TEE-Based Distributed Ledgers and Their Resilience

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik des
Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Marc Leinweber
aus Fulda

Tag der mündlichen Prüfung: 5. November 2025

1. Referent: Prof. Dr. rer. nat. Hannes Hartenstein
Karlsruher Institut für Technologie (KIT)
2. Referent: Prof. Dr.-Ing. Rüdiger Kapitza
Friedrich-Alexander-Universität Erlangen-Nürnberg

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

(Leslie Lamport, 1987)

Abstract

Resilience is the ability of a (distributed) system to withstand any stressful situation without imposing massive restrictions and, above all, without long-term consequences. Permissioned distributed ledgers based on state machine replication (SMR) offer a promising approach to achieving high resilience and fairness in federated systems. SMR provides a fault-tolerant service for clients by relying on all replicas being in a consistent state. The consistent state is achieved through a consensus algorithm, typically an atomic broadcast, that decides on a total order of client requests. In the Byzantine fault model, replicas are assumed to be potentially malicious; a Byzantine fault-tolerant (BFT) protocol withstands a fixed share of malicious actors. Classic BFT SMR protocols require $n > 3t$ replicas and multiple rounds of communication to withstand t faulty replicas, making the implementation complex and limiting achievable throughput and increasing latency. Trusted Execution Environments (TEEs) allow to implement SMR in the so-called hybrid fault model in which replicas are assumed to be potentially Byzantine but the TEE is restricted to only fail by crashing. In the hybrid fault model, SMR requires less communication and can be implemented with a fault tolerance of $n > 2t$ replicas. While many proposals aim to optimize BFT SMR by using TEEs, they still rely on a so-called leader that coordinates the agreement process among the replicas. The leader is known to be a bottleneck and, if it fails, the system has to recover from the failure and elect a new leader. The additional coordination required to elect a new leader can cause significant performance degradation, limiting the achieved resilience. Asynchronous protocols based on directed acyclic graphs (DAGs) eliminate the reliance on distinguished replicas by allowing all replicas to participate equally in the agreement process. While asynchronous approaches and the hybrid fault model independently contribute to increasing the resilience of BFT SMR systems, their combination has largely been unexplored. This dissertation aims to fill this gap by answering the following research question:

What is the achievable performance and resilience of DAG-based, hybrid fault-tolerant state machine replication and under which preconditions can the leaderless nature be safely exploited to maximize throughput?

We proceed in three steps to enhance the resilience and performance of BFT SMR systems and to identify potential trade-offs that arise from the assumption of TEEs and asynchrony in BFT SMR. First, we investigate the fit of TEE-based SMR for consortium-operated applications using the example of Mobility-as-a-Service ticketing systems. We propose an SMR application that uses TEEs to protect sensitive customer and mobility provider data while limiting possibilities for fraud by both customers and mobility providers, and

ensuring correct billing. We find that as long as secure multiparty computation is not competitive in terms of performance, TEE-based SMR can provide significant advantages in terms of efficiency and resilience while providing reasonable confidentiality guarantees. We describe the characteristics of the Mobility-as-a-Service use case and identify similar use cases from other domains, e.g., central bank digital currencies, allowing us to conclude that our findings generalize.

In the second step, we establish the foundation for a comprehensive analysis by proposing and proving TEE-Rider, the first hybrid fault-tolerant, asynchronous, and DAG-based atomic broadcast protocol. TEE-Rider builds upon the DAG-Rider protocol family and an optimized, DAG-aware, and TEE-based causal order broadcast we propose and prove. We then identify fundamental issues that arise from the combination of TEEs and asynchrony in BFT SMR. These are the impossibility of a fault-tolerant setup and the impossibility of garbage collection. Furthermore, we prove that for partially synchronous, TEE-based reliable broadcast it is impossible to reinitialize a TEE after a crash without relying on the participation of all n replicas. We conclude the theoretical contributions with the proposal of the NxBFT SMR framework. Following an assumption-algorithm co-design, NxBFT is built upon TEE-Rider for the "Not eXactly Byzantine" (NxB) operating model to maximize throughput without sacrificing resilience. Moreover, NxBFT leverages SMR state transfer to circumvent the limitations imposed by TEEs and asynchrony and provides, under the assumption of partial synchrony, garbage collection, recovery, and reconfiguration.

