no failures are tolerated) in all three protocols. In doing so, our evaluation focuses on the inherent costs of three classes of atomic multicast protocols (see Table 3.1) and avoids overhead introduced by replication. We use Skeen's protocol as distributed atomic multicast because its ordering mechanism is used by several other protocols (e.g., [23], [39], [50], [69], [86]). Moreover, when groups contain a single process, FastCast [23] and Whitebox [50] atomic multicast protocols behave as in Skeen's protocol. Skeen's protocol is genuine, can order messages in two communication steps, which has been shown to be optimum [88], and assumes that any two groups can communicate. We choose ByzCast as a classical hierarchical atomic multicast protocol. ByzCast is non-genuine and imposes a tree overlay on communication, the minimum overlay that ensures a connected system. In single-process groups, ByzCast does not introduce any overhead particular to tolerating malicious behavior. We implemented prototypes of all protocols in Java.

Our experimental evaluation aims to understand the behavior of the considered protocols in geographically distributed deployments subject to realistic workloads. Our workload extends the well-established TPC-C benchmark to accommodate locality, a common property in geo-distributed systems. In these settings, we seek to answer the following questions:

1. What is the impact of different overlays on FlexCast and hierarchical protocols?
2. How quickly can a protocol order messages addressed to two or more groups?
3. What is the communication overhead of overlay-based hierarchical protocols?
4. What is the communication cost of atomic multicast protocols?

Environment and deployment The experimental setup was configured with 12 server machines and 24 client machines, connected via a 1-Gbps switched network, in CloudLab [34]. The machines are equipped with eight 64-bit ARMv8 cores at 2.4 GHz, and 64GB of RAM. The software installed on the machines was Linux Ubuntu 20.04 (64 bits) and 64-bit Java virtual machine version 11.0.3. Machines communicate via TCP.

We consider an emulated wide-area network that models Amazon Web Services (AWS). Each group represents an AWS region and we experimented with a deployment of 12 AWS regions, as depicted in Figure 3.5 (a). The emulated latencies among regions are based on real measurements in AWS [21]. Enough client processes (to saturate our FlexCast implementation) are uniformly distributed along the 24 client machines that represent each region/group, and they send requests to the nearest group. Upon delivering a message, each message destination replies to the message's sender (client).

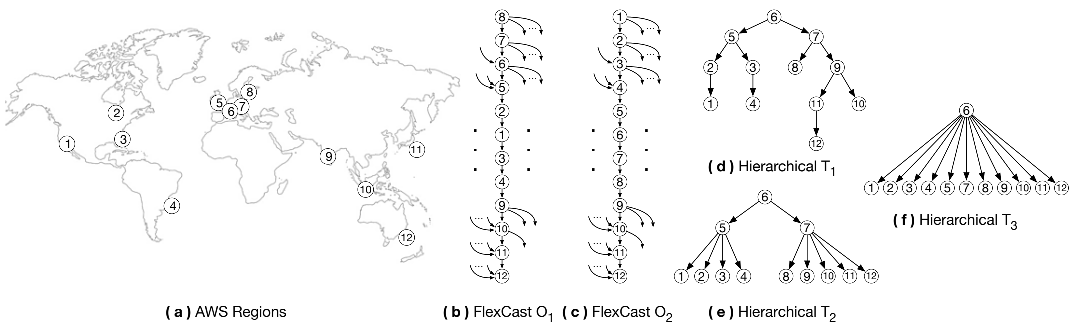


Figure 3.5. AWS regions and different overlays used in our experimental evaluation.

gTPC-C Benchmark We developed gTPC-C, a geographically distributed benchmark inspired by the well-established TPC-C benchmark [29]. We translate TPC-C warehouses into groups, deployed in one or more AWS regions, and TPC-C transactions into messages multicast to their corresponding warehouses.

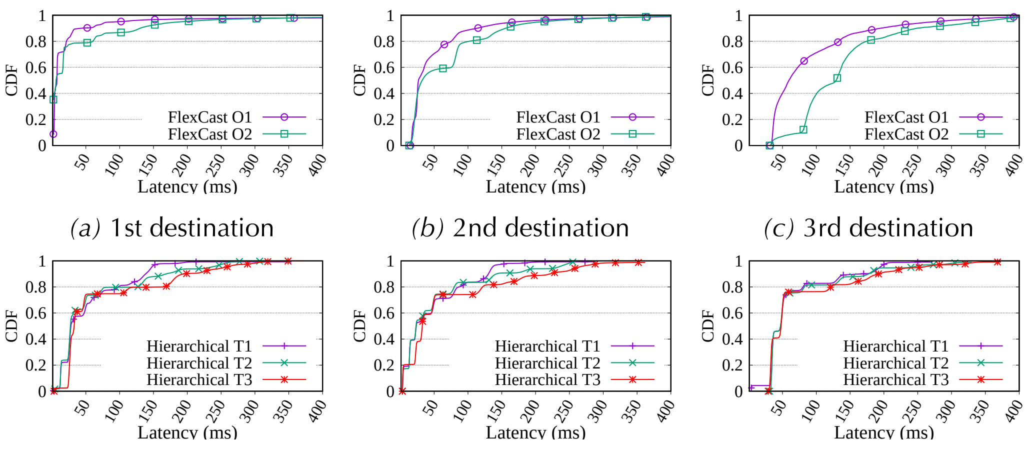
According to the TPC-C benchmark, clients can generate the following transactions (with a given probability): new order (45%), payment (43%), order status (4%), delivery (4%), or stock level (4%). The last three transactions are single-warehouse (local), resulting in a message multicast to the client's home warehouse. Since all multicast protocols perform the same when ordering a message multicast to a single group, in our latency measurements we only consider global transactions, which result in messages addressed to multiple warehouses. Consequently, this workload only contains new order and payment transactions, always involving two or more warehouses. New order transactions can have from 5 to 15 items, where each item has a 2% probability of being issued to a warehouse that is not the client's home warehouse, as defined by TPC-C.

To capture locality, when choosing an additional warehouse to the client's home warehouse, the client picks the nearest warehouse to its home warehouse with a configurable high probability, the *locality* rate; otherwise, the client chooses the next nearest warehouse, and so on, up to the farthest warehouse to the client's home warehouse. Our criteria to define locality is inspired by a common wholesale supplier policy that when an item is not available in the nearest warehouse to a client (i.e., the home warehouse), it is shipped from the closest warehouse that has the item. This locality specification implies that most messages are addressed to only two warehouses (same as in standard TPC-C), and some to three. Very few are addressed to more than three groups, therefore we do not consider these messages in our experiments.

Clients operate in a closed loop issuing one transaction at a time and are deployed in the same region as their home warehouse. Each experiment lasts for a period of approximately one minute, in which clients collect and store latency data. We discard the first and last 10% of the data collected during the experiment to avoid possibly noisy data during warm up and end of execution.

The effect of overlays In the first set of experiments, we investigate the role of overlays on FlexCast and hierarchical protocols. We compare the latency experienced by clients of two FlexCast overlays, and three hierarchical overlays (trees), as depicted in Figure 3.5.

Trees T_1, T_2 and T_3 contain different numbers of inner nodes. In principle, a larger number of inner nodes provides better distribution of communication overhead among these nodes. Trees with many inner nodes, however, may lead to additional communication steps when ordering messages. For overlays O_1 and O_2, we initially selected a starting node (i.e., central node 8 in O_1 and left-most node 1 in O_2). Then, the closest node to the initial one, the closest node to the second chosen node, and so on. Since O_1 and O_2 are complete DAGs, a node is connected to all nodes that succeed it (e.g., the first node is connected to all nodes).



(a) 1st destination *(b)* 2nd destination *(c)* 3rd destination

(d) 1st destination *(e)* 2nd destination *(f)* 3rd destination

Figure 3.6. Latency per destination group when varying overlays in FlexCast and a hierarchical protocol, gTPC-C with 90% locality.

Figure 3.6 and Table 3.2 present the results. We report the latency per group addressed by the message. The latency of the first (respectively, second and third) destination corresponds to the first (respectively, second and third) response the client receives from the groups addressed by the message. O_1 shows better performance than O_2 for all destinations. This happens because O_1 better exploits locality: higher nodes in the DAG have the lowest latencies in the geographical distribution. Hereafter, we evaluate FlexCast using overlay O_1.

		Destination								
		1st			2nd			3rd		
	Ov.	90p	95p	99p	90p	95p	99p	90p	95p	99p
FC	O_1	144.0	279.0	1403.1	398.0	829.0	2243.42	1406.0	2195.0	4542.5
	O_2	156.0	350.0	790.22	416.0	652.0	2006.83	1028.0	1681.5	3112.9
HR	T_1	229.0	267.0	311.0	261.0	288.0	403.0	307.0	386.0	408.0
	T_2	233.0	269.0	311.0	215.0	249.1	351.0	261.0	338.0	375.28
	T_3	311.0	398.0	544.0	381.0	480.0	622.0	397.0	531.6	621.0

Table 3.2. Latency percentiles in milliseconds for each destination group when varying the overlay (Ov.) in FlexCast (FC) and the tree in the hierarchical (HR) protocol, gTPC-C with 90% locality.

Differently than FlexCast, whose performance is largely dependent on the overlay, a hierarchical protocol is not so sensitive to the chosen tree (but see also the discussion in Section 3.3.5), although the trees do have an impact on the performance. T_1 shows slightly better performance in all destinations than T_2 and T_3. This is due to the communication overhead (further discussed in Section 3.3.5) of involving non-destination groups, and also the bottleneck effect of involving the tree root on T_3 for all messages in the system. From these results, we select T_1 to represent a hierarchical protocol in the rest of our evaluation.

Throughput In the second set of experiments, we assess the overall performance of our standard gTPC-C, measured in kilo operations per second (kops), including local and global messages, when deployed in a configuration with 99% locality rate.

We conduct multiple experiments while gradually increasing the number of clients. Figure 3.7 presents the results. Although FlexCast was designed to optimize latency, it can maintain the same throughput as the other protocols up to its saturation point. This effect can be seen by the bend of the throughput curve of FlexCast starting with 960 clients. This behavior is explained by FlexCast's latency-oriented design. As discussed next, FlexCast performs particularly well under high-locality workloads, such as gTPC-C, where most messages are addressed to only two destinations. As the number of clients increases, a larger fraction of messages targets more than two destinations, increasing the number of auxiliary messages required for ordering. This additional coordination delays delivery at lower destinations, ultimately leading to earlier saturation.

In the experiments presented next, we consider configurations with 240 clients. This is justified by the fact that none of the algorithms is subject to queuing effects, which would interfere with their inherent latency.

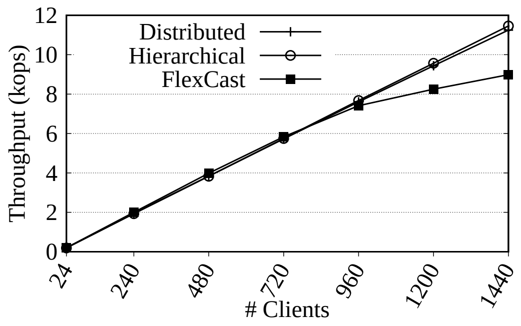
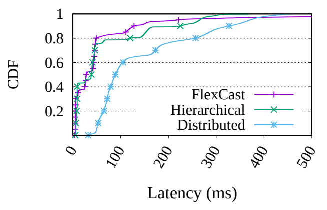


Figure 3.7. Throughput vs. number of clients with 99% locality.

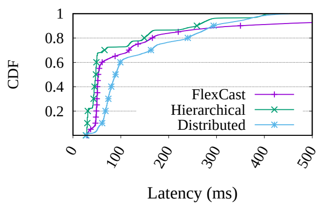
Latency In the third set of experiments, we increase the locality rate and measure the latency experienced by the clients when receiving a response from each of the destinations of a global multicast message.

Figure 3.8 and Table 3.3 present the results. FlexCast outperforms both a distributed and hierarchical protocols in the latency of the first destination group for all three experimented locality rates. We attribute this behavior to the fact that FlexCast benefits from two aspects that reduce the cost of ordering messages in the first destination in a distributed scenario: *(i) Communication steps:* while in a distributed protocol groups addressed by a message need to exchange timestamps before a destination group can deliver a message, in FlexCast the first destination group in the DAG (i.e., the `lca` of the message) can deliver the message as soon as it receives the message from a client; the hierarchical protocol also benefits from this aspect, however, in ByzCast, the `lca` of a message may not be a message destination since it is not a genuine protocol. *(ii) Locality rate:* having a workload with a high locality rate increases the number of messages that FlexCast can deliver using fewer communication steps than both other protocols. This gives FlexCast an advantage since the cost for a communication step may take tens of milliseconds in geographical settings.

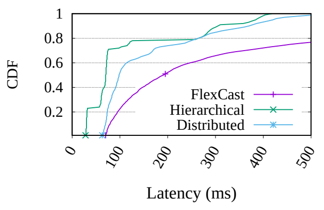
In the second destination, FlexCast performs worse than the hierarchical protocol and outperforms the distributed protocol. As discussed above, hierarchical protocols require only one additional communication step to order a message at the second destination, provided both destinations are directly connected (a condition that typically holds due to locality), while the distributed protocol, in addition to require destination groups to communicate, is also exposed to the convoy effect, which further slows down the delivery of messages [50]. In the third destination, FlexCast latency increases and the simplicity of a hierarchical protocol algorithm pays off. In both the second and third destinations, FlexCast may need extra communication steps to receive the necessary ACK messages to deliver a multicast message m, evaluate possible dependencies, and wait for dependencies to be solved (i.e., waiting for the delivery of previous mes-



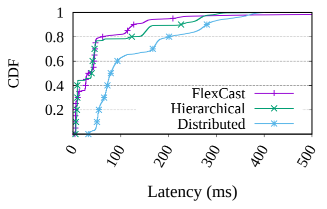
(a) 1st dest., 90% Locality



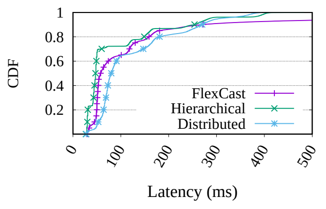
(b) 2nd dest., 90% Locality



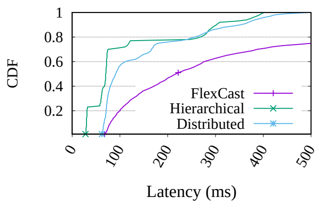
(c) 3rd dest., 90% Locality



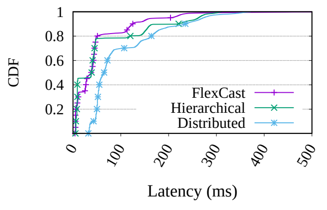
(d) 1st dest., 95% Locality



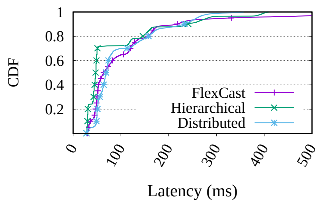
(e) 2nd dest., 95% Locality



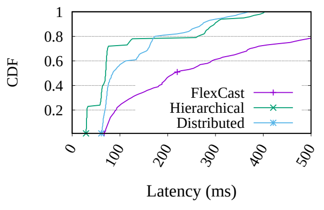
(f) 3rd dest., 95% Locality



(g) 1st dest., 99% Locality



(h) 2nd dest., 99% Locality



(i) 3rd dest., 99% Locality

Figure 3.8. Latency per destination (dest.) group when varying locality rate.

sages ordered before m in ancestor groups). Although FlexCast performs worse than both hierarchical and distributed protocols in the third destination, messages addressed to three (or more) groups are rare in gTPC-C, a characteristic inherited from TPC-C.

As a consequence of FlexCast's C-DAG overlay and the fact that each client in the gTPC-C benchmark is associated with the nearest warehouse, clients send most of their messages to their home warehouse and to the next nearest warehouse. The rate at which this phenomenon happens is regulated by the configured locality. Therefore most messages in the workload have a disjoint destination set. This increases FlexCast's advantage over a distributed protocol when messages are addressed to two groups if the groups are placed consecutively in the C-DAG. The hierarchical protocol also benefits from locality, although as a non-genuine protocol, it introduces communication overhead, quantified in Section 3.3.5. The locality rate also helps to decrease the number of auxiliary messages (i.e., ACK and NOTIF) needed by FlexCast to ensure consistency in the global total order, since interdependencies will be relatively fewer in such a scenario. Table 3.3 shows the latency percentiles (90, 95 and 99) of all destinations when varying the locality rate for all techniques. Although the hierarchical protocol shows

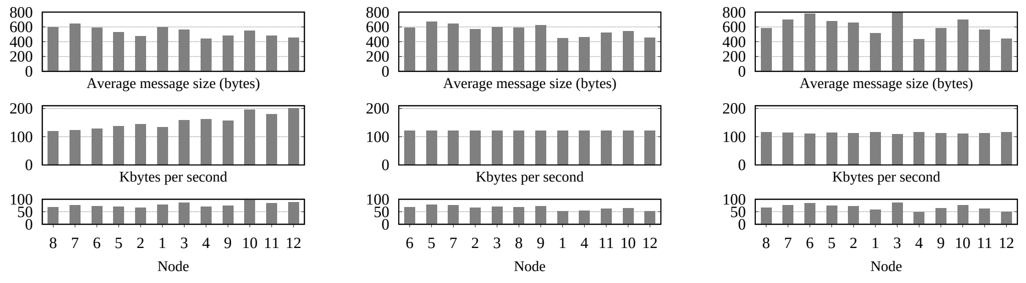
		Destination								
		1st			2nd			3rd		
	Loc.	90p	95p	99p	90p	95p	99p	90p	95p	99p
FC	90%	144.0	279.0	1403.1	398.0	829.0	2243.4	1406.0	2195.0	4542.5
	95%	131.0	217.0	1146.0	288.0	671.4	2192.6	1307.2	2231.6	4211.5
	99%	132.0	218.0	764.0	227.0	458.0	1562.0	1404.9	1975.7	3583.9
HR	90%	229.0	267.0	311.0	261.0	288.0	403.0	307.0	386.0	408.0
	95%	226.0	265.0	307.0	255.0	286.0	403.0	306.0	381.0	405.0
	99%	224.0	264.0	303.0	243.0	284.0	402.0	303.0	376.2	406.8
DT	90%	335.0	377.0	452.0	299.0	367.0	444.0	373.0	423.0	527.7
	95%	284.0	349.0	417.0	275.0	339.0	406.98	365.0	407.0	528.0
	99%	241.0	279.0	370.0	238.0	263.0	355.0	309.5	367.0	415.3

Table 3.3. Latency percentiles in milliseconds for each destination when varying the locality rate (Loc.) for all three protocols: FlexCast (FC), Hierarchical (HR), and Distributed (DT).

on average a better performance when aggregating the latencies of all destinations, FlexCast is more sensitive to locality. In the first destination, FlexCast's reduces 90p latency by 9% when increasing locality from 90% to 99%, while the hierarchical protocol reduces by 3%. Despite its higher latency, the distributed protocol reduces latency by up to 29% when increasing locality from 90% to 99%.

The cost of exchanging histories We also evaluate the amount of information required by each protocol to implement atomic multicast. All protocols propagate the message payload, as defined by gTPC-C, and protocol-specific information, which in the case of FlexCast includes histories. Figure 3.9 displays our findings. Graphs at the top present the number of messages received by each node per second. Graphs in the middle show the average message size per node. Unlike the other protocols with fixed average sizes, FlexCast shows an increase in average message size as nodes ascend the C-DAG topology (see Figure 3.5, C-DAG overlay O_1). This is due to higher nodes requiring more history data from their ancestors. Graphs at the bottom show the overall data exchanged by nodes per second.

In summary, our experiments indicate that FlexCast exhibits distinctive behavior, with higher nodes in FlexCast's C-DAG exchanging a higher amount of data than lower nodes. This results in larger messages compared to the other protocols. On average, a node exchanges 68.5 Kbytes per second in the distributed protocol, 66 Kbytes per second in the hierarchical protocol, and 79 Kbytes per second in FlexCast.

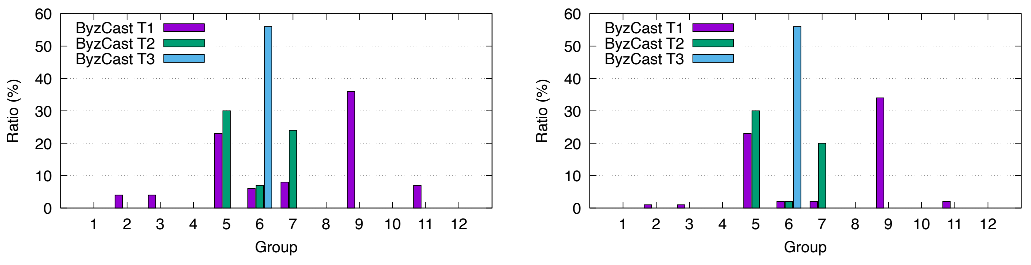


(a) FlexCast *(b)* Hierarchical *(c)* Distributed

Figure 3.9. The amount of information exchanged by each protocol (99% locality, 720 clients).

The overhead of non-genuineness We also evaluate the communication overhead of non-genuine hierarchical protocols (Figures 3.2 and 3.10). Intuitively, communication overhead captures the amount of communication involving a group due to multicast messages not addressed to the group. We express communication overhead as a percentage and define it as 1 minus the ratio between the number of payload messages delivered by a group and the number of payload messages received by the group during an execution of the protocol. We focus on payload messages as these are typically larger than auxiliary messages used in a protocol.

The overhead across groups depends on the tree overlay and the workload. But while all inner groups in a tree are potentially subject to communication overhead, leaf groups have no overhead since they are always in the destinations of messages they receive. Locality also plays a role in communication overhead. A tree can benefit from locality by directly connecting groups that are near each other. This is the motivation behind tree T_1: as locality increases, T_1's overhead decreases, since communication will more likely involve directly connected groups (see Table 3.4).



(a) 95% Locality *(b)* 99% Locality

Figure 3.10. Communication overhead of each group in hierarchical protocols with 95% and 99% of locality.

Tree T_3 has lower communication overhead than T_1, but this comes at the cost of penalizing group 6 (i.e., T_3's root), which has to endure 56% of overhead. In T_1, groups 5 and 9 present high overhead as they are roots (lowest common ancestors) of different subtrees that represent separate geographical regions (America and Asia). The tree root does not have much overhead since locality is high in groups within the Europe region. The same is observed in T_2, where groups 5 and 7 of disjoint subtrees present the highest overheads.