Finally, we contribute an extensive empirical evaluation. To this end, we develop the ABCperf evaluation framework focusing on the fair and straightforward comparison of fault-tolerant SMR and agreement protocols. We investigate the performance characteristics of NxBFT and find that cryptographic operations for signature creation and verification are the main bottleneck. We compare the performance of NxBFT with the state-of-the-art leader-based, hybrid fault-tolerant protocols MinBFT and Chained-Damysus and investigate the impact of the SMR client model (BFT vs. NxB), payload sizes, network sizes, network latencies, and crash faults. While all algorithms can benefit from the NxB client model, NxBFT achieves the highest throughput in all scenarios with up to $\sim$ 500 000 requests per second. All algorithms show an improvement of the end-to-end latency when using the BFT instead of the NxB client model. When small latencies are required, MinBFT and Damysus are at an advantage with Damysus showing competitive throughput and impressively low latencies for small deployments. In contrast to leader-based approaches, NxBFT's performance is almost not impacted when actual crash faults occur.

Contents

1 Introduction

One key focus of system engineers and researchers is to build dependable services: Dependability, as defined by the International Electrotechnical Commission (IEC), is the "ability [of a system or organization] to perform as and when required" [Int15]. A computer system, however, is known to eventually fail caused by a wide range of possible root causes, so-called faults [Avi+04]. Such faults range from simple hardware faults, natural disasters, and human errors to malicious behavior causing performance degradation, limited functionality, or full service outages [Gun+16]. Frequency and severity of failures dominantly determine the confidence in the dependability of a service. Dependability can be improved by relying on fault tolerance: a fault-tolerant system continues to provide a correct service in the event of hardware and software faults [Avi+04; Ros+21].

Over the last 20 years the concept of resilience has evolved: Resilience is no longer solely a property of a technical system that indicates the systems ability to tolerate faults regardless of their intention [Avi+04, Sec. 5.2.2]. Rather, it is the ability of an organization (operating a technical system) to *withstand any stressful situation* without massive restrictions and, above all, without long-term consequences [AV11; RB11; Str12; Ros+21]. The U.S. National Institute of Standards and Technology (NIST) defines resilience as "the ability to anticipate, withstand, recover from, and adapt to adverse conditions, stresses, attacks, or compromises on systems [...]" [Ros+21, p. 1]. In this spirit, resilience is a crucial building block for dependability but does not cover all dependability aspects (e.g., safety and functional correctness). We informally define resilience as follows:

The property of a computer system

1. to offer a service with consistent quality, even under unfavorable conditions, and
2. to quickly recover in the case a quality degradation could not be prevented.

One approach for achieving resilience is to use a distributed ledger. A **distributed ledger** is a record of ordered client requests that is replicated to a set of processes also known as replicas [Kan+21; Int24]. Requests are ordered and added as the result of an agreement process. The ordered requests are forwarded to a higher level (business) application, the actual service, triggering state transitions. While the distributed ledger appears to be a single atomic entity, it is operated on a number of physical computers. Distributed ledgers are divided into permissionless and permissioned ledgers [WG18; Int21]: In a permissionless setting, as it is the case for most prominent cryptocurrencies, everyone is allowed to participate in the agreement process that forms the ledger. In a permissioned setting, the focus of the work at hand, there is a fixed and predefined number of replicas, that are allowed and, typically, required to actively participate in the agreement process.

The idea of a permissioned ledger is not new: A permissioned ledger essentially resembles the **state machine replication (SMR)** approach [Lam78; Sch90] in which the transaction history, i.e., the ledger, is input as operations to an arbitrary state machine. SMR employs agreement algorithms to be able to tolerate a predefined class of faults, i.e., the fault model. Distributed systems research distinguishes several fault models ranging from benign faults like crashes to potentially malicious behavior, i.e., Byzantine faults. In the Byzantine fault model, faulty processes are assumed to deviate arbitrarily from the protocol. A **Byzantine fault-tolerant (BFT)** agreement primitive tolerates Byzantine faulty processes with the fault tolerance defining their maximum number (e.g., $n > t$ [LSP82; CVL10], $n > 2t$ [LLR02; CNV04], or $n > 3t$ [PSL80; CL02] with n being the total number of processes and t being the maximum number of faulty processes).

The notion of a permissioned ledger may seem to be equivalent to a distributed database [Rua+21]. A permissioned ledger, however, is designed as a *decentralized* instead of a distributed system with processes ideally being *operated by independent parties* [Int24, Sec. 3.20]. This design choice is also known as "political decentralization" [But17]. Those parties may have limited trust in each other [WG18], implying that, in general, it is not sufficient to be able to tolerate rather simple faults like crashes. Distributed ledgers are also chosen for increased fairness and sovereignty [FH16; Ern+23]. When looking at the governance of shared resources, such as public transport platforms, monopolies discriminate against competitors and customers [Bun22; Bun23]. Instead of setting out rules and governance in contracts alone, a distributed ledger enables the technical enforcement of the same ("law is code") [FH16; PFR20; Roz+21]. Permissioned ledgers based on BFT SMR appear to be the perfect fit for consortia: the workload is distributed among all consortium members and agreed-upon rules are enforced while severe faults and attacks can be tolerated.

However, BFT SMR comes at a high cost. BFT algorithms have a significant coordination and communication overhead making their implementation complex and limiting the achievable throughput in dependence on n and t. One of the biggest problems of a BFT algorithm is to tolerate **equivocation** [Chu+07]. According to Merriam-Webster [Mer25], an equivocal statement is "usually used to mislead or confuse". In the context of agreement primitives, equivocation means that a party sends different messages to different parties in the same context where a correct party would send the same message to all parties. Without additional assumptions, the system has to consist of at least $n > 3t$ replicas and multiple rounds of communication are required to identify equivocal messages [LSP82; CL02]. Required replica count and required communication clearly drive up operating costs. Moreover, the additional coordination required in case of actual faults, e.g., as it is the case for classic algorithms with a static leader, results in additional performance degradation [Ami+24; BTR24] and may ultimately lead to a complete failure of the algorithm [LD24]. Ultimately, the aforementioned issues in terms of operation cost, achievable throughput, and actual resilience lead to BFT SMR not being widely used or even recommended [Kle16, p. 305]. To address the performance and resilience issues of BFT SMR, research followed multiple directions [Ber+23], for example, optimistic algorithms [Kot+07], streamlined algorithms [Yin+19], asynchronous algorithms [Mil+16], leaderless algorithms [Cra+18], alternative communication topologies [NMR21], and the hybrid fault model [Ver+11].

While providing significant improvements for fault-free cases, most of the optimizations still rely on distinguished replicas or leaders. Such distinguished replicas, however, can become both a bottleneck and a single point of failure [Cra+18; Yin+19; Gra22]. The focus of this dissertation is to investigate the combination of asynchrony and the hybrid fault model to design a practical, resilient, and efficient BFT SMR algorithm. Asynchrony increases the achieved resilience of algorithms because it allows progress to be made more independently of the network quality. Moreover, in asynchrony there cannot be a leader thereby preventing the leader to become a bottleneck and a single point of failure. As described above, the Byzantine fault model requires $n > 3t$ replicas to tolerate t faulty replicas. The hybrid fault model adapts the fault assumptions to be able to achieve an improved resilience for BFT SMR, i.e., a fault tolerance of $n > 2t$, thereby significantly decreasing the required replica count and, thus, operation cost. The literature shows that the hybrid fault model can be combined with other optimization techniques (e.g., streamlined [Dec+22a], tree topologies [Liu+19], optimistic approaches [Ver+11; DCK16]). The combination with asynchronous approaches, however, has largely been unexplored.