Overlay	Locality	Mean overhead	Max
T_1	90%	9.16% (11.18)	36%
	95%	7.33% (11.12)	36%
	99%	5.41% (11.06)	34%
T_2	90%	5.75% (11.31)	30%
	95%	5.08% (10.50)	30%
	99%	4.33% (9.90)	30%
T_3	90%	4.66% (16.16)	56%
	95%	4.66% (16.16)	56%
	99%	4.66% (16.16)	56%

Table 3.4. Mean overhead, standard deviation, and maximum overhead in hierarchical trees when varying the locality rate.

Tables 3.2 and 3.4 suggest a tradeoff: trees with the lowest latencies are subject to higher overhead on average, while trees with worse performance have lower communication overhead on average.

3.4 Reconfiguring atomic multicast

In the previous section, we presented FlexCast, a state-of-the-art atomic multicast protocol that assumes a complete directed acyclic graph (C-DAG) overlay [9]. The performance of an overlay-based atomic multicast protocol is sensitive to the workload characteristics and locality. The workload determines the requests' destinations and their distribution; the locality of a workload defines the geographical proximity between clients and destinations, which affects communication latency. As demonstrated in Figure 3.6 in the previous section, running an overlay-based protocol using distinct overlays with the same workload and locality can output different performances. To avoid static reconfiguration, with the performance consequences of system shutdown followed by start-up, we propose a dynamic reconfiguration mechanism for changing FlexCast's overlay configuration spontaneously.

We define an optimization model that captures the cost of request ordering and proposes an overlay better suited to the ongoing workload submitted to the system. Since both workload and locality may change over the system execution, the model periodically determines whether an alternative overlay should be in place and, if so, triggers reconfiguration.

We have extended FlexCast with a reconfiguration scheme and evaluated it in a WAN environment, as described later. We show that FlexCast's reconfiguration protocol can substantially reduce latency and enhance throughput by transitioning its overlay from a non-optimal to an optimized configuration in a geographically distributed environment. For instance, some of our results demonstrate that throughput increases by approximately 260% following a reconfiguration, while latency decreases by about 83% for groups representing the bulk of the workload, while remaining stable or experiencing a slight degradation for less frequently addressed groups.

3.4.1 Reconfiguration protocol

FlexCast performs better with workloads that experience some form of locality and, as the workload changes, its performance can be affected. Therefore, it is important to adapt to possible locality changes. Most approaches in the literature adapt to changing workload locality by migrating objects across partitions, aiming to convert cross-partition operations into single-partition ones (e.g., [55]). We consider these approaches orthogonal to FlexCast and focus here on the definition and reconfiguration of the overlay network used, one of the main aspects of FlexCast.

FlexCast uses a C-DAG overlay built on a total order of groups. When destination groups are neighbours in this order, coordination to deliver requests is minimal. Otherwise, delivery may be delayed depending on the number of intermediate groups involved and the latencies experienced by the network links connecting such groups. We assume all processes, including servers, clients, and the reconfiguration coordinator (introduced later), start with the same initial C-DAG overlay, constructed based on a shared total order. This overlay is referred to as the initial configuration. Clients tag their requests with the configuration they know, starting with the initial configuration.

The general rationale for reconfiguration is to minimize the overall delivery latency for the ongoing workload. This means that based on *(i)* the current overlay, *(ii)* the average latency between any pair of groups, and *(iii)* the observed rate of requests to each possible subset of destinations, which characterizes the workload, a new overlay is computed. The new overlay is obtained using the minimization function presented next.

Algorithm 5 takes as input the set of destination groups N, the latencies lat between groups, and a workload representation WL that includes the frequency of re-

Algorithm 5 Overlay reconfiguration

```
1: $Input : N$                                              {the set of destination groups}
2: $Input : Lat(N \times N) \rightarrow \mathbb{R}$               {latencies among pairs of groups}
3: $Input : WL = \{(dest, freq) \mid dest \in 2^N \wedge freq \in \mathbb{N}\}$ {for each possible dest., a frequency of reqs}
4: Optimization function $(N, Lat, WL)$ :
5:    Minimize:                                            {find among all total orders in $N$}
6:       $o \in$ permutations$(N)$                               {the one which has minimum}
7:          $\sum_{\forall l \in WL}$ maxPathLat$(o, l.dest, Lat) \times l.freq$   {sum of longest path per destination
                                                           weighted by its frequency in $WL$}
8: Function permutations($n$ : set of destination groups) :
9:    returns the set of all strict total orders with all elements of $n$
10: Function maxPathLat($o$ : a total order in $N$, $d : 2^N$, $lat : Lat$) : $\mathbb{R}$
11:    returns the latency of the longest path among destinations in $d$
12:          following order $o$ and latencies in $lat$
```

quests for each possible destination. Function *permutations* receives the set of destination groups N and computes all possible strict total orders[4] that can be derived from N. Function *maxPathLat* takes as input one possible total order o, a destination set d, link latencies lat between groups, and returns the longest path (i.e., maximum latency between the `lca` and the highest group within d) according to the order in o and latencies in lat. Finally, the optimization function identifies the total order in N that minimizes the sum of the longest path latencies for each destination, weighted by its frequency in the workload *WL*.

To obtain the input parameter *WL*, each group records the frequency of destination group sets for all received requests. To coordinate the reconfiguration protocol, we implemented a separate process called reconfiguration coordinator (*RC*), which is a special client connected to all server groups. *RC* periodically multicasts a CHECKFREQ query to all groups, which respond with the observed workload, including destination frequency information, building the parameter *WL*. Alongside the inputs N, O, and lat, process *RC* can compute the optimization function described in Algorithm 5. Server groups reset their local frequency information once they reply to a CHECKFREQ, to ensure that changes in locality are detected more quickly, without interference from the frequencies recorded in previous iterations.

Once a new overlay o' is computed, process *RC* triggers the reconfiguration mechanism by multicasting a [RECONFIGURE, o'] request with the new overlay defining a new total order o' for all server groups. FlexCast ensures that RECONFIGURE messages are delivered to all groups in the same order. This guarantees that any request sent

[4]A strict total order is irreflexive.

under the old configuration is processed before the RECONFIGURE message, ensuring that all dependencies within the old configuration are resolved before transitioning to the new configuration. Due to asynchrony, a group can receive a request tagged with a new configuration that it has not yet recognized or reconfigured. In such cases, the process temporarily stores these requests in a buffer. Once the group receives the configuration change notification and reconfigures accordingly, the buffered requests are processed.

When a group is ready to deliver the RECONFIGURE message, it initiates the reconfiguration process. This process involves several key steps: suspending the handling of incoming requests, waiting for the lower groups in the new overlay o' to establish connections, initiating connections with the higher groups in o', processing any requests that were previously buffered for the new configuration, and finally, resuming FlexCast's normal operation, now handling requests according to the updated configuration.

Since our reconfiguration coordinator only notifies the servers about the new configuration, a client might send a request r tagged with an outdated configuration and address it to the old r.`lca`(). When a server receives a request tagged with an old configuration, it informs the client about the updated overlay o', instructing the client to switch to the new configuration and retransmit the request to the appropriate `lca` in o'.

3.4.2 Reconfiguration evaluation

This section presents the evaluation of FlexCast's reconfiguration mechanism. We describe the environment and benchmarks used, present the results, and summarize the lessons learned.

Environment and deployment The experimental setup was configured in a private cluster with 12 server machines and 24 client machines connected via a 1-Gbps switched network. Each machine has two AMD EPYC 7282 16-core processors and 32GB of RAM. The software installed on the machines was Linux Ubuntu 18.04 (64 bits) and 64-bit Java virtual machine version 11.0.3. Machines communicate via TCP.

Our FlexCast evaluation aims to understand the behavior of the considered protocols in geographically distributed deployments subject to realistic workloads, and how our reconfiguration scheme can affect system performance. We consider an emulated wide-area network that models Amazon Web Services (AWS), the same as depicted in Figure 3.5. The first reconfiguration experiments ran in a deployment with 3 server groups, distributed alongside 3 AWS regions (Canada, N. Virginia, and São Paulo), as depicted in Figure 3.11. We also executed experiments with 6 server groups (Figure 3.15), running in a deployment with 6 AWS regions spanning 3 continents (America, Europe and Asia).

To evaluate the system, we again employed our gTPC-C benchmark, a geographically distributed locality-aware benchmark, inspired by the well-established TPC-C benchmark [29], in which TPC-C warehouses are translated into groups, deployed in one or more AWS regions, and TPC-C transactions into requests multicast to their corresponding warehouses (see Section 3.3.5 for more details). Clients operate in a closed loop issuing one transaction at a time and are deployed in the same region as their home warehouse. Experiments last for a period of approximately one to two minutes, depending on the scenario, in which clients collect and store performance data. We discard the first and last 10% of the data collected during the experiment to avoid possibly noisy data during warm up and end of execution. We conducted experiments exposing the system to workload locality changes. We analyzed how our reconfiguration protocol responds to these changes.

The impact of reconfiguration In these experiments, our RC process operates periodically (approximately every 10-seconds) calculating possible optimized overlays for the current workload. The workload is determined as follows: $\{[0,1] = 32.6\%; [0,2] = 59.2\%; [1,2] = 4.6\%; [0,1,2] = 3.5\%;\}$, where $[x,...,y] = z\%$ means that requests are multicast to destination groups $x,...,y$ with probability z.

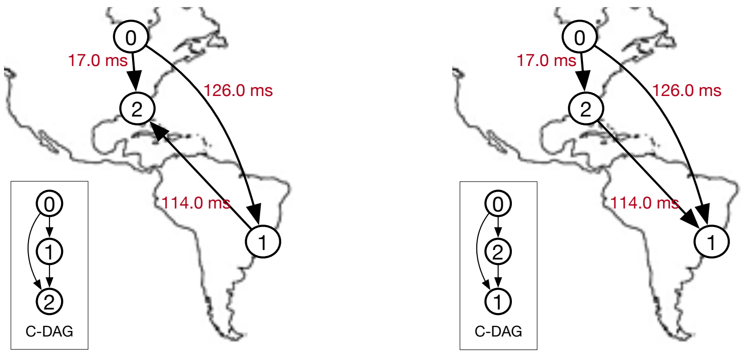


Figure 3.11. Initial configuration C_1 (left); and configuration C_2 (right), computed by Algorithm 5, for an overlay with 3 groups.

Initially, the system is set up with configuration C_1, unfavorable for the workload (Figure 3.11, left). For instance, one reason making configuration C_1 suboptimal for the given workload is that a significant portion (32.6%) of the requests are sent to groups 0 and 2. This leads to potential auxiliary messages being routed from group 0 to group 1 and then from group 1 to group 2 (a path with high latency) before group 2 can deliver those requests. After around 10 seconds, the reconfiguration protocol is activated, the algorithm determines a new overlay optimized for the current workload, and the system adopts this new configuration C_2 (Figure 3.11, right).

During reconfiguration (gray areas in Figure 3.12), the system undergoes a period of performance instability, as evidenced by disruptions in both throughput and latency plots. This instability arises because servers are executing the reconfiguration steps (i.e., allocating new data structures, managing new connections, and processing buffered requests). Moreover, in our protocol, clients must identify and adapt to the new configuration during communication with servers. When a client sends a request based on the old configuration, the server informs the client of the new configuration. The client must then retransmit the request according to the new configuration.

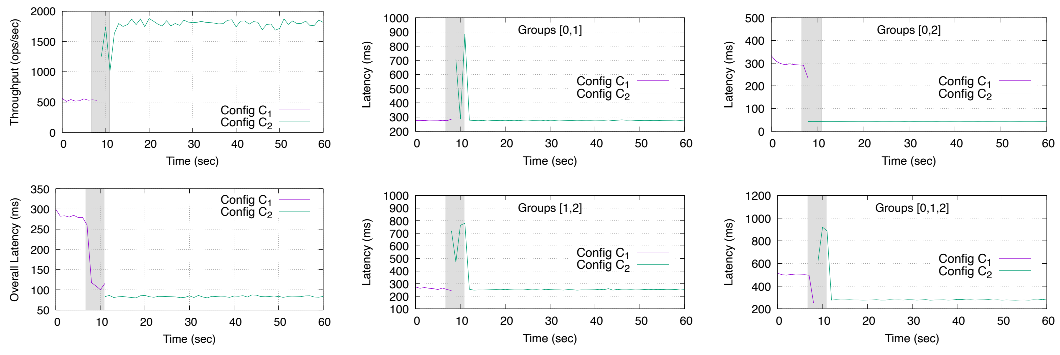


Figure 3.12. The impact of reconfiguration on system performance. Left column: system throughput in requests per second (top) and average latency perceived by the clients in milliseconds (bottom); middle and right columns: the 90th percentile latency per destination group. All plots show data for configurations C_1 and C_2, while the reconfiguration protocol runs 10 seconds into the execution. Gray area is the reconfiguration interval.

After this period of performance instability, the system's throughput improves and the response latency for clients significantly decreases (Figure 3.12, top left). Since clients execute in a closed loop, the lower latencies of configuration C_2 lead to augmented throughput. Results in Figure 3.12 show a significant reduction in request latencies for destinations $[0, 2]$ and $[0, 1, 2]$. With configuration C_2, for destinations $[0, 2]$, the `lca` group 0 can send requests directly to group 2 without involving group 1. This eliminates auxiliary messages traversing the high-latency path between groups 0, 1, and 2. As a result, group 2 can immediately deliver those requests without waiting for ACK messages from group 1, streamlining the communication process.

Regarding destinations $[0, 1, 2]$, the latency improvement is attributed to changes in the role of intermediary groups. In the previous configuration, group 1 was the intermediary, and it was also the group with the highest latency, making the path for receiving requests from group 1 and sending ACKs to group 2 the most latency-intensive. However, in the new configuration, group 2 serves as the intermediary. It receives the request from the `lca` group almost immediately and sends its ACK message in parallel with the `lca` group's request to group 1. This parallel processing allows group 1 to receive both the request and the ACK message nearly simultaneously, resulting in a total path latency that is significantly reduced compared to the previous configuration, resulting in better communication flow and reduced delays, leading to better system performance.

Figure 3.13 showcases FlexCast's capability to adapt to workload changes through multiple reconfigurations. Initially, when the system detected suboptimal performance with the initial overlay, it reconfigured to a more suitable one, resulting in improved throughput and reduced latency. Subsequently, a shift in workload locality, driven by clients, led to decreased performance. In response, the system's monitoring process identified changes in request destination frequencies, recalculated a new overlay, and successfully enhanced overall system performance again.

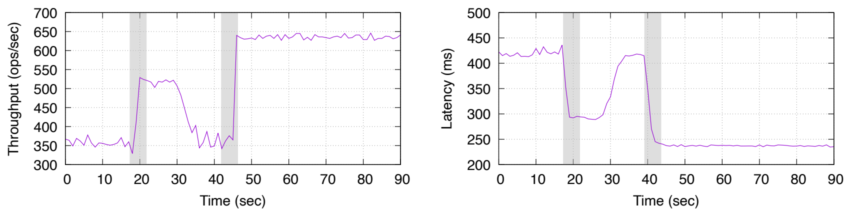


Figure 3.13. Execution of our reconfiguration protocol (gray area intervals) dynamically adapting FlexCast to different workload localities.

Impact on auxiliary messages As demonstrated in Figure 3.14 (left), despite the augmented throughput of configuration C_2, the number of auxiliary messages (ACKs and NOTIFs) decreases, which also contributes to the overall system performance. Figure 3.14 (middle) also reveals a shift in each group's data traffic per second. Specifically, groups 1 and 2 experience an inversion in data volume as their positions in the C-DAG hierarchy change, with the group at the end of the hierarchy typically receiving more data due to request histories and auxiliary messages. All groups see an increase in volume due to the augmented throughput of C_2.

Additionally, the internal size of the dependency processing history graph expands for all groups (Figure 3.14, right) because there is more throughput in the system. This phenomenon benefits groups lower in the hierarchy. These groups gain more

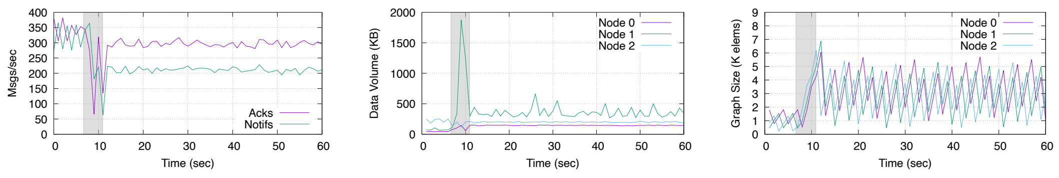


Figure 3.14. The impact of reconfiguration on auxiliary messages (left), data volume (middle), and size of internal data structure (a graph) used to compute dependencies in a group (right). Gray area is the reconfiguration interval.

frequent access to locally available dependency information rather than waiting for this information to arrive via auxiliary messages. This streamlines processing and enhances efficiency by reducing the need for inter-group communication for dependency data.

Increasing system size We also conducted experiments with different numbers of groups in the system. Figure 3.15 presents the results for an execution with 6 groups, where we only show the groups that experienced the most significant latency impacts. Groups not shown experienced negligible changes.

We notice that with more groups in the system, reconfiguration can disproportionately benefit certain groups over others, depending on the request frequencies in the workload. This is evident as, despite a significant increase in overall system throughput (Figure 3.15, left), the latency improvements are not uniformly distributed across all groups. However, destinations $[0, 2]$, which show a substantial improvement in latency (Figure 3.15, middle), account for the largest portion of the workload requests, explaining the marked increase in throughput. Additionally, the number of auxiliary messages also decreases significantly (Figure 3.15, right) after the reconfiguration.

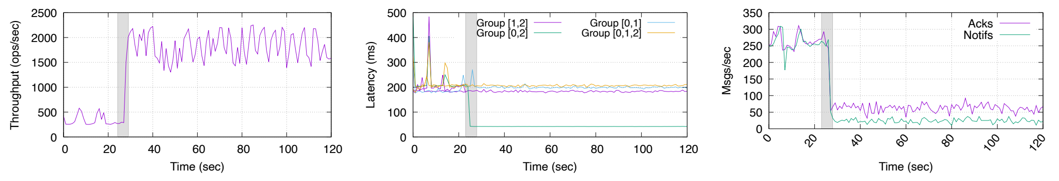


Figure 3.15. Reconfiguration performance with 6 groups and 98% locality rate. Gray area is the reconfiguration interval.

The cost of optimization Table 3.5 presents a comparative analysis of the performance of our optimization function, detailed in Algorithm 5, across various system configurations with different numbers of groups. The table displays the average time, in seconds, required to compute a new overlay based on observed workload destination frequencies and the system's size. We implementend a single and a multi-threaded version of our optimization function. Our observations indicate that while the time needed to compute a new overlay for systems with 3 to 9 groups remains relatively short, it increases significantly as the system size increases, in both versions. This issue arises because the current procedure implemented for Algorithm 1 performs an exhaustive search of all possible overlays. The algorithm could be designed to be more efficient if approached with heuristic methods. However, we did not pursue this optimization because the computation is handled in the background by a separate process, which does not impact the system's responsiveness. Additionally, in large-scale networks such as AWS, having more than 12 regions is generally unnecessary to represent remote areas sufficiently to cover the entire globe.

	3 groups	6 groups	9 groups	12 groups
Multi-thread	0.14	0.17	0.65	423.02
Single-thread	0.14	0.16	1.45	2452.48

Table 3.5. Mean execution time (in seconds) of our optimization function when varying the system size, for single and multi-threaded versions.

3.5 Related work

In this section, we present a comprehensive overview of atomic multicast and reconfiguration protocols proposed in the literature, highlighting their core mechanisms, strengths, and limitations.

3.5.1 Atomic multicast

One of the earliest approaches to atomic multicast, proposed by D. Skeen [13], involves a timestamp-based coordination among message recipients. In this protocol, a multicast message m is first sent to all its destinations. Upon receiving m, each destination assigns it a local timestamp and forwards this timestamp to the other recipients. Once a destination has gathered timestamps from all recipients, it computes the message's final timestamp as the maximum of the collected values. Messages are then delivered in the order of their final timestamps. Although this protocol ensures genuineness, it lacks fault tolerance.

Several atomic multicast protocols extend Skeen's ordering technique to tolerate failures [23], [39], [50], [69], [86]. In all these protocols, the idea is to implement destinations as groups of processes. Thus, messages are addressed to one or more process groups, instead of a set of processes, as in the original protocol. Although some processes in a group may fail, each group acts as a reliable entity, whose logic is replicated within the group using state machine replication [90]. Recent protocols aim at reducing the cost of replication within groups while keeping Skeen's original idea of assigning timestamps to messages and delivering messages in timestamp order. FastCast [23] improves performance by optimistically executing parts of the replication logic within a group in parallel. WhiteBox atomic multicast [50] uses the leader-follower approach to replicate processes within groups. RamCast [69] relies on distributed shared memory (RDMA) to reduce latency.

Delporte and Fauconnier [30] present a genuine distributed atomic multicast protocol that does not rely on exchanging of timestamps to order messages. The protocol assigns a total order to groups and relays messages sequentially through their destination groups following this order. A multicast message m is initially sent to the lowest group in $m.\texttt{dst}$ according to the total order. When the group receives m, it uses consensus to order and deliver m inside the group, then m is forwarded to the next group in $m.\texttt{dst}$, according to the total order of groups. A group that delivers m can only order the next message once it knows m is ordered in all groups in $m.\texttt{dst}$, which is after it receives an END message from the last group in $m.\texttt{dst}$. Although the dissemination of the message follows an order, the END message returns to each group involved and therefore the protocol is a distributed atomic multicast protocol. Besides needing $n+1$ steps to deliver a message, where n is the number of destinations of the message, since groups remain locked until the END message arrives, this protocol is affected by the convoy effect [2].

Some protocols restrict process communication by means of a tree overlay that determines how groups can communicate (e.g., [22, 42]). To order a message m using a tree, m is first sent to the lowest common ancestor group among those in $m.\texttt{dst}$, in the worst case the root of the overlay tree. Then, m is successively ordered by the lower groups in the tree until it reaches all groups in $m.\texttt{dst}$. An important invariant is that lower groups in the tree preserve the order induced by higher groups. Although simple, this protocol is not genuine since a message may need to be ordered by a group that is not in the destination set of the message. While the tree-based protocol proposed by Garcia-Molina and Spauster [42] does not tolerate failures, ByzCast [22] can withstand Byzantine failures.

The Arrow protocol [63] is a non-fault tolerant tree-based protocol that targets open groups. It emerges from the combination of a reliable multicast protocol with a distributed swap protocol. Arrow assumes a graph G and a spanning tree T on G. Initially, each node v in T has $link(v)$ that is its neighbour in T or itself if v is a sink (initially only the root of T). To multicast m a node v sends a message through

$link(v)$, which is forwarded to the root of the tree. By definition, the root has sent the last message before m. As the message is forwarded, edges change direction and v becomes the new root (that has sent the last message, which now is m). Although genuine, this procedure may result in swap messages traversing the diameter of T and only then a multicast, using an underlying reliable multicast, is issued.

Restricting communication as in a tree may lead to simpler atomic multicast algorithms. Moreover, if communication needs to be authenticated, as in Byzantine fault-tolerant protocols, a tree overlay requires fewer keys to be maintained and exchanged between processes than a distributed fully connected protocol. Finally, a fully connected protocol is a reasonable assumption in systems that run within the same administrative domain (e.g., Google's Spanner [26]). In other contexts (e.g., decentralized systems), however, multiple entities from different administrative domains collaborate but do not wish to establish connections with all other domains.

3.5.2 Reconfiguration

We are unaware of any works that address overlay reconfiguration of atomic multicast protocols. However, overlay reconfiguration has ben used to improve network security [33], network fault tolerance [80] and reliability of publish/subscribe systems [82]. In other context, reconfiguration of the set of replicas implementing the system has been used in many different distributed protocols, such as state machine replication [12, 90], quorum systems [3] and blockchains [10].

3.6 Conclusion

Here we summarize our key findings and contributions toward enhancing the performance of communication abstractions in SMR systems. Our first contribution introduced a new property, quiescence, to refine the minimality property, ensuring the integrity of genuine atomic multicast protocols by preventing unnecessary communication, and FlexCast, the first genuine overlay-based atomic multicast protocol, designed to reduce latency without incurring the communication overhead typical of non-genuine solutions. We thoroughly analyzed FlexCast's behavior across diverse experimental conditions and demonstrated its advantages over state-of-the-art atomic multicast protocols in geographically distributed deployments.

Our second contribution was the design and implementation of a dynamic reconfiguration mechanism for FlexCast. This extension enables the protocol to adapt its communication overlay in real time based on workload locality and system conditions. Through careful evaluation, we showed that this mechanism leads to substantial performance improvements in dynamic environments, making FlexCast not only efficient in static topologies but also robust and responsive to change.

3.6.1 FlexCast

As overlay-based, FlexCast accounts for reduced connectivity in different deployment scenarios. As genuine, it favors geographical locality and avoids communication overhead. To combine both aspects, FlexCast assumes a complete DAG overlay. Since messages may enter the overlay at different groups (nodes) of the DAG, each group takes local ordering decisions. We draw the following conclusions from our experimental evaluation:

- FlexCast is more sensitive to the chosen overlay than the hierarchical protocol when it comes to latency. The chosen tree, however, has an impact on the hierarchical protocol's communication overhead.
- FlexCast consistently outperforms the distributed protocol (a genuine algorithm) in all configurations experimented. FlexCast performs better than the hierarchical protocol in the first destination group and worse in the latency of the second and third destinations. However, messages addressed to three (or more) groups are rare in TPC-C and gTPC-C. As a genuine protocol, FlexCast has no communication overhead (Section 3.3.5), in contrast to a hierarchical protocol.
- The hierarchical protocol has a tradeoff between latency and communication overhead. Although communication overhead is inherent to non-genuine atomic multicast protocols, in the hierarchical protocol, trees with the best performance have the highest overhead and vice-versa.

One interesting challenge solved by FlexCast and not yet addressed by other atomic multicast protocols is how to ensure global acyclic order out of local ordering information from different groups. This is achieved using a sophisticated history-based protocol. We presented FlexCast's design, its implementation, and proposed a new benchmark to evaluate it: gTPC-C integrates geographical distribution and locality to the well-known TPC-C benchmark. FlexCast shows important latency reduction in geographically distributed settings when compared to a latency-optimum genuine atomic multicast algorithm and a non-genuine hierarchical protocol.

3.6.2 FlexCast's reconfiguration

In this second contribution, we extended FlexCast with a reconfiguration mechanism. We proposed, implemented, and evaluated FlexCast's reconfiguration scheme, enabling it to dynamically adapt to workload shifts. We show that our reconfiguration protocol delivers a system able to dynamically adapt to the changing demands of the workload, outputing better performance as system conditions evolve, making the system well-suited for dynamic environments. We draw the following conclusions from our experimental evaluation:

- Our reconfiguration protocol allows FlexCast to adapt to the locality of the current workload and modify its communication overlay in real time, resulting in significantly improved performance with the newly proposed configurations.
- Although the overall system performance is not significantly impacted, the algorithm for calculating a new configuration becomes inefficient as the number of groups increases. However, this issue could be addressed by employing an alternative approach that uses heuristics rather than brute force to determine a new overlay.

3.6.3 Final remarks

Overall, our results validate the design goals of FlexCast: to deliver low-latency atomic multicast in geographically distributed systems without incurring communication overhead. By combining a C-DAG overlay structure with a genuine, quiescent execution model and a dynamic reconfiguration mechanism, FlexCast proves to be a flexible and efficient solution for modern distributed systems. Its ability to adapt to workload locality and evolving deployment conditions makes it a promising direction for future dependable SMR systems.

Chapter 4

State Management in SMR Systems

Despite its success (e.g., [16, 26, 44, 56]), SMR remains complex to implement, and its overall performance is often constrained by the efficiency of its internal components. In this chapter, we are particularly interested in one of its specific components that is responsible for state management and synchronization in SMR systems.

In an SMR architecture, *state synchronization* is a key component impacting overall performance. It ensures that all correct replicas maintain a consistent internal state. State synchronization is crucial when a replica joins the system, recovers from a failure, or needs to catch up with faster replicas. As part of the state synchronization procedure, a replica receives the application state from one or more operational replicas. Many classic replication libraries delegate the responsibility of maintaining the state to the application level (e.g., [12]). As a result, the replication library itself is not responsible for managing the application state, which often leads to unsatisfactory performance during state synchronization, especially in the presence of Byzantine failures.

In this chapter we propose a shift in how SMR frameworks handle state management and synchronization, advocating for the integration of advanced data structures into SMR systems to enhance state management. We identify two key aspects that should be part of such data structures: *clustering* and *self-validation*.

Clustered data structures, in which data is logically grouped or partitioned (i.e., sharded) to improve locality and enable efficient verification, are widely used in authenticated, concurrent, and cache-aware systems [19, 46, 70, 96]. These structures are particularly advantageous in our scenario because they allow data integrity to be proven at any moment, ensuring security even under adversarial conditions. Moreover, unlike traditional Byzantine Fault Tolerant (BFT) systems, which require a quorum to validate reads or provide proofs, clustered structures can operate with a single operational replica, thereby improving performance while maintaining robustness.

Under malicious threats, a *self-validating* clustered data structure, in which data is partitioned and each cluster contains sufficient metadata for standalone integrity verification, enables replicas to verify data concurrently. Honest replicas can quickly detect invalid clusters, blacklist a Byzantine replica, and refetch corrupted clusters from another source. In contrast, a regular data structure can only detect corruption after downloading the entire state, resulting in significantly longer synchronization times and increased data transfer.

We illustrate our proposed approach introducing two novel data structures: B+AVL tree and SVCSkipList. A B+AVL tree is a novel cluster-based data structure inspired by AVL* [41] and Merkle B+Trees, building on their advantages while avoiding their shortcomings. A SVCSkipList is a self-validating clustered skiplist. Skiplists are widely used data structures for implementing key-value stores, with notable applications in systems such as LevelDB (Google) [49], RocksDB (Facebook) [36], and Redis [87]. One of their main advantages is the simplicity of providing interfaces that support essential operations such as search, insert, and delete, making them ideal for application-level use. We implemented both proposed data structures and evaluated them in different settings, showing that they achieve better performance than state-of-the-art baseline approaches in many scenarios, especially in the presence of Byzantine failures.

4.1 Self-validating and clustered data structures

A self-validating data structure is one that inherently includes mechanisms or metadata to verify its correctness and integrity. Merkle trees or similar balanced (AVL) trees (e.g., Merkle-Patricia trees [94]) are self-validating data structures typically used in blockchains. In addition to providing efficient data access, these structures empower replicas with the ability to prove the integrity of the state to the clients. In a Merkle-tree, each leaf holds a piece of the state and its hash, and each inner node holds the hash of its children. Merkle trees provide succinct proofs of data integrity of any node of the tree. One can prove in logarithmic time and space that a leaf v belongs to the tree by providing the hash of the siblings of the tree nodes in the path from v to the tree's root.

A clustered data structure is an arrangement where related data elements are stored together, typically to optimize access patterns or improve cache performance (e.g., [19]). This is often achieved by grouping data based on specific attributes or keys. Merkle trees allow data to be clustered into smaller chunks (e.g., [41]), each of which can be independently validated. This means that during state transfer, the receiving replica can verify the integrity of the data received by checking hashes at various levels of the tree. If any part of the data has been tampered with or corrupted, it can be identified and isolated without compromising the entire state. This self-validation mechanism not only enhances the reliability of state transfer but also improves the efficiency of the process, as only the invalid portions of the state need to be re-transferred and re-validated.

Moreover, clustered data structures can significantly improve the efficiency of serialization and deserialization. In some blockchain systems, for instance, entries from a tree data structure are grouped into batches, allowing the system to serialize and deserialize these batches rather than processing each entry individually. This batching approach reduces overhead and accelerates state synchronization, leading to better overall performance. Peers in CometBFT, for example, use AVL+ trees (see Section 4.2.1) for state management [27]. A peer serving a snapshot traverses the tree, groups tree nodes into clusters, and serializes the whole cluster. Serializing a cluster that contains multiple tree nodes is significantly more efficient than serializing each node individually. This efficiency arises from grouping all the nodes within a cluster together in contiguous memory, which streamlines the serialization process.

When nodes are stored contiguously, the entire cluster can be serialized in a single operation, eliminating the need for looping through individual nodes. In contrast, if a binary tree is used where nodes are scattered across memory, each node must be serialized separately, requiring a loop to iterate through all nodes. Figure 4.1 compares the performance of serializing and deserializing a tree using the two techniques, clustered data and individual data items. Both serialization and deserialization benefit significantly from clustered nodes, and the advantage increases with cluster size. The clustered approach outperforms the serialization and deserialization of individual nodes by a factor of 26× when clusters contain 4096 data items, and by a factor of nearly 6× when clusters contain 1024 data items.

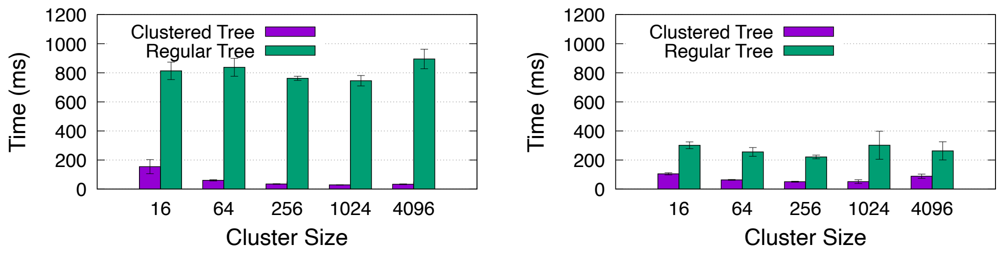


Figure 4.1. Time to fully serialize (left) and deserialize (right) a clustered and a regular tree with 1 million elements, and various cluster sizes (each element is 512 bytes).

Although clustering speeds up serialization and deserialization, it introduces the overhead of building clusters during state transfer. A more aggressive alternative is to use data structures naturally organized around larger data units, such as AVL* trees [41] and Merkle B+Trees [40], where a peer serving a snapshot can avoid on-the-fly tree traversal and cluster construction. Recovering peers also benefit by not needing to allocate individual tree nodes.

In addition to (de)serialization efficiency, clustered structures offer another crucial advantage in Byzantine settings. When state is organized into independently verifiable clusters, corrupted clusters can be detected immediately upon receipt, without requiring the download of the entire tree or structure. This enables early detection and rejection of invalid data, significantly speeding up recovery. In contrast, non-clustered structures must often download and process the complete structure before validation can occur, exposing systems to much higher delays and inefficiencies when facing faulty or malicious peers.

We begin by proposing a specialized data structure within the context of blockchains, a specific instance of state machine replication (SMR) systems, using a Merkle-tree-like structure. We then generalize this approach by extending it to broader SMR systems, where we aim to integrate these advanced data structures to improve state management and synchronization. Moreover, we explore adapting other types of data structures beyond AVL trees to support clustering and self-validation.

4.2 B+AVL trees: efficient blockchain state synchronization

In this section, we address state transfer in blockchain systems. Similarly as in generic SMR systems, it occurs when a peer recovers from a failure or joins the blockchain network. To catch up with the rest of the system, the peer must update its state to match the operational peers. Modern blockchain systems (e.g., [27, 94]) use periodic snapshots of the system state to improve state transfer performance. A recovering peer first downloads a recent snapshot and subsequent blockchain data (i.e., full blocks or block headers containing summaries). The peer installs the snapshot and then replays the transactions that occurred after it to fully reconstruct the current system state.

In many blockchain systems, peers store state in a special data structure, such as a Merkle tree [76] or Merkle-Patricia-tree [94]. In Tendermint's AVL+ trees [1], every leaf node stores a cryptographic hash of its data, and every inner node stores a hash calculated from the hash of its children. A recovering peer can validate a snapshot by computing the root hash of the Merkle tree and comparing it with the root hash stored in the trusted block header.

To decrease the time it takes for a new peer to join the blockchain, a snapshot can be divided into *clusters*, each containing multiple tree nodes, and a recovering peer can download clusters from many operational peers concurrently. Variations of this technique have been implemented by both blockchain (e.g., [28]) and Byzantine-fault tolerant state machine replication systems (e.g., [11, 12, 18]). Some approaches have been proposed to improve the performance and robustness of this scheme [40, 41]. Building clusters from the state tree to serve a recovering peer and rebuilding tree nodes from clusters can be an expensive operation involving tree traversal, serialization, and deserialization of tree nodes. This cost can be substantially reduced by storing the system state in "cluster-based" data structures, as in AVL* trees [41] and Merkle B+Trees [40]. While AVL* trees embed state subtrees into clusters, Merkle B+Trees store the leaves of a cell in a cluster. In both cases, a peer serving the state to a recovering peer must only traverse and serialize the list of existing clusters. The recovering peer receives large data units, saving resources similarly to batching [38], and avoids allocating individual tree nodes by relying on clusters.

In this context, we introduce B+AVL trees, a novel cluster-based data structure inspired by AVL* and Merkle B+Trees, building on their advantages while avoiding their shortcomings. B+AVL trees are balanced binary trees that use rotations, like AVL trees, and integrate Merkle hashes to enable individual validation of nodes and subtrees, as in AVL* trees. However, in AVL* trees, maintaining balance through rotations can affect cluster roots, often requiring cluster splits before or after rotations and adding significant complexity. B+AVL trees, inspired by B+Trees, avoid these complications by filling clusters sequentially and splitting them when full, without requiring complex cross-cluster rotations. They are more space-efficient than AVL* trees, allowing clusters to store more data, and they provide more compact validation proofs than Merkle B+Trees.

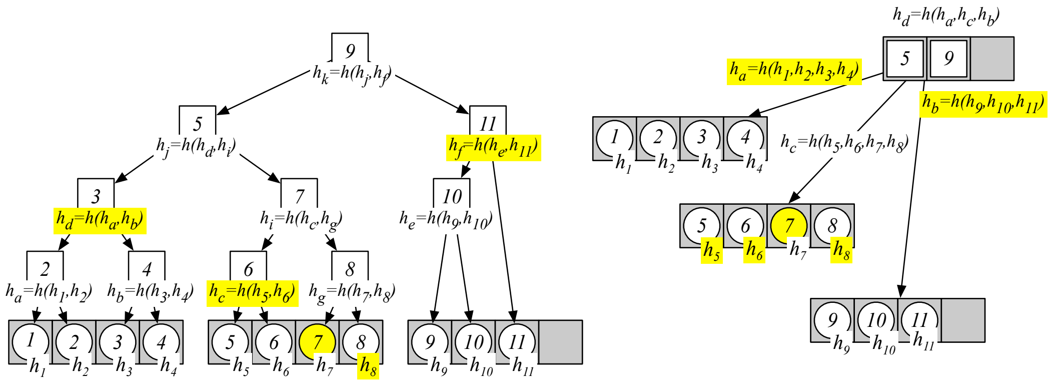
Table 4.1 compares the main properties of blockchain data structures discussed in this section. It highlights the size of cryptographic proofs (proof size) and whether it supports self-verifiable clusters. The proof size denotes the total number of elements (hashes) required to verify the inclusion of a specific entry in the tree. The comparison considers AVL+ trees, AVL* trees, Merkle B+Trees, and B+AVL trees, where n denotes the total number of entries in a tree, and b the number of entries in a B-tree cell.

Data structure	**Proof size**	**Self-verifiable clusters**
AVL+ tree	$\mathcal{O}(\log_2 n)$	no
AVL* tree	$\mathcal{O}(\log_2 n)$	yes
Merkle B+Tree	$\mathcal{O}(\log_b n \cdot b)$	yes
B+AVL tree	$\mathcal{O}(\log_2 n)$	yes

Table 4.1. Blockchain data structures (n: number of entries, b: entries in a B-tree cell).

4.2.1 AVL* trees and Merkle B+Trees

AVL* trees extend AVL and AVL+ trees. An AVL tree is a self-balancing binary tree where the heights of a node's children differ by at most one. When this balance is violated (after insertions or deletions), rotations restore it, ensuring logarithmic-cost search, insertion, and deletion operations. Insertions may require up to two rotations; deletions can require rotations proportional to the tree's height. An AVL+ tree is a variant where only leaves store values; inner nodes exist only in memory and can be recomputed from the leaves, reducing persistent storage needs.



(a) AVL* tree (b) Merkle B+tree

Figure 4.2. Different data structures used to store state in a blockchain peer: (a) AVL* tree and (b) Merkle B+Tree. (Circles depict keys and values, white squares contain inner keys, used to navigate the data structure, gray area represents data clusters.) The proof that 7 is a valid element in the AVL* tree includes hashes h_8, h_c, h_d and h_f, while in the Merkle B+Tree the proof includes hashes h_5, h_6, h_8, h_a and h_b (highlighted in yellow).

AVL* trees [41] are AVL+ trees where leaves are batched into larger data structures (i.e., clusters) to improve state management and synchronization (see Figure 4.2 (a)). To provide a proof of integrity of any element of an AVL* tree (i.e., leaf), tree nodes are labeled with cryptographic hashes, leading to what is known as a hash tree or Merkle tree. In a Merkle AVL tree, each leaf has the hash of its value, and each inner node has the hash of its children. In Figure 4.2 (a), the highlighted hashes h_8, h_c, h_d, and h_f form the proof required to verify the inclusion of leaf 7 in the tree. These hashes are kept up-to-date as the tree evolves, and to retrieve them, a peer traverses the path from the leaf to the root, collecting the sibling hashes along the way. During validation, a peer uses these hashes to reconstruct the path up to the root and compares the resulting root hash to the trusted root hash stored in the corresponding block header. If the computed and stored root hashes match, leaf 7 is confirmed as a valid entry. A proof in a Merkle AVL tree has size $O(\log_2 n)$, where n is the number of tree leaves.

B-trees generalize binary trees by allowing each inner node to have between $\lceil m/2 \rceil$ and m children, where m is the tree order [61]. All leaves are at the same level, and searches, insertions, and deletions have logarithmic time complexity. B-trees are designed for storage and communication efficiency by configuring cell sizes via the tree order. In a B+Tree, internal nodes store copies of the keys, while leaves store keys and values; leaves may also include pointers to their successors for faster sequential access.

Differently from AVL* trees, which require sophisticated algorithms to maintain parts of the tree in clusters without violating tree invariants, Merkle B+Trees are a straightforward variation of B+Trees, extended to provide proofs of integrity [40]. Each entry in a leaf cell has the hash of its value, and each entry in an inner cell has a hash computed based on the hash of its children. In Figure 4.2 (b), the integrity of leaf 7 can be checked with hashes h_5, h_6, h_8, h_a and h_b. A proof in a Merkle B+Tree has size $O(b \times log_b\, n)$, where n is the number of leaves in the tree and b the number of elements in a cell.

Discussion Both AVL* trees and Merkle B+Trees improve blockchain data management and state synchronization by embedding clustering into the tree structure. Cluster-based designs enable efficient data serialization and deserialization, reducing communication overhead. Another advantage is the use of *self-verifiable* clusters: a recovering peer can immediately validate a cluster upon receipt by checking an embedded subtree root proof [41], without rebuilding the full tree. This significantly improves recovery performance in the presence of malicious peers, which might otherwise delay synchronization by serving invalid data (see Section 4.2.3).

4.2.2 The B+AVL tree

The B+AVL tree is an advanced data structure designed for efficient storage, search, and proof generation over large collections of key-value pairs. It combines features of self-balancing binary search trees (like AVL trees) and clustered leaf storage (as in B+Trees), allowing both efficient updates and compact proofs. The B+AVL tree maintains balance across two levels: the global tree structure and the internal organization of clusters within leaves.

B+AVL tree structure A B+AVL is a tree at two levels. Firstly, it is a self-balancing binary search tree with each inner node containing the hashes of its two child nodes and a copy of the smallest key in their right subtree (see Figure 4.3). Differently from an AVL* tree, however, leaf nodes do not only host space for a single key-value pair.

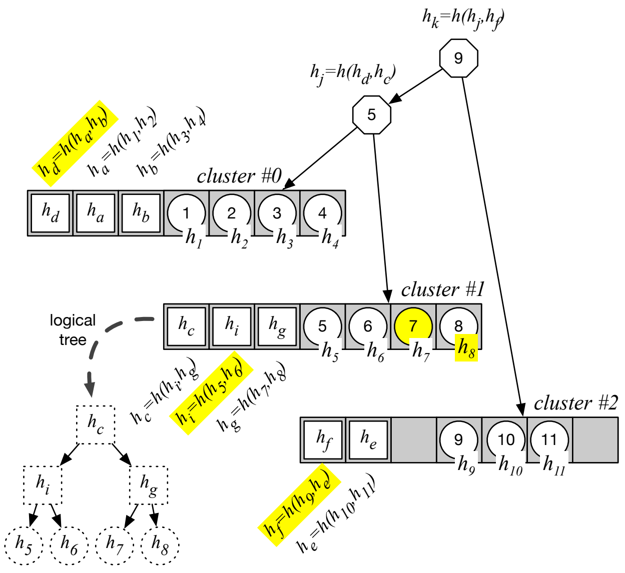


Figure 4.3. The B+AVL tree. Gray area represents data clusters, circles depict keys, octagons contain inner keys, white squares are hashes that build up a logical tree. The proof that 7 is a valid element contains hashes h_8, h_i, h_d and h_f.

Instead, leaf nodes are implemented like B+Tree leaf nodes, effectively representing a cluster of data. This allows to have clusters in contiguous memory for efficient serialization. When a cluster is full, it is split and the two halves are hosted in two new leaf nodes. A new inner node is then added to link the leaves.

To avoid linear terms in the size of a proof for individual elements, which would be introduced by the flat clusters, each cluster is augmented with an additional, *in-cluster*, hash tree. This is not a search tree and its sole purpose is to provide logarithmic-sized proofs to elements of a cluster. We refer to the root of this tree as the *cluster root*. A cluster root provides the first elements of the hash proof for a whole cluster. When a cluster proof and the proof of an element within a cluster are concatenated, a logarithmic-sized proof for element within the overall tree is obtained.

A B+AVL tree rotates inner nodes by treating the leaves containing a cluster, no matter the size, as a unit. Like an AVL tree, the height of two child nodes can differ by 1 at most. However, because a cluster vary from being half-full to full, the height difference between child nodes in the in-cluster tree can also differ by up to 1. Consequently, after expanding clusters in the logical tree view, the height difference between the children of any node in a B+AVL tree can be up to 2, compared to the maximum difference of 1 in an AVL tree. This, however, avoids the complex rotations across clusters that a AVL* tree needs to undergo to stay balanced.

B+AVL clusters A B+AVL tree represents a key-value store with fixed-sized keys and variable sizes values. Each cluster contains three arrays for keys, values and hashes respectively. The keys array has known size and contains sorted keys. Each key is augmented with the size and starting position of the corresponding value in the values array. This permits new values to simply be appended to the values array and limit data movement during insertions. We pre-allocate an estimated size for the value array and if the overall size is surpassed, the array is re-allocated. For a cluster with M keys, the hashes array contains $2M - 1$ entries to represent the nodes of in-cluster hash tree in flat memory. The first half of this array contains the hashes of inner nodes of the tree embedded in the cluster, while the second half contains direct hashes of key-value pairs. Thanks to the splitting mechanism, it is easy to link consecutive clusters to simplify range queries.

Searches and proofs To find a value, the inner nodes are traversed until a leaf node is reached. During this traversal, a cluster proof can be obtained by keeping track of the hash values of the sibling's hash for each node encountered. Within the cluster, the value is retrieved with a binary search in the keys array. Because the in-cluster tree is not a search tree, the proof for the specific element can not be obtained while searching. Instead, one must traverse the in-cluster hash tree upwards knowing the parent relationship[1] of each node. A proof of an individual element is obtained by concatenating the cluster proof and the element proof within the cluster. For instance, in Figure 4.3, the integrity of leaf 7 can be checked with hashes h_8, h_i, h_d, h_f, and the root hash in the block header.

B+AVL tree rebuild To rebuild the tree, a peer must request all clusters, in any order, and validate them individually. To do that, the key and value arrays are deserialized, and the in-cluster tree is recomputed by the recovering peer. Then, using the computed cluster root, the cluster proof provided by a correct peer will allow the recovering peer to ensure its validity. For instance, in Figure 4.3, cluster #1 can be proved with hashes h_d and h_f. Once all clusters are deserialized, those need to be sorted and the inner tree structure can be rebuilt. Since this internal structure contains only hashes and key copies, without any actual data, and all data has been previously verified, the integrity of the reconstructed tree is guaranteed, eliminating the possibility of an invalid overall structure. To ensure the same tree structure is obtained, we must note that every inner node of the B+AVL tree contains a copy of the smallest key in the left-most cluster of its right subtree. The inner node is found at a certain height in the tree structure and keeping track of this key height information in all clusters is enough to fully encode

[1]For a 0-indexed array, parent(i) $= \lfloor \frac{i-1}{2} \rfloor$.

and hence rebuild the structure of the inner tree. The same approach is used in the AVL+ and AVL* trees.

4.2.3 B+AVL tree evaluation

We now provide technical details about the data structures implementation and the experimental setup. We then compare B+AVL trees to competing approaches considering performance, proof size, space efficiency, and state synchronization aspects.

Implementation and setup We implemented prototypes of all data structures in Go. Our implementations are based on Tendermint's AVL+ tree [1], retaining only essential features. We use Amino (a Proto3 subset) for serialization. The implementation of the B+AVL tree is publicly available to support reproducibility and further experimentation[2]. We evaluate four data structures: B+AVL, AVL*, B+Tree, and a baseline AVL+ implementation, from now on called Baseline, in which clusters are filled with serialized leaves until the target size, possibly spanning subtrees and lacking proof validation.

We evaluate the data structures in both standalone and distributed scenarios. The standalone tests focus on insertion/search performance, proof sizes, and space use. In the distributed setting, we assess state synchronization costs and resilience to Byzantine attacks. A recovering peer uses a scheme like the one used in AVL* [41] to track cluster indices and detect when the full tree is retrieved. We simulate an attack where a malicious peer corrupts random clusters. Our benchmarks compare B+AVL to the Baseline tree, which must refetch the entire state upon attack, highlighting B+AVL 's efficiency under such conditions.

The standalone experimental setup is composed of a machine equipped with two AMD EPYC 7282 16-Core processors, 32 GB of RAM, running Linux Ubuntu 18.04.6, and Go version 1.20.2. The distributed experiments were conducted in CloudLab [34], where the setup was configured with 3 to 12 server machines (active peers) and 1 client machine (recovering/new peer), all interconnected via a 1-Gbps switched network. We emulated a WAN environment by incorporating latencies observed in AWS regions across Europe. The machines are equipped with eight 64-bit ARMv8 (Atlas/A57) processors, 64 GB of RAM, running Linux Ubuntu 20.04, and Go version 1.13.8.

Performance Our initial experiments assess the performance of constructing various trees through incremental insertions and searches within AVL* trees, Merkle B+Trees, B+AVL trees and the Baseline tree. We compare these techniques using clusters of 256

[2]https://github.com/MicheleCattaneo/B_plus_AVL

elements while varying the overall tree size. In all experiments, each element is 512 bytes in size, with 4-byte keys.

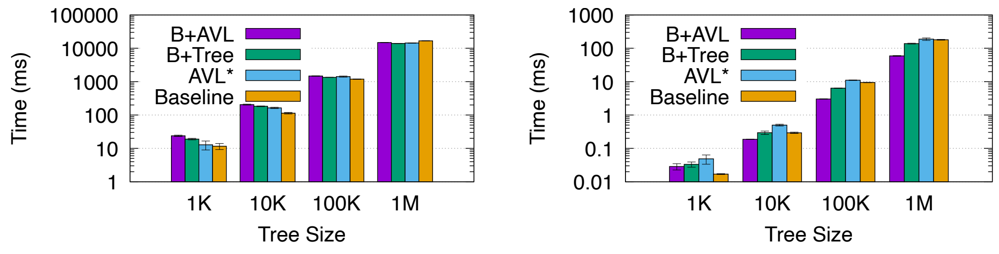


Figure 4.4. Performance of insert (left) and search (right) operations for all four data structures, using trees with 512-byte elements and 256-element clusters, across varying tree sizes. The y-axis (log scale) shows the average time to insert all elements or search for 10% of them. Whiskers indicate standard deviation.

Figure 4.4 (left) shows the time required by the four techniques to construct trees of five different sizes, using the same sequence of elements from a shared dataset. For smaller trees (1K and 10K elements), the B+AVL tree performs slightly worse than the others, followed by the B+Tree, due to B+AVL's need to recompute all hashes in a cluster after each insertion. While optimizing hash computation could improve this, it would increase complexity. As tree size grows (100K and 1M elements), performance differences between techniques diminish.

Figure 4.4 (right) shows the search performance of the four techniques when querying 10% of the elements in trees of varying sizes. All techniques use the same key sequence. For small trees, the Baseline tree performs best, while for larger trees, the B+AVL tree outperforms the others. This is due to its contiguous memory layout within clusters, which improves cache utilization by minimizing cache misses during search operations. Specifically, using the Linux `perf` tool to collect hardware performance counters, we observed that the B+AVL tree achieved the lowest cache miss rate of 26.7%, followed by the B+Tree that exhibited a cache miss rate of 29.0%. The AVL* tree showed a rate of 30.6%, and the Baseline tree had the highest rate, at 33.1%.

Proof size Figure 4.5 presents the distribution of proof sizes for trees with 1 million elements while varying the cluster sizes. The proof size for each element was measured after all elements were inserted. The Merkle B+Tree (bottom left) exhibits a linear increase in proof size (see also Figure 4.2 (b)). This increase occurs because proving a value requires the hashes of all siblings for each node. Consequently, as the cluster size grows, the number of required hashes also increases, a significant drawback for B+Trees.

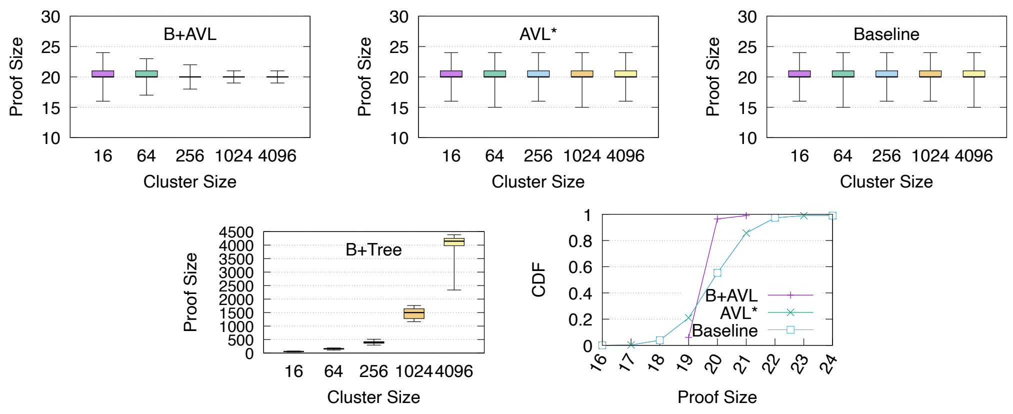


Figure 4.5. Proof sizes (boxplots) measured on a tree with 1 million elements after all insertions, varying the cluster sizes. Whiskers represent the minimum and maximum values. The CDF illustrates the distribution of proof sizes for the B+AVL, AVL*, and Baseline trees with a cluster size of 1024, revealing that B+AVL proofs have lower variance.

In contrast, B+AVL, AVL*, and the Baseline tree exhibit much smaller proof sizes that remain unaffected by cluster size. Additionally, the B+AVL tree demonstrates a desirable feature of reduced variance in proof size as the cluster size increases. AVL trees maintain balance by ensuring the height difference between children of a node is at most one, while the length of a proof is directly proportional to the depth of the leaf nodes, which presents some variance. In contrast, B+AVL trees build subtrees within clusters to have a maximum depth difference of only one. This design results in a smaller variance in proof length when considering individual clusters. As the cluster size increases, this balanced structure exerts a greater influence on the overall proof length, providing the B+AVL tree with a distinct advantage. This advantage is also perceived in the cumulative distribution functions shown in the bottom right figure, where the B+AVL tree demonstrates more consistent and shorter proof lengths compared to the AVL* and Baseline trees.

Space efficiency In the next set of experiments, we evaluate the space efficiency of all techniques. We define space efficiency as the ratio of the actual number of clusters for the tree to the minimal number required in an ideal scenario. For example, a space efficiency of 2 indicates that there are twice as many clusters as would be needed in the optimal case. The worst-case scenario for the B+Tree and B+AVL trees is when all clusters are half full [24], leading to space efficiency of 2. The Baseline tree (not depicted in the plots) offers the best space efficiency, equals to 1. Therefore, we will

Data structure	Cluster size				
	16	64	256	1024	4096
AVL*	1.58 (±0.01)	1.56 (±0.01)	1.50 (±0.02)	1.51 (±0.02)	1.48 (±0.04)
B+Tree	1.42 (±0.00)	1.43 (±0.00)	1.43 (±0.06)	1.41 (±0.17)	1.40 (±0.23)
B+AVL	1.42 (±0.00)	1.44 (±0.01)	1.44 (±0.03)	1.43 (±0.13)	1.41 (±0.24)

Table 4.2. Average (and standard deviation) space efficiency of the three data structures with varying cluster sizes.

focus our discussion on the remaining three trees.

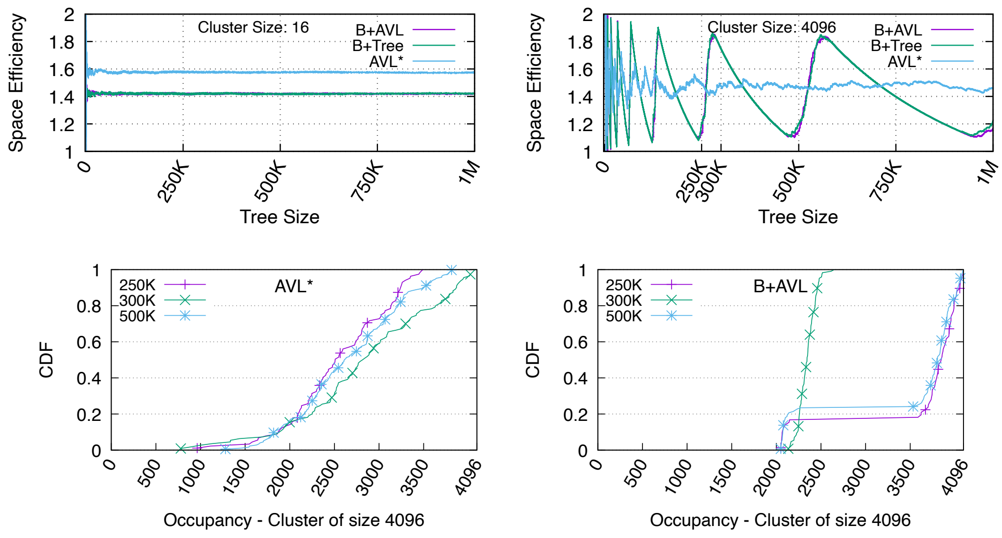


Figure 4.6. Space efficiency after inserting each element, with 16 (top left) and 4096 (top right) clusters. The x-axis shows the number of elements in the tree, and the y-axis represents space efficiency. Cumulative distribution functions for cluster occupancy distribution at three different points during element insertion with clusters of size 4096, for AVL* (bottom left) and B+AVL (bottom right).

In Figure 4.6, the two plots at the top illustrate the space efficiency after each element is inserted. The x-axis represents the number of elements in the tree, while the y-axis depicts the space efficiency. On the left, the cluster size is set to 16, and on the right, it is set to 4096. We observe that for small cluster sizes, all three trees exhibit stable space efficiency as the tree grows, with the B+AVL and B+Tree showing the best results. However, with larger cluster sizes, all trees display more unstable space efficiency as the tree grows, especially the B+AVL and B+Tree.

While the average space efficiency is similar across all trees with cluster size 4096 (see Table 4.2), the AVL* maintains a more consistent space efficiency compared to the B+AVL and B+Tree, which exhibit greater fluctuations as elements are inserted. This behavior occurs because B+Trees tend to fill all clusters before increasing their heights, given that elements are uniformly distributed for insertion. When clusters are nearly full (i.e., space efficiency close to one), subsequent inserts tend to cause splits, reducing space efficiency. As most clusters split (i.e., space efficiency approaching two), they become half full and start to fill up again. This issue does not arise with the AVL*, since the tree height is not as closely linked to the number of clusters.

This phenomenon is also observable in the two cumulative distribution functions at the bottom of Figure 4.6, where we measure the cluster occupancy distribution at three different points during element insertion with clusters of size 4096. The AVL* (bottom left) displays a more uniform cluster occupancy distribution, ranging from half full to full at all three points. In contrast, the B+AVL (bottom right) has varying distributions: with 300K elements, as the space efficiency approaches 2 (see the plot on top), the distribution indicates that most clusters are nearly half full. However, with 250K and 500K elements, where space efficiency approaches the optimal, most clusters are nearly full.

State synchronization In the distributed state synchronization experiments, we compare B+AVL only to the Baseline tree, as it reflects the widely used approach in practice (e.g., Tendermint's AVL+) despite lacking self-verifiable clusters and requiring full re-transfer after attacks. We exclude AVL* and B+Tree because they are either standalone structures not used for state synchronization (B+Tree) or employ similar cluster verification to B+AVL (AVL*), thus offering comparable resilience but without B+AVL's space and proof efficiency.

Figure 4.7 compares the B+AVL with the Baseline tree, which builds clusters by simply filling them as it traverses the tree, resulting in non-self-verifiable clusters. The plots illustrate results from experiments conducted in an emulated wide-area network with a tree containing 100K elements. The plots on the left show state synchronization with only honest peers, while those on the right include one Byzantine. The top graphs display the time (in milliseconds) required to transfer the entire tree with varying cluster sizes and 9 peers serving clusters. The middle plots illustrate the impact of varying the number of peers serving clusters with a cluster size of 256. The bottom graphs depict the number of clusters sent by each technique as the cluster size varies with 9 peers serving clusters.

We observe that in the absence of attacks, the B+AVL performs slightly better than the Baseline, despite sending more clusters. This advantage is attributed to the fact that the serialization, communication, and rebuild times of the Baseline are higher than in

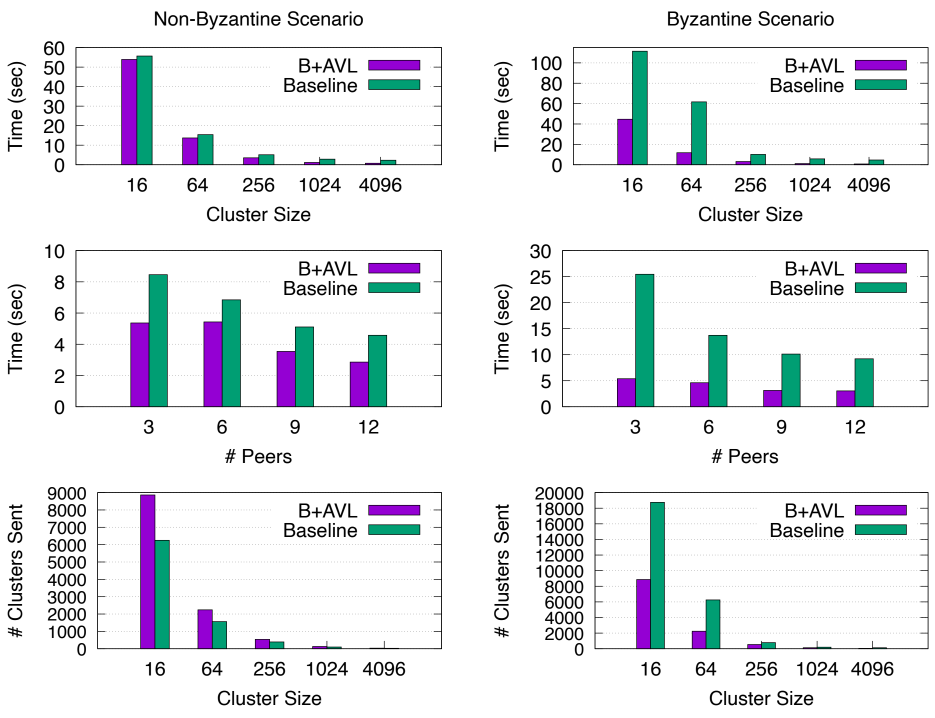
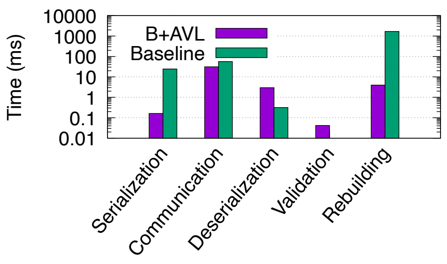


Figure 4.7. State synchronization time of B+AVL and Baseline trees without attacks (left column) and with 1 Byzantine peer (right column) for a tree of size 100K: performance versus cluster size with 9 peers (top); performance versus number of peers with 256 clusters (middle); and the number of clusters sent during synchronization as we vary cluster size with 9 peers (bottom).

B+AVL (Figure 4.8 left). Serialization in the Baseline tree needs traversing the entire tree to fill up the clusters. In the B+AVL tree, clusters are directly accessible, allowing for more efficient serialization, as the memory in each B+AVL cluster is contiguous. Consequently, serialization can be performed with a single call per cluster, compared to the Baseline, which requires a call for each leaf. Additionally, contiguous memory in the B+AVL clusters improves cache performance, particularly for keys, as multiple keys can fit into the same cache line. This efficient memory usage further speeds up the serialization process (see also Figure 4.1). Rebuilding is also faster in the B+AVL tree due to its streamlined structure. In the B+AVL, there is only one inner node per cluster, whereas in the Baseline tree, there is one inner node for each leaf node. This means the Baseline has significantly more inner nodes to rebuild, slowing down the process. Deserialization instead takes longer in the B+AVL tree since the hash values of the logic tree within the cluster have to be computed, in order to validate the individual clusters. In contrast, the Baseline tree only requires the deserialization of individual



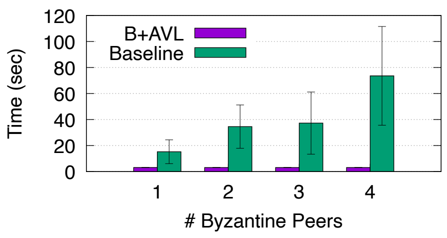


Figure 4.8. Left: performance breakdown showing time (ms, log scale) for each phase of state synchronization with a 100K-element tree and 256-element clusters. Right: B+AVL and Baseline tree performance under Byzantine attack, with 100K elements and 256-element clusters, varying the number of Byzantine peers.

leaves, making this process more straightforward and less resource-intensive.

Under Byzantine attacks, B+AVL trees enable peers to verify data integrity with minimal computational overhead. Their small proofs allow rapid verification and the advantage of self-verifiable clusters becomes clear. Honest peers using B+AVL trees can quickly detect invalid data, blacklist the Byzantine peer, and refetch corrupted clusters from another source. In contrast, the Baseline tree can only detect corruption after downloading the entire tree, leading to much longer synchronization times and increased data transfer. Figure 4.7 right shows that the Baseline tree can take up to 4× longer and require twice as many clusters as B+AVL.

The performance gap worsens for the Baseline tree as the number of potential Byzantine peers increases, as shown in Figure 4.8 right, which illustrates the impact of varying the number of Byzantine peers up to four. With more Byzantine peers, the Baseline tree becomes increasingly inefficient, leading to longer synchronization times. In contrast, the B+AVL tree maintains strong performance, demonstrating its superior resilience and efficiency in managing Byzantine attacks. Interestingly, in some cases, the B+AVL tree completes state synchronization faster in Byzantine scenarios than in non-Byzantine ones (e.g., Figure 4.7 top with a cluster size of 16). This can happen if the Byzantine peer is in a distant region on the network. Once blacklisted, the peer fetches clusters only from closer peers, speeding up synchronization. This underscores the B+AVL tree's ability to effectively manage Byzantine peers and optimize peer selection, further improving synchronization times in certain conditions.

4.3 SVCSkipList: efficient SMR state synchronization

In this section, we discuss self-validating clustered data structures to enhance state management and synchronization in general-purpose SMR systems. In the previous section, we introduced the B+AVL tree, a data structure to enable efficient state validation in blockchain systems. Blockchains are a specific instance of SMR systems, in which the state is represented as a sequence of blocks, and each block contains a set of transactions. We now extend this concept to a more general context, and propose to use self-validating clustered data structures in generic SMR systems. We propose a novel data structure, the SVCSkipList, a self-validating clustered skiplist, and integrate it into a well-known SMR framework, BFT-SMaRt [12], to enhance its state management capabilities. In this section, we will discuss the adoption of such a structure, highlighting its potential benefits and applicability in improving SMR resilience and performance.

4.3.1 Skiplists

A skiplist maintains a set S of ordered elements organized into a sequence of linked lists $S_0, S_1, S_2, \ldots, S_t$. The base level S_0 contains all elements of S in sorted order, together with sentinel elements $-\infty$ and $+\infty$. Each higher level S_i, for $i \geq 1$, stores a subset of the elements from the level immediately below, S_{i-1}.

The method used to select elements from one level to the next defines the type of skiplist. In the randomized variant, each element of S_{i-1} is independently promoted to S_i with a fixed probability p (typically $p = 0.5$). Alternatively, deterministic skiplists [78] use fixed rules to ensure that between any two elements in S_i there are at least one and at most three elements in S_{i-1}.

In both variants, the sentinel elements $-\infty$ and $+\infty$ are always promoted to higher levels, and the number of levels t is maintained as $O(\log n)$. The top level S_t contains only the sentinel elements; the node storing $-\infty$ at this level is referred to as the start node s.

A node in S_{i-1} whose element does not appear in S_i is called a *plateau node*, whereas a node whose element is also present in S_i is called a *tower node*. Consequently, between any two tower nodes at the same level there exists a sequence of plateau nodes. In deterministic skiplists, the number of plateau nodes between two tower nodes is bounded between one and three, while in randomized skiplists the expected number is one.

Each node in a skiplist is associated with an element of the set and participates in two types of structural relationships (see Figure 4.9). Vertically, nodes corresponding to the same element are linked across consecutive levels, forming a tower. Horizontally,

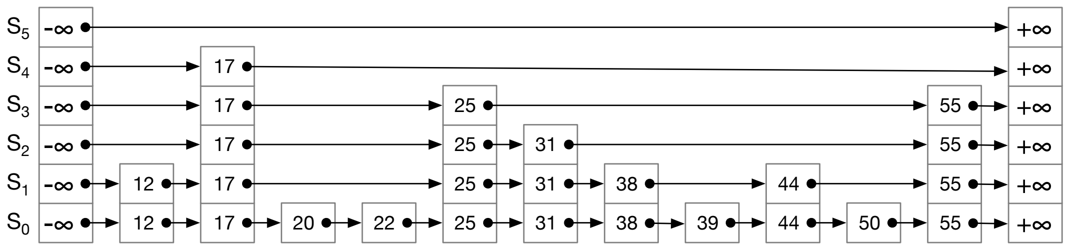


Figure 4.9. Example of a skiplist.

nodes within the same level are linked in sorted order, enabling traversal along that level. Boundary cases are handled using sentinel elements, which terminate vertical and horizontal traversals at the lowest and highest levels, respectively.

The basic skiplist operations are as follows:

- **Search**: To search for an element x, start from the topmost list at the leftmost node, move right until reaching a node with key greater than or equal to x. If equal, return it; otherwise, drop down one level and continue until reaching S_0.
- **Insert**: First, perform a search to locate the position where x should be inserted in S_0. Then, insert x at this position in S_0 and promote it to higher levels with probability p, creating a new node and inserting it in each level traversed during promotion.
- **Delete**: Perform a search to locate x. Then, remove its corresponding nodes from all levels where it appears, updating pointers from the previous nodes accordingly.

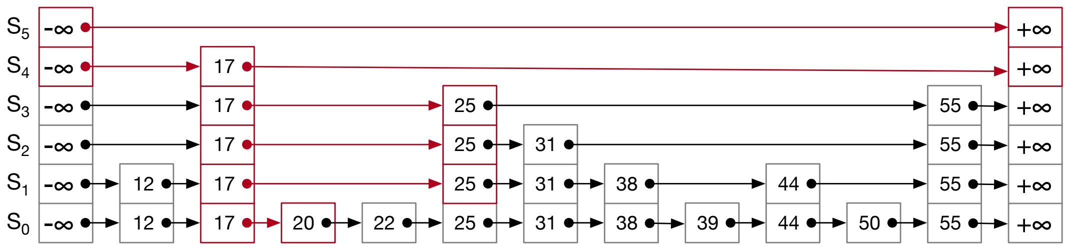


Figure 4.10. Search for element 17 in the skiplist of Figure 4.9. The nodes visited and the links traversed are highlighted in red.

In deterministic skiplists, the search procedure is guaranteed to run in $O(\log n)$ time. For randomized skiplists, it is well known [47] that the same procedure runs in expected $O(\log n)$ time. With high probability, the height t of the structure is $O(\log n)$, and the expected number of nodes visited per level is constant. Moreover, experimental studies [84] have shown that randomized skiplists outperform balanced tree structures such as 2–3 trees and red–black trees in practice. In our SVCSkipList implementation detailed later, we adopt a randomized skiplist design, as it offers better performance and simplicity compared to deterministic skiplists, while still providing logarithmic time complexity for search, insert, and delete operations.

Self-validating clustered skiplists A skiplist can incorporate a cryptographic, collision-resistant hashing scheme to enable self-validation (e.g., [46]). A collision-resistant hash function h ensures it is computationally infeasible to find two distinct inputs with the same output, thus guaranteeing data integrity. In our approach, a hash of each node is computed using function h recursively (see Figure 4.11(a)): if a node n is in the base level, its hash is computed directly as the digest of its data (key and value). Otherwise, n is an internal node, and its hash is computed as the digest of the concatenation of the hashes of all its immediate lower-level children, which are the lower-level nodes between n's key and its right neighbor's key. Therefore, the value $h(S_0)$ stored at the start node S_0 is called *root hash*, and serves as a digest representing the entire skiplist.

While hashing schemes are a robust mechanism for data structure self-validation, they fall short when it comes to supporting efficient state synchronization. In the design presented in [46], for instance, the entire skiplist is treated as a single entity for validation, meaning that any state synchronization involving this data structure would require transferring and validating the entire skiplist state. This approach, while secure, is inefficient when dealing with large states or when operating in environments prone to Byzantine faults.

We posit that, in the context of state machine replication (SMR), a self-validating clustered data structure must meet the following constraints:

1. **Deterministic Operations**: All procedures for manipulating the data structure and maintaining clusters must be deterministic, ensuring that operations are executed consistently across all replicas.

2. **Clustering**: The data structure must support a clustering scheme that groups data into subsets based on specific, deterministic criteria, ensuring efficient serialization/deserialization and transfer.

3. **Structural Integrity**: Each cluster must preserve the essential properties of the non-clustered version of the data structure, ensuring that fundamental operational characteristics, such as time and space complexity, are retained. This allows operations within a cluster to mirror those in the original layout, preserving correctness and efficiency.

4. **Self-Validation**: The data structure as a whole, and each of its clusters, must be self-verifiable, enabling independent cluster transmission and validation.

5. **Versioning**: In an SMR system, replicas may be at different states due to network delays or different processing speeds. The data structure must incorporate versioning to ensure consistency, with periodic checkpoints allowing replicas to respond with the same state, regardless of local progress.

Such data structures enable clusters to be transferred in parallel and across multiple replicas during a state synchronization. Moreover, if a Byzantine fault is detected in one cluster, only that specific cluster needs to be retransmitted, rather than the entire data structure. Upon receiving and validating all necessary clusters, the receiving replica must be able to reconstruct the complete data structure deterministically. We ensure that our SVCSkipList design adheres to these constraints as we detail in the following sections.

Clustering strategy We discuss two mechanisms for clustering a skiplist: *static partitioning* and *dynamic partitioning*. In the following sections, we examine the key properties of each approach, highlighting their respective advantages and disadvantages, and discussing how they impact performance, scalability, and the amount of space required to store and manage the structure (space efficiency).

Static partitioning: A common clustering strategy is *static* or *range partitioning*. In this approach a skiplist can be divided into fixed, independently managed segments. Each cluster is a self-validating independent skiplist with its own root hash. To represent the entire structure, cluster root hashes can be aggregated into a single root hash using a binary hash tree: cluster hashes form the leaves, and parent nodes recursively hash pairs of children up to the root. However, a notable drawback of this approach is its sensitivity to workload distribution [70]. It performs well under uniformly distributed keys, resulting in balanced clusters and efficient space usage. In contrast, skewed workloads can cause uneven cluster sizes and data distribution, resulting in reduced space efficiency and potentially degrading overall performance.

Dynamic partitioning: To address the limitations associated with static partitioning techniques, we explore the concept of *dynamic partitioning*. In this approach, clusters are defined by selecting a level $l \geq 1$ such that the elements at level l define the cluster roots. This technique partitions the skiplist into multiple clusters, each bounded by elements at level l, as illustrated in Figure 4.11. Consequently, the set of nodes at level l serves as the list of cluster roots, effectively defining the set of clusters. The number of nodes at this level directly corresponds to the total number of clusters in the skiplist. This approach also enables the application to specify the approximate desired cluster size. Selecting higher levels results in larger clusters (e.g., Figure 4.11(a)), while

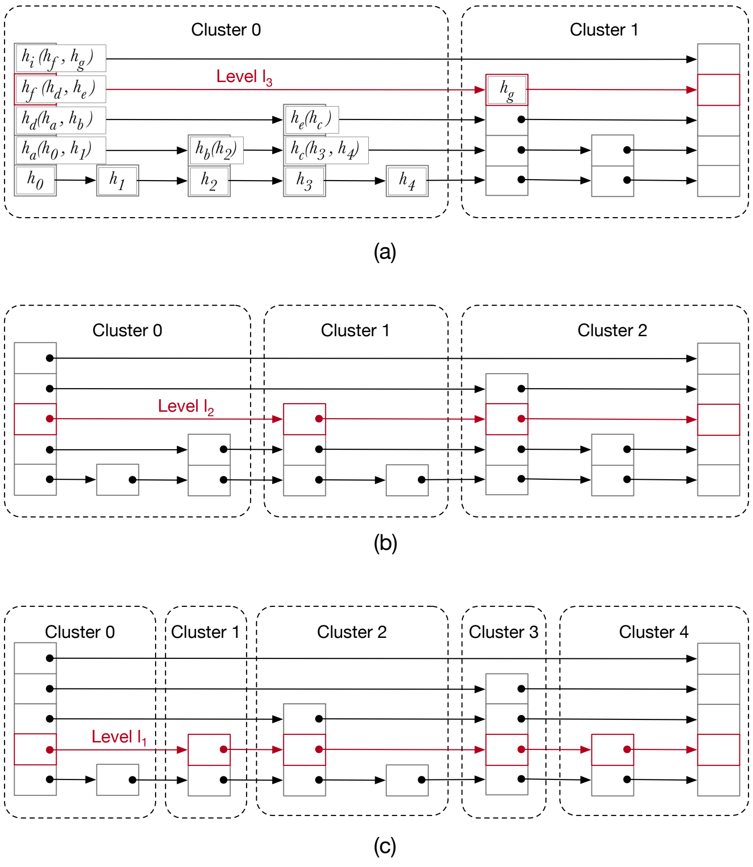


Figure 4.11. Dynamically partitioned clustered skiplists. In (a), we use cluster 0 to illustrate how the root node's hash (h_i) is computed as the hash of the concatenated hashes of its children. This process applies to each cluster and is recursively performed throughout the entire skiplist.

choosing lower levels yields smaller clusters (e.g., Figure 4.11(c)). It also supports self-validation by computing a hash for each cluster root, allowing for independent cluster validation.

Moreover, dynamic clustering allows clusters to reflect the actual data distribution rather than fixed sizes, improving performance, especially when some keys are accessed more frequently. This flexibility enhances retrieval, validation, and adaptability to varying workloads and evolving datasets.

Versioning strategy To implement versioning, copy-on-write can be used by augmenting each skiplist node with metadata indicating its creation or last modification version. The skiplist evolves with its version number incremented periodically, as detailed later with replica checkpointing. When a replica at version v modifies the skiplist and the affected nodes are from an older version $v' < v$, it creates new copies of these nodes (or references) labeled with v, preserving previous versions. This ensures deterministic state synchronization since replicas may progress at different rates and hold different views of the state. By specifying the desired version during synchronization, a replica obtains a consistent state, while source replicas use version metadata to identify and transmit only elements belonging to the requested version, ensuring correctness and consistency.

4.3.2 SVCSkipList in detail

We now introduce SVCSkipList, our Self-Validating Clustered SkipList with a hashing scheme and a clustering mechanism for independent transfer and validation of skiplist segments. These improvements address the inefficiencies of full-state transfer by enabling smaller, verifiable portions of the state to be processed separately. We present the main aspects, implementation, and some of the algorithms behind the SVCSkipList. While static partitioning offers simplicity, its limitations under skewed workloads led us to adopt a dynamic partitioning scheme that better adapts to key distributions and ensures balanced, efficient clustering.

Preliminar definitions Algorithm 6 defines the core components of the SVCSkipList. The global variables track system-wide parameters, such as the current version v, cluster level l (i.e., the level that defines cluster boundaries), node-level generation probability p, and seed s. Our system requires that the probability p and seed s be shared among all replicas to preserve SMR determinism. By using the same probability and seed to generate node levels, all replicas construct identical skiplist structures for the same sequence of operations. This guarantees that the data structure remains consistent across replicas, maintaining the correctness of the replicated state. The structure evolves over versions i, with arrays used to track different versions of the head and tail sentinels (i.e., *head*$[i]$, *tail*$[i]$), as well as the height $h[i]$ and size $n[i]$ (i.e., number of elements) of each version. The *head*$[i]$ node is the global head of the SVCSkipList at version i, storing the structure's *root hash* for that version, which is essential for self-validating both the entire skiplist and its individual clusters.

Elements in the SVCSkipList are encapsulated within self-verifiable versioned nodes (SVNode), which maintain integrity across versions through version-aware links. Each SVNode holds a key k, value val, and version v, along with a hash used for verification. Nodes use arrays to track links at different versions, with directional pointers (i.e., *left*$[i]$, *right*$[i]$, *up*$[i]$, *down*$[i]$) for each version i, and are assigned a level ℓ. A deletion flag del is used to indicate the logical removal of nodes. Physical deletion is

Algorithm 6 SVCSkipList General Definitions

```
1: Global variables:
2:     $v : \mathbb{N}$                                  {Current version}
3:     $p : \mathbb{R}$                                  {Probability for level generation}
4:     $s : \mathbb{Z}$                                  {Seed for deterministic level generation}
5:     $l : \mathbb{N}$                                  {Cluster level to define cluster boundaries}
6:     $\forall i : head[i] : \text{Array}\langle\text{SVNode}\rangle$    {Head node at version $i$}
7:     $\forall i : tail[i] : \text{Array}\langle\text{SVNode}\rangle$    {Tail node at version $i$}
8:     $\forall i : h[i] : \text{Array}\langle\mathbb{N}\rangle$          {Skiplist height at version $i$}
9:     $\forall i : n[i] : \text{Array}\langle\mathbb{N}\rangle$          {Number of elements at version $i$}
10: Each SVNode has:
11:     $k : \mathbb{Z}$                                  {Key}
12:     $val : \text{Bytes}$                              {Value}
13:     $v : \mathbb{N}$                                  {Version of the node}
14:     $hash : \text{Bytes}$                             {Hash of this node}
15:     $\forall i : left[i], right[i] : \text{Array}\langle\text{SVNode}\rangle$   {Directional links at}
16:     $\forall i : up[i], down[i] : \text{Array}\langle\text{SVNode}\rangle$       {Version $i$}
17:     $\ell : \mathbb{N}$                               {Level of the node}
18:     $del : \text{Bool}$                               {Deletion marker (true if logically deleted)}
19: Each Cluster has:
20:     $id : \mathbb{N}$                                 {Unique cluster identifier}
21:     $n_b : \mathbb{N}$                                {Number of base-level nodes}
22:     $k_r : \mathbb{Z}$                                {Right key delimiter}
23:     $f : \text{SVNode}$                               {First node in the cluster}
24:     $\pi : \text{List}\langle\text{Hash}\rangle$      {Proof path for authentication}
```

not a critical feature in evaluating our data structure and was not implemented in our prototype. Instead, deletions are implemented by marking nodes as logically deleted, with physical removal and garbage collection left as future enhancements.

Each Cluster encapsulates a contiguous segment of the skiplist at the base level and contains structural and verification metadata. However, a physical Cluster object is instantiated only upon a state synchronization request, following the procedure detailed later in Algorithm 8. When a cluster object is instantiated, the field id uniquely identifies the cluster and corresponds to its index within the list of clusters determined by the cluster level l. The field n_b records the number of base-level nodes contained in the cluster. The delimiter k_r marks the rightmost boundary key, which defines the cluster's key range. The pointer f references the first base-level node of the cluster. Finally, π stores the authentication proof, used during verification to ensure the integrity of the cluster independently of the rest of the structure.

The basic operations to *search*, *insert*, and (logically) *delete* elements in the skiplist are implemented based on the classic skiplist algorithms [84], extended to support versioning. The data structure includes a function to increment the current version, enabling the tracking of state changes over time. Replicas produce new versions synchronously at corresponding state points by applying identical sets of commands, this ensures consistency across the system. The *insert* operation leverages the previously defined parameters, such as the probability p and seed s, to deterministically assign levels to new nodes. If the given key already exists, its associated value is updated. During insertion, a local hash is also computed by serializing the key and value, ensuring element-level integrity. Additionally, the *insert* operation adopts the copy-on-write strategy described earlier for versioning: when a node is created or modified in a new version, such as when updating its value after a version increment, a new node object is created for the new version holding the new value, while the original node object with the previous value is preserved in its previous version. This approach guarantees that each version maintains a consistent and immutable snapshot of the skiplist state at the time of creation. Such immutability will be essential later during state synchronization.

In addition, the SVCSkipList provides a recursive function to compute the hashes of nodes, enabling structural verification. This function can be invoked to compute the hash of the entire skiplist or selectively over individual clusters, as part of the validation process described in the following sections.

Self-validation As shown in Algorithm 7, we define the `computeHashes` function to ensure verifiability across versions by computing cryptographic hashes over the versioned skiplist. The function initiates hash computation from the head node of version i and invokes `compHashRec`, which recursively traverses the skiplist in a top-down manner. For internal nodes, `compHashRec` collects the hashes of child nodes within a bounded key range, concatenates them, and computes a hash of the combined value (as demonstrated in Figure 4.11(a)). For leaf nodes, the hash is derived directly from the serialized key and value. This recursive procedure can be used either to compute hashes for the entire skiplist starting from the global head or to validate individual clusters by initiating the recursion from a cluster's head node. This unified approach enables efficient and consistent verification across both global and cluster scopes.

Clustering To enable scalable state synchronization and modular verification, the SVCSkipList is logically partitioned into *clusters* with contiguous segments of base-level nodes grouped based on structural and versioning boundaries.

Algorithm 8 details a procedure to retrieve clusters from the skiplist at a specific version v within a specified index range $[r_s, r_e]$. This range corresponds to the indices of clusters within the list of clusters defined at level l. The need for this range is explained in the next section. The main function, `getClusters`, iterates over the range,

Algorithm 7 Recursive Hash Computation

```
1: Func computeHashes($i : \mathbb{N}$):
2:   $n \leftarrow head[i]$                                  {Get head node at version $i$}
3:   compHashRec($n, i$)                                     {Recursive computation}
4: Func compHashRec($n$ : SVNode, $i : \mathbb{N}$) → Bytes:
5:   if $n.down[i] \neq \varnothing$ then                    {If n in version i is an inner node}
6:     if $n.right[i] = \varnothing$ then                    {Bound-key for children}
7:       $r \leftarrow +\infty$
8:     else
9:       $r \leftarrow n.right[i].k$
10:    $aux \leftarrow n.down[i]$                            {Start from leftmost child}
11:    $L \leftarrow []$                                     {List to hold child hashes}
12:    while $aux \neq \varnothing \wedge aux.k < r$ do      {Iterate children in range}
13:      $L$.append.(compHashRec($aux, i$))                  {Recursive call}
14:      $aux \leftarrow aux.right[i]$                       {Move to next sibling}
15:    $n.hash \leftarrow \mathrm{H}(\bigoplus L)$           {Hash concat. child hashes}
16:  else
17:    $n.hash \leftarrow \mathrm{H}(n.k, n.val)$            {Hash leaf node}
18:  return $n.hash$
```

invoking getCluster for each cluster index. The getCluster function instantiates a cluster for the specified version v by first locating the starting node of the cluster using findClusterStart, which retrieves the cluster root node from the list of clusters defined by level l. It updates the right key delimiter k_r which marks the cluster boundary at version v, and generates a cryptographic proof π to authenticate the cluster's content relative to the skiplist head at that version. The function proceeds to identify the first base-level node within the cluster through findBaseNode. Then it counts the total number of base-level nodes n_b up to the right delimiter, all at version v. Finally, the fully initialized cluster is returned.

Each Cluster object has a dedicated function genProof to compute a compact authentication proof π, which is a list of hashes, including the minimal set of hashes from upper levels necessary to authenticate the cluster's integrity and the SVCSkipList's *root hash* (from $head[v]$) for the given version v. Starting from the cluster's delimiter node, it ascends the skiplist hierarchy, identifying at each level the necessary sibling nodes that contribute to the hash verification path.

Each Cluster also supports serialization and reassembly, allowing conversion to and from a compact representation suitable for transmission. Furthermore, each cluster can validate its contents by using the same functions defined in Algorithm 7 for recomputing local hashes and verifying them against the expected *root hash* along the proof path.

Algorithm 8 Retrieve Clusters in Version v

```
 1: Func getClusters($v : \mathbb{N}, r_s : \mathbb{N}, r_e : \mathbb{N}$) → List⟨Cluster⟩:
 2:     $C \leftarrow []$                                    {List of clusters}
 3:     for $i \leftarrow r_s$ to $r_e$ do
 4:         $c \leftarrow$ getCluster($i, v$)                {Get cluster at index i and version v}
 5:         $C$.append($c$)
 6:     return $C$

 7: Func getCluster($idx : \mathbb{N}, v : \mathbb{N}$) → Cluster:
 8:     $c \leftarrow$ Cluster($idx$)                        {Create cluster with given id}
 9:     $n \leftarrow$ findClusterStart($head[v], l, idx$)   {Find cluster head}
10:     $c.k_r \leftarrow n.right[v].k$                      {Right key delimiter of cluster}
11:     $c.\pi \leftarrow$ genProof($n, head[v], v$)         {Generate auth. proof}
12:     $c.f \leftarrow$ findBaseNode($n, v$)                {Get first base-level node in cluster}
13:     $c.n_b \leftarrow$ countNodes($n, v, c.k_r$)         {Count base-level nodes}
14:     return $c$
```

State synchronization with SVCSkipList Our SVCSkipList enhances the state synchronization protocol of SMR libraries by enabling efficient, versioned, and authenticated cluster-based transfers. The process (detailed in Algorithm 9) begins when a replica requests a state synchronization for a specific version v. The source replica calculates its range of clusters, based on its replica id and the total number of clusters in version v, retrieves them using the cluster extraction mechanism (Algorithm 8), and serializes each cluster's structure, metadata, and validation data π, into a compact format. These serialized clusters are transmitted via specialized state synchronization channels in the SMR framework, extended to support our approach.

Upon reception, the requesting replica deserializes the received data to reconstruct each cluster and validates it using the procedure described in Algorithm 7. Specifically, it computes the cluster hashes using the proof structure and validates them against the SVCSkipList *root hash* provided in π, ensuring correctness and preventing inconsistencies. Validated clusters are accumulated in a set of received clusters. The replica continues to refetch any invalid clusters until all expected clusters are successfully received and verified. Once the complete set of clusters is available and validated, the replica reconstructs the SVCSkipList by integrating all clusters and restoring their structural connections.

4.3.3 Integration into BFT-SMaRt

We integrated our SVCSkipList into the BFT-SMaRt framework [12]. This is justified by BFT-SMaRt's modular approach, which features a well-defined state management module, a Java-based implementation, and a multicore-aware threaded architecture

Algorithm 9 State Synchronization Procedure

```
1: Global variables for replica $r$:
2:     $r.receivedClusters$ : Set⟨Cluster⟩              {Set of received valid clusters}
3:     $r.totalClusters$ : $\mathbb{N}$                 {Expected total number of clusters}
4:     $r.id$ : $\mathbb{N}$                            {Replica id}
5: Func stateSyncRequest($v$ : $\mathbb{N}$):
6:     sendSyncRequest($v$)                             {Replica requests state sync for version $v$}
7: Func processSyncRequest($v$ : $\mathbb{N}$):          {Source replica receives req.}
8:     $(r_s, r_e) \leftarrow$ clusterRange($v, r.id$)   {Calc. cluster index range}
9:     $C \leftarrow$ getClusters($v, r_s, r_e$)         {Retrieve clusters (Algorithm 8)}
10:    $\sigma \leftarrow$ serialize($C$)                {Serialize clusters and validation info $\pi$}
11:    sendClusters($\sigma, v$)                        {Send clusters to requesting replica}
12: Func receiveClusters($\sigma$ : Bytes, $v$ : $\mathbb{N}$):
13:    $C \leftarrow$ deserialize($\sigma$)              {Deserialize clusters and validation info $\pi$}
14:    for all $c \in C$ do
15:        if validateCluster($c, v$) then              {validate each cluster separately}
16:            $receivedClusters$.add($c$)              {Add valid cluster to set}
17:        else
18:            refetchCluster($c.id, v$)                {Refetch invalid cluster}
19:    if $|receivedClusters| = totalClusters$ then     {Rebuild skiplist when}
20:        rebuildSkiplist($receivedClusters, v$)       {all clusters received/valid}
```

that facilitates high-performance Byzantine fault-tolerant replication. To achieve this integration, we made several modifications to the existing codebase. In this section, we describe the key changes.

Data structure implementation We integrated our implementation of the SVCSkipList into the BFT-SMaRt library, exposing state manipulation operations to the application layer. Our approach also performs checkpoints; however, while BFT-SMaRt's baseline approach creates a snapshot by serializing and copying the entire state, checkpointing in our approach means creating a new version by simply incrementing the version number and recomputing hashes for the previous version. This design not only reduces checkpointing overhead, but also offloads application state management to the replication library itself, relieving developers from manually tracking or serializing the state.

State transfer protocol We modified the state transfer protocol to leverage the clustered nature of the SVCSkipList. This allows replicas to collaboratively and concurrently transfer separate ranges of clusters instead of the entire state, accelerating state synchronization. Our implementation uses the procedure described in Algorithm 9.

Additionally, we extended the BFT-SMaRt framework to support state transmission over a dedicated socket, decoupling state management communication from the consensus and replica coordination channels. This enhancement benefits both the baseline BFT-SMaRt standard state management module and our new module based on the SVCSkipList. We also addressed a limitation in BFT-SMaRt's baseline implementation that restricted state sizes to approximately 2 GB due to Java's maximum byte array size. By partitioning checkpoints into multiple byte arrays instead of a single one, it now supports arbitrarily large state sizes.

Validation mechanism We implemented a mechanism that enables replicas to verify the integrity of received clusters independently, enhancing fault tolerance and reducing reliance on consensus for validation. An SMR framework can be extended so that replicas agree during consensus on the *root hash* of each version of the SVCSkipList. This root hash can then be securely shared among replicas as part of the state synchronization protocol. Alternatively, a replica may initiate state synchronization by requesting the root hash from other replicas and waiting to receive $f+1$ matching values, thereby ensuring consistency without modifying the consensus protocol. During state synchronization, clusters are validated incrementally upon arrival, as detailed in Algorithm 9. This approach contrasts with the baseline BFT-SMaRt mechanism, which performs validation only after the entire state has been fully downloaded, necessitating a complete re-fetch if corruption is detected.

Configuration options We added configuration options to allow users to specify parameters such as the SVCSkipList's cluster level, promotion probability, and seed. We adopted this static configuration approach for simplicity; however, designing a lightweight protocol for replicas to agree on a common seed—or even periodically rotate it—is straightforward. Similarly, the ability to support dynamic reconfiguration of the cluster level, adapting cluster sizes on the fly according to application demands, is a powerful capability enabled by our data structure.

These modifications were designed to ensure our SVCSkipList could be seamlessly integrated into the existing BFT-SMaRt framework while providing enhanced performance and fault tolerance through its self-validating clustered state management capabilities.

4.3.4 SVCSkipList evaluation

Rationale We compare the performance of our SVCSkipList to a baseline (i.e., BFT-SMaRt) technique in typical SMR deployments: replicas with independent failure

Workload (key-value store):	RM (read-most): 80% reads, 20% updates UH (update-heavy): 50% reads, 50% updates
Number of clients:	200, 400, ... (passed point of highest power, see Section 4.3.4)
Number of replicas:	4 and 7 replicas (1 and 2 malicious, respectively)
State size:	1 GB, 2 GB, and 4 GB
Number of clusters:	32749 (level 5), ..., 2053 (level 9), ..., 24 (level 15)
Checkpoint frequency:	each 2000 consensus instances, where each instance can contain up to 1024 operations

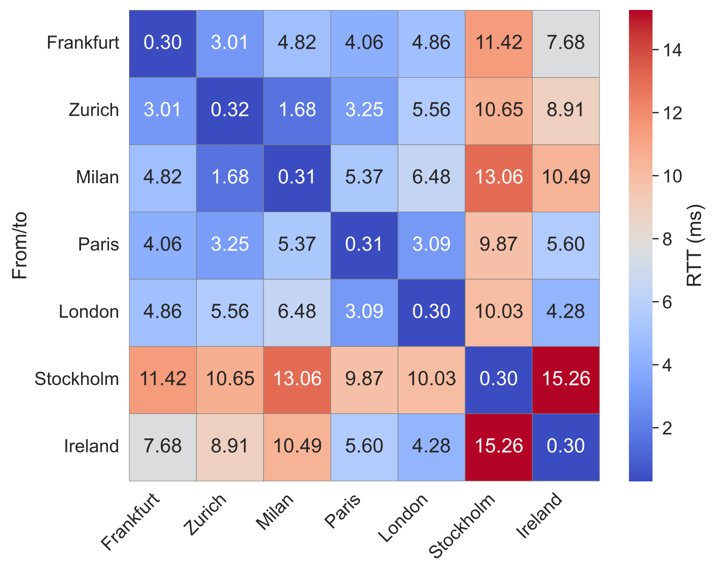
Table 4.3. Parameters used in the evaluation.

modes (e.g., different availability zones [6]) within the same geographical region. The experimental evaluation seeks to answer the following questions:

1. How do the techniques compare under normal conditions (i.e., request execution without state synchronization)?
2. What's the cost of checkpointing in each approach?
3. How long does it take to synchronize state without attacks?
4. How does cluster size impact state synchronization?
5. How does state synchronization impact execution?
6. How long does it take to synchronize state under attacks?

We aim to understand how techniques compare under various workloads, state sizes, replica numbers, and cluster sizes. Table 4.3 summarizes the space of parameters considered in our evaluation.

Preliminaries Before presenting the results of our evaluation, we describe the experimental setup used to assess the performance and resilience of our approach. Here we detail the hardware environment, the emulated public cloud configuration, the application, and the setup of clients and replicas. We also outline key execution parameters



From/to	Frankfurt	Zurich	Milan	Paris	London	Stockholm	Ireland
Frankfurt	0.30	3.01	4.82	4.06	4.86	11.42	7.68
Zurich	3.01	0.32	1.68	3.25	5.56	10.65	8.91
Milan	4.82	1.68	0.31	5.37	6.48	13.06	10.49
Paris	4.06	3.25	5.37	0.31	3.09	9.87	5.60
London	4.86	5.56	6.48	3.09	0.30	10.03	4.28
Stockholm	11.42	10.65	13.06	9.87	10.03	0.30	15.26
Ireland	7.68	8.91	10.49	5.60	4.28	15.26	0.30

Figure 4.12. Average round-trip time (RTT in milliseconds) between pairs of availability zones used in our emulated public cloud environment.

and specific configurations of the BFT-SMaRt framework to ensure repeatability and relevance to realistic deployment scenarios.

Hardware All experiments were conducted in a private cluster with machines running Ubuntu 18.04.6 LTS with 32 GB of RAM and an AMD EPYC 7282 16-core processor, comprising 32 hardware threads (16 cores with 2 threads per core). The CPU operates in 64-bit mode and supports a maximum frequency of 2.8 GHz.

Emulated public cloud In this environment, we emulated the bandwidth and latency of Amazon EC2 M5 instances (`m5.large`), a standard and cost-effective option for practical setups, deployed in seven availability zones in Europe [6]. The maximum network bandwidth between two machines was configured to 1 Gbps. The average latency patterns between machines are shown in the latency heat map in Figure 4.12[3] and configured with 5% jitter.

Application We implemented a simple key–value store inspired by YCSB [25], using the BFT-SMaRt server-side application interface. Keys are integers and values are random byte arrays of size 1 KB. To control the initial state size, the skiplist is pre-populated with a sufficient number of key–value pairs to reach the desired state size, ranging from 1 GB to 4 GB depending on the experimental scenario. Keys are inserted sequentially in increasing order until the target state size is reached. During

[3]https://www.cloudping.co/

the experiments, clients access keys following a uniform distribution.

The same application was used for the SVCSkipList and the baseline setups to ensure a fair comparison. The key difference lies in state management: the baseline uses BFT-SMaRt's default mechanism, meaning the skiplist is managed entirely by the application and fully serialized/deserialized during checkpointing and state installation. In contrast, our approach integrates the SVCSkipList directly into a newly designed state management module (Section 4.3.3), leveraging its clustered and self-validating features.

Clients We implemented client processes using BFT-SMaRt's client proxy interface to generate a mixed workload of reads and writes. Clients operate in a closed-loop fashion, issuing each new request only after receiving a response to the previous one. All clients and replicas were uniformly distributed across the emulated zones described earlier.

Replicas Each replica pre-initializes its SVCSkipList deterministically, ensuring the system starts with a consistent initial state. As already mentioned, we consider state sizes of 1 GB, 2 GB, and 4 GB, where a proportional number of 1 KB elements were inserted into the skiplist during initialization. Experiments were conducted using server configurations with either four replicas (capable of tolerating up to one Byzantine failure) or seven replicas (capable of tolerating up to 2 Byzantine failures). The 4-replica experiments were deployed in Frankfurt, Zurich, Milan, and Paris (see Figure 4.12), while the 7-replica experiments spanned all zones under consideration. To simulate Byzantine faults, one or two replicas were deliberately configured to corrupt data during state synchronization, thereby emulating malicious behavior.

Executions Experimental executions ranged from 1 to 3 minutes, depending on the specific scenario. Initial warm-up data and final termination periods were discarded to ensure that only steady-state performance was analyzed.

BFT-SMaRt *parameters* The BFT-SMaRt framework was configured to operate in Byzantine fault tolerance mode, with both the log and checkpoint mechanisms using in-memory storage instead of disk persistence to reduce I/O overhead during experimentation. Most configuration parameters were kept at their default values to preserve standard behavior; however, some were tuned to enable support for large application states. In particular, the checkpoint frequency was adjusted based on the size of the state being tested, ensuring efficient operation and avoiding excessive memory consumption or checkpoint delays during the experiments.

Performance under normal conditions To evaluate system performance under normal conditions, we varied the number of concurrent clients issuing operations to the replicated service, aiming to identify saturation points and assess how our structure handles high contention. We measured throughput and latency with a 1GB application state, comparing SVCSkipList to the original BFT-SMaRt baseline under read-most (RM) and update-heavy (UH) workloads.

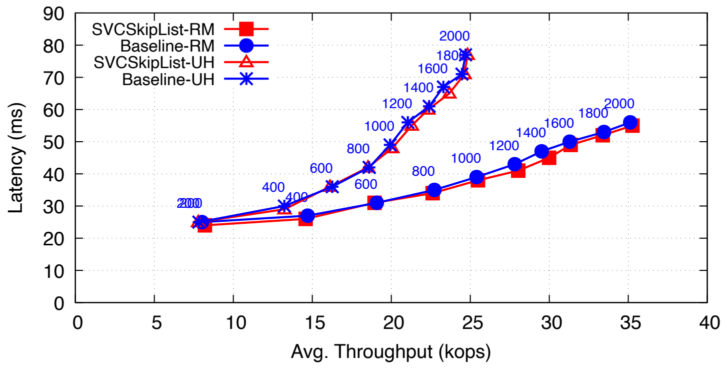


Figure 4.13. Throughput versus latency across different workloads and client counts for both SVCSkipList and the Baseline approach.

Figure 4.13 presents the throughput (in kops, kilo operations per second) and average latency (in milliseconds). As the number of clients grows from 200 to 2000, both techniques scale effectively, showing a consistent increase in throughput. For instance, in the RM workload the SVCSkipList achieves performance comparable to the Baseline, with differences typically under 2% even at higher loads. These results indicate that our approach maintains performance comparable to the Baseline across workloads, with minor advantages due to reduced checkpointing overhead, which helps lower the average processing cost per request.

Based on this data, we calculate the points of highest power in the system, that is, where the ratio of throughput divided by latency is at its maximum. This ensures that our evaluation reflects the operational point with the best efficiency in terms of processing cost per request. Although this does not represent the absolute maximum throughput of the system, it indicates the inflection point where the system reaches its peak effective throughput before latencies start to increase significantly due to queueing effects. Hence, the points used for all subsequent experiments were chosen as follows: for the RM workload, 1200 clients (where throughput was around 28 kops with average latency of 41–43 ms), and for the UH workload, 600 clients (where throughput was around 16 kops with average latency of 36 ms).

Impact of workload Table 4.4 presents the throughput and latency for both workloads using a 1GB state, during normal execution. The results show that for the RM workload, the SVCSkipList achieves a slightly lower average latency (41.9 ms) compared to the Baseline (43.0 ms), with similar median latencies; however, it exhibits a notable improvement at higher percentiles. Specifically, while both systems have

				Throughput (kops)	**Latencies** (ms)				
WL	**R#**	**Algorithm**	**State**	Avg.	Avg.	50^{th}	99^{th}	99.9^{th}	99.99^{th}
RM	4	SVCSkipList	1GB	26.0	41.9	39.0	81.0	646.0	1016.0
	4	Baseline	1GB	25.0	43.0	40.0	67.0	1481.0	1820.0
	4	SVCSkipList	2GB	25.5	43.6	40.0	67.0	1335.0	1788.0
	4	Baseline	2GB	24.2	47.4	41.0	70.0	2872.0	3450.0
	4	SVCSkipList	4GB	23.1	44.7	39.0	68.0	2440.0	3710.0
	4	Baseline	4GB	16.9	60.4	41.0	309.0	6403.0	6419.0
UH	4	SVCSkipList	1GB	16.1	36.2	35.0	54.0	701.0	1129.0
	4	Baseline	1GB	16.2	36.6	35.0	53.0	1489.0	1828.0
RM	7	SVCSkipList	1GB	9.0	132.5	129.6	198.3	849.3	929.3
	7	Baseline	1GB	8.8	135.0	130.6	197.3	1538.3	1901.3
	7	SVCSkipList	2GB	8.8	134.0	129.6	204.0	1604.3	1698.3
	7	Baseline	2GB	8.4	144.2	129.6	229.0	3486.3	3562.0

Table 4.4. Throughput and latency during normal execution in various configurations at highest power (i.e., max throughput over latency ratio). Best throughput and latency for each configuration in blue. SVCSkipList outperforms Baseline in almost all cases.

comparable 99th percentile latencies (81.0 ms vs. 67.0 ms), the SVCSkipList exhibits substantially lower tail latencies at the 99.9th and 99.99th percentiles (646.0 ms and 1016.0 ms) compared to the Baseline (1481.0 ms and 1820.0 ms), due to its lower checkpointing overhead (see Figure 4.14). In the UH workload, the average and median latencies are nearly identical between the two approaches; however, the SVCSkipList again significantly reduces tail latencies. At the 99.9th percentile, SVCSkipList reports 701.0 ms versus 1489.0 ms for the Baseline, and at the 99.99th percentile, 1129.0 ms compared to 1828.0 ms.

Impact of state size Table 4.4 also shows the impact of increasing state size on system performance during normal execution. As the application state grows from 1 GB to 4 GB, with four replicas and the RM workload, the throughput of both the SVCSkipList and the Baseline techniques gradually decreases. However, the decline is notably steeper for the Baseline: while the SVCSkipList sustains throughput levels close to 26 kops up to 2 GB and only drops to around 23 kops at 4GB, the Baseline throughput drops from 25 kops at 1GB to just 16.9 kops at 4 GB. This suggests that our approach handles larger states more efficiently, again due to its lower checkpointing overhead.

Similarly, latency results show that the SVCSkipList maintains more stable average and median latencies as the state size increases. For example, the Baseline's average

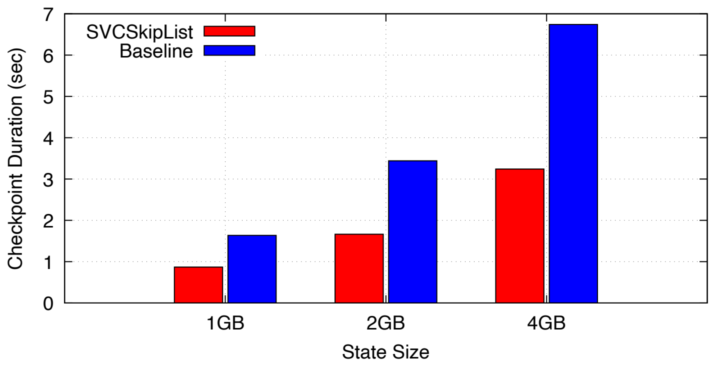


Figure 4.14. Average time to create a checkpoint for both techniques under the RM workload.

latency increases from 43.0 ms to 60.4 ms as state size grows from 1GB to 4GB, while the SVCSkipList grows only modestly from 41.9 ms to 44.7 ms. Furthermore, the tail latencies of the Baseline degrade as state size grows, while the SVCSkipList keeps them significantly lower. Overall, these results indicate that our structure scales better with state size, offering improved stability in both throughput and worst-case latency as the system state grows.

Impact of checkpointing Figure 4.14 presents the checkpointing times for both techniques across different state sizes, with four replicas and the RM workload, reported in seconds. The results demonstrate that SVCSkipList multiversioning scheme consistently outperforms Baseline checkpoints. For instance, with a 1GB state, SVCSkipList takes approximately 0.87 seconds versus 1.64 seconds for the Baseline, almost halving the time. As state size increases, the benefits become more evident: for 2GB, SVCSkipList takes 1.66 seconds compared to 3.44 seconds for the Baseline, while for 4GB, checkpointing time is 3.24 seconds against 6.74 seconds. This reduction stems from the more efficient copy-on-write multiversioning approach used in SVCSkipList, which avoids data copying overhead during checkpoint creation. It only increments the version number and recomputes hashes, rather than serializing the entire state at once as in the Baseline approach.

State synchronization without attack We now evaluate the state synchronization times for the SVCSkipList and Baseline techniques, measured as the time taken to synchronize the full application state across replicas in a benign (attack-free) setting. Since the replica executing state synchronization here is one that crashed and is returning to the system, we may refer to it as the *recovering* replica.

For each configuration, we ran three independent experiments to collect data for

		No Attack			Under Attack		
R#	**Algorithm**	**Avg.**	**Min**	**Max**	**Avg.**	**Min**	**Max**
4	SVCSkipList	36.3	35.8	37.3	36.1	35.5	36.6
4	Baseline	39.3	37.9	41.8	51.5	48.9	56.3
7	SVCSkipList	35.7	31.3	39.5	37.3	32.7	40.2
7	Baseline	44.2	39.2	51.7	103.2	90.0	121.0

Table 4.5. State synchronization time (sec) with and without attacks. Best results for each configuration in blue.

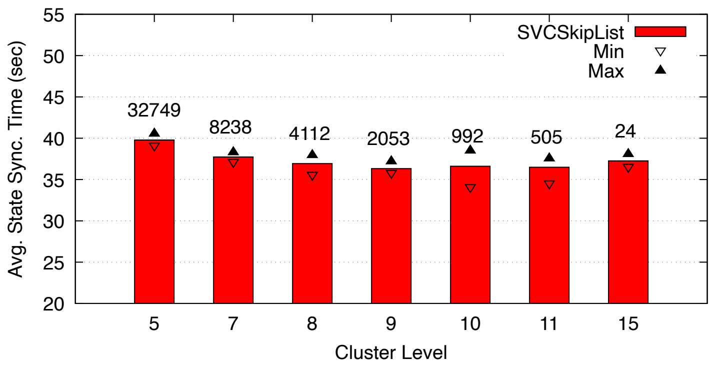


Figure 4.15. State synchronization time for SVCSkipList with 4 replicas under the RM workload, shown across different cluster levels (x-axis) and cluster sizes (values above bars).

computing average, minimum, and maximum values. Table 4.5 shows two group sizes: 4 replicas and 7 replicas. For the 4-replica configuration, SVCSkipList achieved an average synchronization time of approximately 36.3 seconds, while the Baseline required around 39.3 seconds, with the Baseline's times ranging from 37.9 to 41.8 seconds and SVCSkipList showing more consistent performance between 35.8 and 37.3 seconds. For the 7-replica configuration, the advantage of SVCSkipList became more evident, with an average time of 35.7 seconds compared to 44.2 seconds for the Baseline, which also exhibited higher variability. These results indicate that our optimized state transfer mechanism reduces synchronization time and improves consistency across runs, particularly as system scale increases.

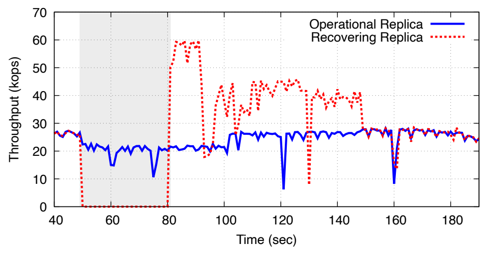
We also evaluate the impact of varying the number of clusters on state synchronization time for SVCSkipList, in the same setting. Results in Figure 4.15 indicate a non-linear relationship between cluster level and synchronization time. For example, with cluster level 5 (32,749 clusters), the average synchronization time was approx-

				Throughput (kops)	Latencies (ms)				
WL	**R#**	**Algorithm**	**State**	Avg.	Avg.	50th	99th	99.9th	99.99th
RM	4	SVCSkipList	1GB	20.3	44.0	43.0	81.0	315.8	856.5
	4	Baseline	1GB	20.1	46.2	44.0	85.3	380.3	1507.1
	4	SVCSkipList	2GB	21.2	50.2	46.3	152.0	981.3	1985.4
	4	Baseline	2GB	21.2	54.5	51.0	143.0	766.2	2677.4
	7	SVCSkipList	1GB	8.1	145.2	144.3	163.6	733.3	915.7
	7	Baseline	1GB	8.0	155.5	147.0	231.3	1301.3	1531.4
	7	SVCSkipList	2GB	8.0	145.5	144.6	174.3	934.1	1415.8
	7	Baseline	2GB	8.0	155.9	146.0	162.0	1018.8	1104.4

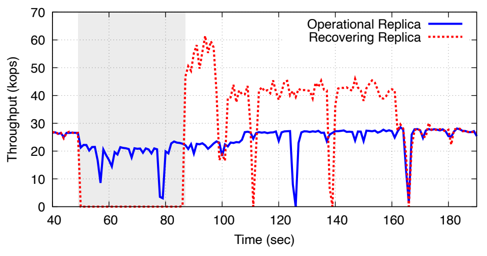
Table 4.6. Throughput and latency during state synchronization in various configurations at highest power (i.e., max throughput over latency ratio). Best throughput and latency for each configuration in blue. SVCSkipList outperforms Baseline in almost all cases.

imately 39.8 seconds, whereas level 9 (2,053 clusters) achieved the lowest average time of around 36.3 seconds. Increasing the cluster level beyond this point did not yield further improvements: level 15 (only 24 clusters) increased slightly to 37.3 seconds. These findings suggest that moderate cluster granularity (such as level 9) offers an optimal balance between metadata overhead and per-cluster transfer efficiency, resulting in reduced synchronization times. Hence, the next experiments adopt level 9 as the configuration for our clustering strategy.

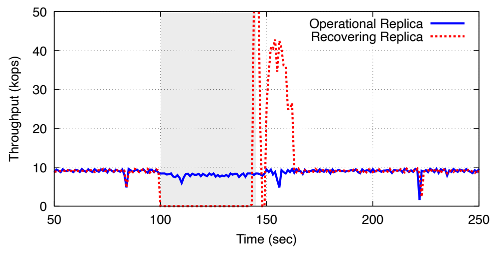
Figures 4.16a and 4.16b illustrate the impact of state synchronization on system throughput for both the SVCSkipList and Baseline techniques in a 4-replica configuration without malicious behavior. Each plot presents the throughput of an operational replica as well as the recovering replica over time, with the gray area indicating the synchronization interval. During synchronization, the throughput of the operational replica experiences a slight decline. As expected, the throughput of the recovering replica drops to zero throughout the synchronization period. Upon completion of the synchronization, the recovering replica exhibits a transient spike in throughput due to the backlog of queued requests accumulated during the synchronization process. Subsequently, both the operational and recovering replicas converge to throughput levels comparable to those observed before the state transfer. The synchronization duration is notably shorter for SVCSkipList. Moreover, Table 4.6 provides additional insights. Although the throughput of both techniques is similar during state synchronization, SVCSkipList has lower latency than the Baseline in most percentiles.



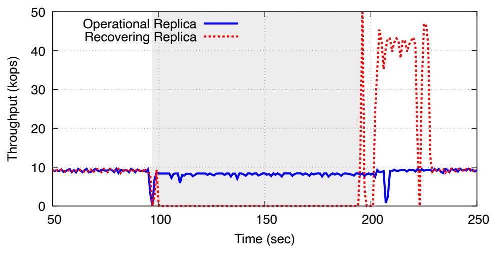
(a) SVCSkipList, 4 replicas, no attack



(b) Baseline, 4 replicas, no attack



(c) SVCSkipList, 7 replicas, under attack



(d) Baseline, 7 replicas, under attack

Figure 4.16. Throughput over time for SVCSkipList and Baseline, showing both an operational replica and the recovering replica, in a system with four replicas without attacks (top) and seven replicas under attack by two Byzantine replicas (bottom). The gray area indicates the state synchronization period.

State synchronization under attack We also investigate the behavior of state synchronization under adversarial conditions. We emulate Byzantine attacks in which a faulty replica corrupts the state before sending it during synchronization. More precisely, the attack involves altering the value of a randomly selected key in the key-value store before transmission. For SVCSKIPLIST, the corruption occurs within a single cluster, which is the unit of data transmission in our design.

Table 4.5 shows the average, minimum, and maximum state synchronization times for both approaches under attack. With four replicas (i.e., one Byzantine replica), SVCSKIPLIST achieved an average synchronization time of approximately 36.1 seconds, while the Baseline required about 51.5 seconds, also suffering higher variance. Notably, the synchronization time of SVCSKIPLIST under attack remains equivalent to that observed without attacks, due to it's ability to independently validate clusters, discarding only invalid ones to be promptly re-fetched, all while maintaining the parallel transfer and validation of remaining clusters.

With seven replicas (i.e., two Byzantine replicas), SVCSKIPLIST completed synchronization in an average of 37.3 seconds, whereas the Baseline approach took significantly longer, averaging 103.2 seconds, again showing higher variance. These results demonstrate that SVCSkipList consistently outperforms the Baseline under attack scenarios, achieving up to 64% lower synchronization time in the seven-replica configuration, while also showing more stable performance across runs.

SVCSkipList demonstrates reduced synchronization times even with additional Byzantine replicas. This is due to its design enabling parallel cluster transfers and per-cluster self-verification, allowing honest replicas to be used concurrently and corrupted clusters to be efficiently discarded and promptly re-fetched. In contrast, the Baseline approach suffers a significant degradation under attacks, with synchronization time nearly doubling in the presence of two Byzantine replicas. This results from its monolithic state transfer approach, which necessitates restarting the entire synchronization upon any corruption detection, thereby limiting scalability and resilience under Byzantine faults.

Figures 4.16c and 4.16d compare the interference on system throughput for both SVCSkipList and Baseline approaches during state synchronization under attack by two Byzantine replicas in a seven-replica configuration. The results demonstrate that SVCSkipList significantly reduces synchronization time under attack with minimal impact on overall system throughput, while the Baseline incurs substantially longer synchronization times, adversely affecting system performance.

4.4 Related work

To contextualize our contributions, we review related efforts across three key areas: state synchronization in replicated systems, blockchain-based synchronization techniques, and the use of skiplists as data structures. Prior research in state machine replication—especially in Byzantine fault-tolerant (BFT) systems—has explored various mechanisms for checkpointing, recovery, and state transfer. In parallel, blockchain platforms have developed Merkle-based structures to support scalable and verifiable synchronization, driven by the demands of decentralization and large-scale state. Finally, skiplists have been widely adopted in systems literature for their simplicity and efficiency, yet their potential remains underexplored in the context of BFT state management. We discuss representative works in each domain and position our contributions in relation to them.

4.4.1 State synchronization

Improving the speed of state synchronization without the need to process the entire transaction log has been a central focus for many Byzantine fault-tolerant state machine replication (BFT SMR) systems. Although previous work has addressed strategies for checkpointing and recovery in BFT SMR [20, 32], the use of specialized data structures to facilitate transfer and validation remains unexplored. In [12], the authors advance BFT-SMaRt's checkpointing and synchronization techniques. The SMR platform handles state as an opaque entity through application-provided functions. Sequential checkpointing is proposed to mitigate the performance impact during normal operation by staggering state snapshots across replicas. While a complete snapshot is obtained from a single replica, the missing command log is partitioned and retrieved in parallel from multiple replicas. Additional replicas provide hashes of the state and log segments, enabling validation.

In [18], the authors propose proactive recovery within PBFT, a seminal approach to state organization for optimizing transfer in BFT systems. The state is structured as a sequence of fixed-size pages, each corresponding to a leaf of a Merkle tree. Page contents are opaque to the SMR platform, as they are managed by the application. Checkpoints are periodically created at consensus intervals, during which replicas produce a copy of the Merkle tree using a copy-on-write strategy. During recovery, a replica first obtains a certified Merkle tree root (from $f + 1$ matching replies) for a given checkpoint. It then recursively retrieves internal nodes (metadata) and leaf pages from different replicas. Replies include the page digests and the last checkpoint number. Once all pages under a common parent node are collected, the parent can be verified. This process recurses up to the root, confirming consistency for the checkpoint.

While in [18] state and metadata use different structures, [41] proposes a single structure that combines application state and facilitates partitioning and verification: the AVL* tree, a Merkleized AVL tree. Designed for blockchains, the AVL* tree provides key-value store functionality at the application level while organizing tree chunks for efficient state transfer, reconstruction, and rapid validation, and also reduces the throughput impact of checkpointing. During normal operation, updates propagate incrementally: when a transaction affects a chunk, only the chunk's hash and its path to the root are recomputed. However, to keep the AVL* balanced, nodes may move from one chunk to another, adding complexity to the algorithms. A recovering or new peer obtains a trusted Merkle root and the number of chunks from a subsequent block header. The peer then downloads chunks in parallel from different peers, validating each chunk independently by verifying its proof of inclusion against the Merkle root and recomputing internal hashes from its leaves to the chunk root. A deterministic reconstruction process guarantees consistent tree structures across correct peers.

Our work has some similarity with [41], which is in the context of blockchains, however, we either optimize tree structures in terms of cache efficiency, performance, proof sizes and space efficiency (B+AVL tree), or target generic SMR systems and use different data structures, such as skiplists, instead of tree structures (SVCSkipList), borrowing the advantages of skiplists already mentioned before and opening to possible future algorithmic improvements based on skiplist optimizations and variations [95].

Concerns about the impact of checkpointing and fast state transfer also appear in the crash fault-tolerant SMR literature. In [58], the authors propose using different replicas to store parts of the state (partitions), enabling recovery to fetch different partitions in parallel from different replicas. The approach ensures that partition snapshots remain consistent with respect to cross-partition commands. The logs are applied to each partition to ensure a correct, up-to-date, and complete state. Along similar lines, on-demand state transfer is proposed in [75]. In this approach, parts of the state are transferred as needed by the recovering replica, which begins processing the log and new commands without necessarily completing the installation of the entire state.

4.4.2 Blockchains

Unlike traditional SMR systems, which often deem Merkle trees too expensive or only utilize them periodically for synchronization, many blockchain systems already leverage Merkle-based data structures for state storage. Today, as state sizes in blockchain systems continue to expand, synchronizing becomes more costly. Consequently, Merkle trees have garnered considerable attention. These trees serve a vital role in enabling light clients, empowering them to query specific leaves efficiently and verify their integrity without the need to download the entire state or transaction history.

In Geth [43], Ethereum's Go implementation, state synchronization involves requesting individual nodes of the tree. Peers are unable to anticipate the duration of this synchronization process as they lack knowledge of the total number of nodes [45]. Since the Merkle tree is a component of the consensus rules, with Merkle roots stored in block headers, peers can authenticate the accuracy of received nodes. However, the performance of both requesting and providing peers is adversely affected by the small size of nodes (less than 1 KB), their randomized distribution in the database, and the substantial state size (tens of GBs), when nodes are requested individually.

Batching nodes of a tree could be a potential solution, yet it poses challenges in ensuring verifiability while safeguarding honest peers against malicious attacks. For instance, in OpenEthereum [81], snapshots are periodically created by serializing the entire state and segmenting it into sizeable chunks, each associated with hashes published in a manifest file. However, since the manifest isn't integrated into the consensus mechanism, verifying the correctness of a chunk before downloading all chunks isn't feasible. Thus, the successful completion of state synchronization hinges on obtaining a correct manifest, necessitating strong assumptions like trusting a specific peer or presuming a majority of connected peers to be honest. In Tendermint IAVL+ snapshots, tree nodes are serialized and grouped into fixed-size chunks [1]. However, because Tendermint's block header lacks snapshot hashes, peers cannot verify chunks incrementally, underscoring the need for a tree structure with built-in chunking support.

Blockchain scalability improvements can also be classified into two main categories: on-chain and off-chain. On-chain solutions enhance the blockchain infrastructure itself, such as efficient consensus algorithms [60][85] and sharding [71][98]. Off-chain solutions, or “Layer-2” (L2) techniques, offload computation and storage to reduce the load on the main blockchain (L1). Examples include State Channels [79], and rollups [17]. zkSync Lite [64], a zero-knowledge rollup, achieves a performance nearly 6 times faster than Ethereum.

In the context of optimizing rollup mechanisms, Sparse Merkle trees have been adopted. In [72] the authors study the sparse Merkle tree algorithms presented in zkSync Lite, and propose an efficient batch update algorithm to calculate a new root hash given a list of account (leaf) operations. Using the construction in zkSync Lite as a benchmark, their algorithm improves the account update time from $O(log_n)$ to $O(1)$ and reduces the batch update cost by half using a one-pass traversal. Advances in cryptography have also introduced generalizations of Merkle trees called accumulators, enabling $O(1)$ proofs of set-membership and state-less blockchain clients [14].

4.4.3 Skiplists

In a comprehensive survey [95], skiplists are argued to be simple, easy to implement, and have the same asymptotic complexity as tree-based counterparts. Experiments comparing AVL and self-adjusting trees experimentally show that skiplists have better search, insert, and delete performance. Seen as an alternative to balanced trees, skiplists have become widely adopted in various systems due to their simplicity, as they do not require re-balancing. Since the original skiplist was introduced [84], a considerable body of optimizations has emerged, including concurrent lock-free operations, multi-versioning, and partitioning. Partitioning the skiplist can reduce its height, thereby decreasing the number of accesses and pointer updates, as demonstrated in [70]. This approach is generally considered simpler to implement than in tree-based structures. Both [48] and [31] address the issue of verifying information retrieved from a non-trusted entity. The first employs hashed skiplists, while the second uses skiplists to store hashes of table entries maintained in a Database Management System (DBMS). The data stored and hashes are provided by a trusted source. Users retrieving data from the untrusted storage can verify it with the stored hashes and public information from the source. Additionally, several key-value store implementations based on skiplists are available.

Although skiplists have been extensively studied, the combination of versioning, partitioning, and self-validation in a single skiplist was not found in the literature. Furthermore, the application of skiplists to optimize state management in BFT SMR systems remains unexplored.

4.5 Conclusion

In this chapter, we presented two complementary data structures that address key aspects in state machine replication under Byzantine fault tolerance. Our goal was to improve the efficiency and robustness of state management, both in terms of what is stored (data structure layout) and how it is synchronized (state transfer protocols). Below, we summarize the benefits and evaluation results of each structure.

4.5.1 B+AVL tree

We addressed the challenge of efficient state transfer in blockchain systems, particularly for the recovery of out-of-sync peers. While existing solutions rely on snapshots and Merkle-based validation to avoid full transaction replay, they face limitations in clustering efficiency and proof sizes. We proposed B+AVL trees, a novel data structure that combines the balancing and validation strengths of AVL trees with the space efficiency of B+Trees. Unlike AVL* trees, B+AVL trees simplify cluster maintenance by preventing leaf movement across clusters and provide more compact validation proofs than Merkle B+Trees. Experimental results demonstrate the practical advantages of this approach.

Table 4.7 presents a quantitative analysis of our findings. It highlights the key advantages of B+AVL trees and provides a concise summary of the results across various metrics, comparing the performance of all techniques normalized to the Baseline tree, with smaller values representing better performance.

In summary:

- The B+AVL tree demonstrates consistency in performance across insertion and search operations, comparable to other tree structures. Notably, it outperforms all other techniques in search operations with large trees, since it stores keys contiguously within a cluster, optimizing cache utilization.
- The B+AVL tree demonstrates notably smaller Merkle proof sizes compared to traditional B+Trees and exhibits more stable proof sizes than the AVL* tree as the cluster size increases, as reflected in the standard deviation values for proof sizes in Figure 4.5.
- The Baseline tree achieves the highest space efficiency by fully filling all clusters except the last. The B+AVL, however, presents superior space efficiency compared to the remaining techniques, especially evident with smaller cluster sizes.

		Baseline tree	**AVL* tree**	**B+Tree**	**B+AVL tree**
Search	Small tree	**1**	2.86	1.95	1.68
	Large tree	1	1.04	0.76	**0.32**
Insert	Small tree	**1**	1.10	1.64	2.05
	Large tree	1	0.86	**0.83**	0.89
Proof size	Small cluster	1	1	2.85	**0.99**
	Large cluster	1	0.99	192.20	**0.98**
Space efficiency	Small cluster	**1**	1.58	1.42	1.42
	Large cluster	**1**	1.48	1.40	1.41
No Byz. State Sync.	Small cluster	1	—	—	**0.97**
	Large cluster	1	—	—	**0.32**
	Few peers	1	—	—	**0.63**
	Many peers	1	—	—	**0.62**
One Byz. State Sync.	Small cluster	1	—	—	**0.40**
	Large cluster	1	—	—	**0.15**
	Few peers	1	—	—	**0.21**
	Many peers	1	—	—	**0.33**
Many Byz. State Sync.	1 byz. peer	1	—	—	**0.20**
	4 byz. peers	1	—	—	**0.04**

Table 4.7. Summary of the results of our B+AVL tree for various metrics evaluated, comparing the performance of all techniques normalized to the Baseline tree. The values represent the average performance observed across multiple experiments. Smaller values indicate better performance. Best results shown in boldface.

While all techniques exhibit similar average space efficiency, the B+AVL and B+Tree demonstrate more unstable variance as the tree expands (see standard deviation values for large clusters in Table 4.2).

- In state synchronization, the B+AVL outperforms the Baseline tree in both non-Byzantine and Byzantine settings. Its faster sync stems from efficient serialization and rebuilds enabled by a compact memory layout. Under attack, self-verifiable clusters allow rapid detection and recovery, while the Baseline suffers from costly retries. The performance gap grows with more Byzantine nodes, and in some cases, B+AVL performs even better under attack due to optimized recovery paths.

4.5.2 SVCSkipList

We also introduced the concept of self-validating clustered data structures as a principled approach to improving state management in Byzantine fault-tolerant state machine replication (BFT SMR). We instantiated this idea with the SVCSkipList, a novel data structure that enables efficient, parallel, and independently verifiable state transfer. By tightly integrating state representation and validation into the replication framework, our solution addresses critical limitations of existing SMR libraries, especially in the presence of Byzantine faults.

Execution Phase	**Key Benefit of SVCSkipList**
Normal Execution	Comparable or higher throughput; significantly lower tail latency in all workloads
Checkpointing	Up to 2× faster for large states due to multiversioning
Sync (No Attack)	Faster and more consistent synchronization times, especially in large replica settings
Sync (Under Attack)	Up to 64% lower times thanks to parallel, cluster-based verification

Table 4.8. Summary of performance improvements achieved by SVCSkipList compared to baseline.

We integrated our proposed SVCSkipList data structure into the well-known BFT-SMaRt framework and evaluated it in a wide-area network (WAN) environment. Our evaluation demonstrates that SVCSkipList provides robust performance across a range of workloads and system sizes, offering improvements in both normal execution and state synchronization scenarios. Table 4.8 presents a concise comparison of the performance gains across execution phases.

In summary:

- **Normal execution:** Our approach, employing the SVCSkipList, achieves throughput comparable to the baseline in all configurations, and consistently higher in larger state sizes. It maintains stable average and tail latencies, even as state grows from 1GB to 4GB. Notably, it also outperforms the baseline in high-percentile latencies (e.g., 99.9% and 99.99%)—reducing tail latencies by up to 60%, and exhibits more resilience to performance degradation in larger deployments (7 replicas).

- **Checkpointing:** Our approach reduces checkpointing times significantly thanks to efficient multiversioning and skiplist structure, and halves checkpointing time for large states (up to 4GB) compared to the baseline.

- **State synchronization without attack:** Our approach shows slightly faster and more consistent synchronization times (e.g., 36.3s vs. 39.3s for 4 replicas), and scales better in larger replica groups (e.g., 35.7s vs. 44.2s for 7 replicas).

- **State synchronization under Byzantine attack:** It also achieves up to 64% lower synchronization time (e.g., 37.3s vs. 103.2s with 7 replicas), and avoids full state transfer restarts by validating clusters independently in parallel, while the baseline suffers from cascading retries upon any detected corruption.

4.5.3 Final remarks

Together, the B+AVL tree and SVCSkipList offer a principled and practical approach to improving Byzantine-resilient state machine replication. The B+AVL tree addresses storage and proof size limitations. Both the B+AVL tree and the SVCSkipList data structures enhance recovery and synchronization by enabling parallelism and cluster-based verification. By integrating these designs into replication frameworks, systems can significantly reduce performance overheads and improve resilience in adversarial and large-scale environments.

Chapter 5

Conclusion

This thesis addressed fundamental challenges in the design and implementation of State Machine Replication (SMR) systems, focusing on efficient communication and state management. Although SMR provides a powerful abstraction for building fault-tolerant distributed systems, its performance and scalability remain limited by the costs of its main structural components, such as replica coordination and state synchronization. The work presented here proposed novel techniques to mitigate some of these limitations, demonstrating that it is possible to achieve higher throughput and lower latency without compromising correctness or dependability.

5.1 Summary of contributions

The main contributions of this thesis span two complementary dimensions: communication and state management.

- **Quiescence:** The introduction of a novel atomic multicast property to refine minimality, ensuring correctness and performance during communication. While minimality specifies when processes may exchange messages, quiescence indicates when they should cease communication.

- **FlexCast:** An overlay-based and genuine atomic multicast protocol that reduces communication overhead while preserving ordering guarantees. By leveraging a directed acyclic graph (DAG) overlay and local ordering decisions, FlexCast achieves lower latency and better scalability in geographically distributed environments than state-of-the-art protocols.

- **Reconfigurable FlexCast:** A dynamic reconfiguration mechanism that enables the multicast overlay to adapt to workload and network changes.

- **B+AVL trees:** A novel data structure combining the balancing and verification strengths of AVL trees with the compactness of B+Trees. B+AVL trees enable efficient and verifiable state transfer by providing smaller proofs, stable performance across workloads, and optimized memory layout.

- **SVCSkipList:** The first realization of a *self-validating clustered data structure* for general-purpose Byzantine fault-tolerant SMR. The SVCSkipList supports parallel and independently verifiable state transfer, achieving substantial performance gains in throughput, checkpointing, and synchronization, even under Byzantine conditions.

Together, these contributions advance the understanding and practical realization of scalable and dependable replicated systems. They demonstrate how a combination of communication and data structure optimizations can deliver measurable improvements in latency, throughput, and recovery efficiency while maintaining strong consistency guarantees.

5.2 Future directions

While this thesis makes substantial progress, it also opens several avenues for future exploration. These can be grouped into two main categories: (i) incremental extensions to the current work, and (ii) broader research directions inspired by its findings.

5.2.1 Incremental extensions

- **Learning-based overlay optimization:** While FlexCast reconfiguration adapts overlays to workload locality, future work could explore using AI or machine learning to detect patterns in traffic, latency, or node reliability, and proactively optimize the overlay. For example, a lightweight model could predict hotspots, anticipate failures, or identify groups of replicas that would benefit from local ordering adjustments, reducing latency and communication overhead in dynamic or geographically distributed deployments. Hybrid approaches that combine genuine and hierarchical multicast could also be guided by such models to further improve efficiency.

- **Adaptive integration of data structures:** The proposed data structures, B+AVL trees and SVCSkipList, could be integrated into existing blockchain platforms or SMR libraries in a way that allows the system to adaptively select which structure to use based on workload characteristics, state size, or performance requirements. For example, B+AVL trees could be used for compact, cache-efficient state updates, while SVCSkipList could handle large-scale or Byzantine-resilient state transfers. Such adaptive selection, combined with optimizations for memory management, serialization, and caching, would enable more efficient and flexible deployment in real-world environments.

- **Extending other data structures with clustering and self-validation:** Beyond B+AVL trees and SVCSkipList, future work could investigate how other types of data structures, such as graphs, tries, or hash-based indexes, could be enhanced with clustering, multiversioning, and self-validating mechanisms. Studying these structures would allow evaluation of performance trade-offs, applicability to different workloads, and resilience under Byzantine or high-latency conditions. Such exploration could broaden the range of verifiable state management techniques for SMR and blockchain systems, and provide guidance for workload-specific optimizations.

5.2.2 High-level research directions

Beyond the immediate extensions discussed above, this work also opens broader research directions that can be pursued based on the mechanisms and insights developed in this thesis.

- **Integration with modern consensus engines:** The proposed data structures for state synchronization could also be incorporated into other widely used SMR or blockchain frameworks, such as Tendermint [15], or HotStuff [97], to evaluate their impact in realistic deployment scenarios. This would provide a deeper understanding of how optimized multicast and verifiable state management interact with existing consensus algorithms.

- **Decoupling ordering and payload dissemination:** Future work could explore atomic multicast techniques that decouple the dissemination of request ordering from the actual payload delivery. For example, a tree-based overlay could be used to establish a consistent global order efficiently, while a complete DAG (C-DAG) overlay could simultaneously disseminate the payload to replicas using direct connectivity. This separation would allow each overlay to exploit its strengths, fast ordering with minimal communication in the tree, and efficient, parallel payload distribution in the C-DAG, potentially improving latency, throughput, and scalability in large or geographically distributed SMR deployments.

Overall, these directions build directly upon the foundations laid by this thesis, focusing on practical extensions that can bridge the gap between experimental research and large-scale dependable deployments.

5.3 Final Remarks

In conclusion, this thesis contributes both theoretical and practical advances toward scalable, fault-tolerant replication. By rethinking how replicas communicate and manage state, it shows that the traditional trade-offs between performance and dependability can be mitigated through carefully designed mechanisms. The techniques proposed here, from overlay-based multicast to self-validating data structures, lay the groundwork for the next generation of dependable distributed systems capable of meeting the performance and reliability demands of modern applications.

Bibliography

[1] *ADR 053: State Sync Prototype* [n.d.]. https://github.com/tendermint/-tendermint/blob/master/docs/architecture/adr-053-state-sync-prototype.md.

[2] Ahmed-Nacer, T., Sutra, P. and Conan, D. [2016]. The convoy effect in atomic multicast, *2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 67–72.
URL: *https://doi.ieeecomputersociety.org/10.1109/SRDSW.2016.22*

[3] Alchieri, E., Bessani, A., Greve, F. and Fraga, J. d. S. [2017]. Efficient and Modular Consensus-Free Reconfiguration for Fault-Tolerant Storage, *International Conference on Principles of Distributed Systems*.

[4] Alchieri, E., Dotti, F., Marandi, P., Mendizabal, O. and Pedone, F. [2018]. Boosting state machine replication with concurrent execution, *2018 Eighth Latin-American Symposium on Dependable Computing (LADC)*, pp. 77–86.

[5] Alchieri, E., Dotti, F. and Pedone, F. [2018]. Early scheduling in parallel state machine replication, *ACM SoCC*.

[6] Amazon Web Services [2025]. Amazon EC2 Instances, https://aws.amazon.com/ec2/. Accessed: 2025-06-25.

[7] Batista, E., Alchieri, E., Dotti, F. and Pedone, F. [2019]. Resource utilization analysis of early scheduling in parallel state machine replication, *9th Latin-American Symposium on Dependable Computing (LADC)*.

[8] Batista, E., Alchieri, E., Dotti, F. and Pedone, F. [2022]. Early scheduling on steroids: Boosting parallel state machine replication, *Journal of Parallel and Distributed Computing* .
URL: *https://www.sciencedirect.com/science/article/pii/S0743731522000375*

[9] Batista, E., Coelho, P., Alchieri, E., Dotti, F. and Pedone, F. [2023]. Flexcast: Genuine overlay-based atomic multicast, *Proceedings of the 24th International Middleware Conference*, Middleware '23, Association for Computing Machinery, New York, NY, USA, p. 288–300.

[10] Bessani, A., Alchieri, E., Sousa, J., Oliveira, A. and Pedone, F. [2020]. From byzantine replication to blockchain: Consensus is only the beginning, *International Conference on Dependable Systems and Networks*.

[11] Bessani, A., Santos, M., Felix, J. a., Neves, N. and Correia, M. [2013]. On the efficiency of durable state machine replication, *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, USENIX Association, USA, p. 169–180.

[12] Bessani, A., Sousa, J. and Alchieri, E. E. [2014]. State machine replication for the masses with bft-smart, *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 355–362.

[13] Birman, K. P. and Joseph, T. A. [1987]. Reliable communication in the presence of failures, *ACM Trans. Comput. Syst.* **5**(1): 47–76.
URL: *https://doi.org/10.1145/7351.7478*

[14] Boneh, D., Bünz, B. and Fisch, B. [2019]. Batching techniques for accumulators with applications to iops and stateless blockchains, *Annual International Cryptology Conference*, Springer, pp. 561–586.

[15] Buchman, E., Kwon, J. and Milosevic, Z. [2018]. The latest gossip on BFT consensus, *CoRR* **abs/1807.04938**.
URL: *http://arxiv.org/abs/1807.04938*

[16] Burrows, M. [2006]. The chubby lock service for loosely-coupled distributed systems, *OSDI*.

[17] Buterin, V. [n.d.]. An incomplete guide to rollups, `https://vitalik.eth.limo/general/2021/01/05/rollup.html`.

[18] Castro, M. and Liskov, B. [2002]. Practical byzantine fault tolerance and proactive recovery, *ACM Trans. Comput. Syst.* **20**(4): 398–461.
URL: *https://doi.org/10.1145/571637.571640*

[19] Chilimbi, T. M., Hill, M. D. and Larus, J. R. [1999]. Cache-conscious data structures: Design and implementation, *Proceedings of the ACM SIGPLAN Conference*

on Programming Language Design and Implementation (PLDI).
URL: *https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/ccds.pdf*

[20] Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M. and Riche, T. [2009]. Upright cluster services, *SOSP*.

[21] Cloudping [2022]. AWS Latency Monitoring Website.
URL: *https://www.cloudping.co/grid*

[22] Coelho, P., Junior, T. C., Bessani, A., Dotti, F. and Pedone, F. [2018]. Byzantine fault-tolerant atomic multicast, *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 39–50.

[23] Coelho, P., Schiper, N. and Pedone, F. [2017]. Fast atomic multicast, *DSN*.

[24] Comer, D. [1979]. Ubiquitous b-tree, *ACM Comput. Surv.* **11**(2): 121–137.
URL: *https://doi.org/10.1145/356770.356776*

[25] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R. and Sears, R. [2010]. Benchmarking cloud serving systems with YCSB, *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, ACM, pp. 143–154.

[26] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P. et al. [2012]. Spanner: Google's globally-distributed database, *OSDI*.

[27] *Cosmos network* [n.d.]. https://cosmos.network/.

[28] *Cosmos SDK* [n.d.]. https://github.com/cosmos/cosmos-sdk.

[29] Council, T. P. P. [1996]. Tpc benchmark c standard specification, *http://www. tpc. org/tpcc/spec/tpcc_current. pdf* .

[30] Delporte-Gallet, C. and Fauconnier, H. [2000]. Fault-tolerant genuine atomic multicast to multiple groups, *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS)*, pp. 107–122.

[31] Di Battista, G. and Palazzi, B. [2007]. Authenticated relational tables and authenticated skip lists, *in* S. Barker and G.-J. Ahn (eds), *Data and Applications Security XXI*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 31–46.

[32] Distler, T. [2021]. Byzantine fault-tolerant state-machine replication from a systems perspective, *ACM Comput. Surv.* **54**(1).
URL: *https://doi.org/10.1145/3436728*

[33] Drees, M., Gmyr, R. and Scheideler, C. [2016]. Churn- and dos-resistant overlay networks based on network reconfiguration, *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*.

[34] Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., Akella, A., Wang, K., Ricart, G., Landweber, L., Elliott, C., Zink, M., Cecchet, E., Kar, S. and Mishra, P. [2019]. The design and operation of CloudLab, *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 1–14.
URL: *https://www.flux.utah.edu/paper/duplyakin-atc19*

[35] Dwork, C., Lynch, N. and Stockmeyer, L. [1988]. Consensus in the presence of partial synchrony, *Journal of the ACM* **35**(2): 288–323.

[36] Facebook [2013]. Rocksdb: A persistent key-value store for flash and ram storage. https://github.com/facebook/rocksdb.

[37] Fischer, M. J., Lynch, N. A. and Paterson, M. S. [1985]. Impossibility of distributed consensus with one faulty processor, *Journal of the ACM* **32**(2): 374–382.

[38] Friedman, R. and van Renesse, R. [1997]. Packing messages as a tool for boosting the performance of total ordering protocols, *Proceedings of the 6th International Symposium on High Performance Distributed Computing, HPDC '97, Portland, OR, USA, August 5-8, 1997*, IEEE Computer Society, pp. 233–242.

[39] Fritzke, U., J., Ingels, P., Mostefaoui, A. and Raynal, M. [1998]. Fault-tolerant total order multicast to asynchronous groups, *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, pp. 228–234.

[40] Fynn, E. [2021]. *Scaling Blockchains*, PhD thesis, Università della Svizzera Italiana (USI).

[41] Fynn, E., Buchman, E., Milosevic, Z., Soulé, R. and Pedone, F. [2022]. Robust and fast blockchain state synchronization, *in* E. Hillel, R. Palmieri and E. Rivière (eds), *26th International Conference on Principles of Distributed Systems, OPODIS 2022, December 13-15, 2022, Brussels, Belgium*, Vol. 253 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 8:1–8:22.

[42] Garcia-Molina, H. and Spauster, A. [1989]. Message ordering in a multicast environment, *[1989] Proceedings. The 9th International Conference on Distributed Computing Systems*, pp. 354–361.

[43] *Geth v1.9.0: Six months distilled* [n.d.]. https://blog.ethereum.org/2019/07/10/geth-v1-9-0/.

[44] Glendenning, L., Beschastnikh, I., Krishnamurthy, A. and Anderson, T. [2011]. Scalable consistency in scatter, *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*.

[45] *Go Ethereum FAQ* [n.d.]. https://geth.ethereum.org/docs/faq.

[46] Goodrich, M. T. and Tamassia, R. [2001]. Efficient authenticated dictionaries with skip lists and commutative hashing, *Tech. Rep.* .
URL: *https://cs.brown.edu/cgc/stms/papers/hashskip.pdf*

[47] Goodrich, M. T., Tamassia, R. and Goldwasser, M. H. [2014]. *Data Structures and Algorithms in Java*, 6 edn, John Wiley & Sons, Hoboken, NJ.

[48] Goodrich, M., Tamassia, R. and Schwerin, A. [2001]. Implementation of an authenticated dictionary with skip lists and commutative hashing, *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01*, Vol. 2, pp. 68–82 vol.2.

[49] Google [2011]. Leveldb. https://github.com/google/leveldb.

[50] Gotsman, A., Lefort, A. and Chockler, G. [2019]. White-box atomic multicast, *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, pp. 176–187.

[51] Guerraoui, R. and Schiper, A. [2001]. Genuine atomic multicast in asynchronous distributed systems, *Theor. Comput. Sci.* **254**(1-2): 297–316.

[52] Hadzilacos, V. and Toueg, S. [1994a]. A modular approach to fault-tolerant broadcasts and related problems, *Technical report*, USA.

[53] Hadzilacos, V. and Toueg, S. [1994b]. A modular approach to the specification and implementation of fault-tolerant broadcasts, *Technical report*, Department of Computer Science, Cornell.

[54] Herlihy, M. and Wing, J. M. [1990]. Linearizability: A correctness condition for concurrent objects, *ACM Transactions on Programing Languages and Systems* **12**(3): 463–492.

[55] Hoang Le, L., Fynn, E., Eslahi-Kelorazi, M., Soulé, R. and Pedone, F. [2019]. Dynastar: Optimized dynamic partitioning for scalable state machine replication, *Int. Conference on Distributed Computing Systems*.

[56] Hunt, P., Konar, M., Junqueira, F. P. and Reed, B. [2010]. Zookeeper: wait-free coordination for internet-scale systems, *ATC*, Vol. 8.

[57] *IAVL+ implementation* [n.d.]. `https://github.com/tendermint/iavl`.

[58] Junior, E. G., Alchieri, E., Dotti, F. L. and Mendizabal, O. M. [2024]. Reducing persistence overhead in parallel state machine replication through time-phased partitioned checkpoint, *J. Internet Serv. Appl.* **15**(1): 194–211.
URL: *https://doi.org/10.5753/jisa.2024.3891*

[59] Kapritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L. and Dahlin, M. [2012]. All about eve: execute-verify replication for multi-core servers, *OSDI*.

[60] Kiayias, A., Russell, A., David, B. and Oliynykov, R. [2017]. Ouroboros: A provably secure proof-of-stake blockchain protocol, *in* J. Katz and H. Shacham (eds), *Advances in Cryptology – CRYPTO 2017*, Springer International Publishing, Cham, pp. 357–388.

[61] Knuth, D. [1973]. *The Art Of Computer Programming, vol. 3: Sorting And Searching*, Addison-Wesley.

[62] Kotla, R. and Dahlin, M. [2003]. High throughput byzantine fault tolerance, *Technical Report UTCS-TR-03-58*, University of Texas.

[63] Kuhn, F. and Wattenhofer, R. [2004]. Dynamic analysis of the arrow distributed protocol, *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, Association for Computing Machinery, New York, NY, USA, p. 294–301.
URL: *https://doi.org/10.1145/1007912.1007962*

[64] Labs, M. [n.d.]. Zksync: scaling and privacy engine for ethereum, `https://github.com/matter-labs/zksync`.

[65] Lamport, L. [1978]. Time, clocks, and the ordering of events in a distributed system, *Communications of the ACM* **21**(7): 558–565.

[66] Lamport, L. [1989]. The part-time parliament., *Technical Report 49*, Digital Equipment Corporation, Systems Research Centre.

[67] Lamport, L. [1998]. The part-time parliament, *ACM Transactions on Computer Systems* **16**(2): 133–169.

[68] Lamport, L. [2005]. Generalized Consensus and Paxos, *Technical report*, Microsoft Research Technical Report MSR-TR-2005-33.

[69] Le, L. H., Eslahi-Kelorazi, M., Coelho, P. R. and Pedone, F. [2021]. Ramcast: Rdma-based atomic multicast, *Proceedings of the 22nd International Middleware Conference* .

[70] Li, Z., Jiao, B., He, S. and Yu, W. [2022]. Phast: Hierarchical concurrent log-free skip list for persistent memory, *IEEE Transactions on Parallel and Distributed Systems* **33**(12): 3929–3941.

[71] Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S. and Saxena, P. [2016]. A secure sharding protocol for open blockchains, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, Association for Computing Machinery, New York, NY, USA, p. 17–30.
URL: *https://doi.org/10.1145/2976749.2978389*

[72] Ma, B., Pathak, V. N., Liu, L. and Ruj, S. [2023]. One-phase batch update on sparse merkle trees for rollups.
URL: *https://arxiv.org/abs/2310.13328*

[73] Mao, Y., Junqueira, F. P. and Marzullo, K. [2008]. Mencius: building efficient replicated state machines for wans, *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI)*, pp. 369–384.

[74] Marandi, P. J., Bezerra, C. E. B. and Pedone, F. [2014]. Rethinking state-machine replication for parallelism, *ICDCS*.

[75] Mendizabal, O. M., Dotti, F. L. and Pedone, F. [2017]. High performance recovery for parallel state machine replication, *in* K. Lee and L. Liu (eds), *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, IEEE Computer Society, pp. 34–44.
URL: *https://doi.org/10.1109/ICDCS.2017.193*

[76] Merkle, R. C. [1988]. A digital signature based on a conventional encryption function, *in* C. Pomerance (ed.), *Advances in Cryptology — CRYPTO '87*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 369–378.

[77] Moraru, I., Andersen, D. G. and Kaminsky, M. [2013]. There is more consensus in egalitarian parliaments, *SOSP*.

[78] Munro, J. I., Papadakis, T. and Sedgewick, R. [1992]. Deterministic skip lists, *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, Society for Industrial and Applied Mathematics, USA, p. 367–375.

[79] Negka, L. D. and Spathoulas, G. P. [2021]. Blockchain state channels: A state of the art, *IEEE Access* **9**: 160277–160298.

[80] Obelheiro, R. R. and Fraga, J. d. S. [2007]. Overlay network topology reconfiguration in byzantine settings, *13th Pacific Rim International Symposium on Dependable Computing*, pp. 155–162.

[81] *OpenEthereum WarpSync* [n.d.]. https://openethereum.github.io/wiki/Warp-Sync.

[82] Parzyjegla, H., Muhl, G. and Jaeger, M. [2006]. Reconfiguring publish/subscribe overlay topologies, *26th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'06)*, pp. 29–29.

[83] Pedone, F. and Schiper, A. [1999]. Generic broadcast, *Distributed Computing*, Springer, pp. 94–106.

[84] Pugh, W. [1990]. Skip lists: a probabilistic alternative to balanced trees, *Commun. ACM* **33**(6): 668–676.
URL: *https://doi.org/10.1145/78973.78977*

[85] Rocket, T., Yin, M., Sekniqi, K., van Renesse, R. and Sirer, E. G. [2020]. Scalable and probabilistic leaderless bft consensus through metastability.
URL: *https://arxiv.org/abs/1906.08936*

[86] Rodrigues, L., Guerraoui, R. and Schiper, A. [1998]. Scalable atomic multicast, *International Conference on Computer Communications and Networks*, pp. 840–847.

[87] Sanfilippo, S. et al. [2009]. Redis. https://redis.io.

[88] Schiper, N. and Pedone, F. [2008a]. On the inherent cost of atomic broadcast and multicast in wide area networks, *International conference on Distributed computing and networking (ICDCN)*, pp. 147–157.

[89] Schiper, N. and Pedone, F. [2008b]. Solving atomic multicast when groups crash, *International Conference On Principles Of Distributed Systems (OPODIS)*, Springer, pp. 481–495.

[90] Schneider, F. B. [1990]. Implementing fault-tolerant services using the state machine approach: A tutorial, *ACM Computing Surveys* **22**(4): 299–319.

[91] Shvachko, K., Kuang, H., Radia, S. and Chansler, R. [2010]. The hadoop distributed file system, *MSST*.

[92] Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P. and Abadi, D. J. [2012]. Calvin: fast distributed transactions for partitioned database systems, *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 1–12.

[93] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E. and Wilkes, J. [2015]. Large-scale cluster management at google with borg, *EuroSys*.

[94] Wood, G. et al. [2014]. Ethereum: A secure decentralised generalised transaction ledger, *Ethereum project yellow paper* **151**(2014): 1–32.

[95] Xing, L., Vadrevu, V. S. P. K. and Aref, W. G. [2025]. The ubiquitous skiplist: A survey of what cannot be skipped about the skiplist and its applications in data systems, *ACM Comput. Surv.* **57**(11).
URL: *https://doi.org/10.1145/3736754*

[96] Xing, L., Vadrevu, V. S. P. K. and Ghanem, W. [2023]. The ubiquitous skiplist: A survey of what cannot be skipped about the skiplist and its applications in data systems, *ACM Computing Surveys* .
URL: *https://dl.acm.org/doi/10.1145/3736754*

[97] Yin, M., Malkhi, D., Reiter, M. K., Gueta, G. G. and Abraham, I. [2019]. Hotstuff: Bft consensus with linearity and responsiveness, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, Association for Computing Machinery, New York, NY, USA, p. 347–356.
URL: *https://doi.org/10.1145/3293611.3331591*

[98] Zamani, M., Movahedi, M. and Raykova, M. [2018]. Rapidchain: Scaling blockchain via full sharding, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, Association for Computing Machinery, New York, NY, USA, p. 931–948.
URL: *https://doi.org/10.1145/3243734.3243853*