While asynchronous SMR was long considered impractical, HoneyBadgerBFT [Mil+16] and the DAG-Rider family [Kei+21; Dan+22; Spi+22; Spi+24; Aru+25] showed that asynchronous atomic broadcast, an agreement primitive suitable for the coordination between replicas in SMR, is practical and rather simple to implement. The DAG-Rider atomic broadcast family uses a directed acyclic graph (DAG) and a causal order broadcast as central components and demonstrates both impressive throughput and resilience. Throughput benefits from the inherent parallelism of the leaderless design that, for atomic broadcast, allows each replica to propose unique requests per decision. The fact that algorithms following the DAG-Rider idea use reliable broadcast as the only communication primitive make them a promising candidate for the combination with the hybrid fault model.

Hybrid fault models have been investigated for at least two decades [CNV04]. In a hybrid fault model, processes are equipped with a trusted subsystem that is assumed to only fail by crashing. With the advent of hardware-based Trusted Execution Environments (TEEs), e.g., Intel Software Guard Extensions (SGX) [McK+13; CD16], that allow confidential and integrity protected execution of code, the assumption of a trusted subsystem became realistic and led to a plethora of research in the field of hybrid fault models. Approaches either execute significant parts of the agreement algorithm within the TEE (e.g., CCF [How+23]) or they use a service deployed to the TEE combining digital signatures with monotonic counters (e.g., MinBFT [Ver+11]). Both variants prevent the creation of valid but equivocal messages. For asynchronous, DAG-based algorithms, however, an *efficient* transformation allowing actual deployment, and a thorough investigation are missing.

Unsurprisingly, the use of a TEE does not come for free. First, it must be recognized that if the assumption of a trusted subsystem does not hold, the SMR guarantees inevitably break. Second, we observe that although many proposals for TEE-based systems exist, only few address practical deployability. Most proposals lack procedures for setup, crash recovery, and garbage collection. Third, we will show that TEEs actually make crash recovery harder. Crash recovery is, however, a pivotal feature when designing a resilient SMR system as all

hardware eventually breaks [Dis21]. In summary, with this dissertation we pursue the following central research question:

What is the achievable performance and resilience of DAG-based, hybrid fault-tolerant state machine replication and under which preconditions can the leaderless nature be safely exploited to maximize throughput?

We approach an answer to the question in three parts. First, we propose an SMR-based, privacy-enhancing, and fair federated ticketing platform for public and shared transport services. We show that such a setting benefits from the use of a TEE not only for increased performance and fault tolerance but also for privacy and utility reasons. We argue that in such a highly regulated setting an attack on the TEE seems unlikely and its benefits clearly outweigh the potential risks. Moreover, the ticketing platform is an example of a use case in which the client will know when a replica gives incorrect responses. This eliminates the need for client-side broadcasting of requests, which we identify as a bottleneck in SMR, and renders voting on responses obsolete.

In a second step, we lay the theoretical foundations of an asynchronous, DAG-based, hybrid fault-tolerant SMR algorithm by answering the following research question:

What optimizations can be applied to DAG-Rider when operating in the hybrid fault model?

To design TEE-Rider, we prove that DAG-Rider [Kei+21] itself is omission fault-tolerant as long as $n > 2t$ and a BFT reliable broadcast is used. By doing so, we deliver evidence for a long-standing belief in the research community that omission fault-tolerant SMR in asynchrony with a fault tolerance of $n > 2t$ can be efficiently implemented. Moreover, we provide the necessary TEE for the hybrid fault-tolerant causal order broadcast, a hybrid fault-tolerant common coin required to circumvent the FLP impossibility [FLP83], and optimize the communication complexity. We build upon this result to propose the NxBFT SMR algorithm designed for the "Not eXactly Byzantine" (NxB) operating model. NxBFT and NxB are an assumption-algorithm co-design focused on resilience, throughput, and practicality: While, in general, operating in a Byzantine environment, we assume that operators do not tamper with the business logic of the application. The partially relaxed fault model of NxB allows maximum utilization of the inherent parallelism of TEE-Rider, thereby increasing throughput significantly. Moreover, by requiring partial synchrony for SMR state transfer, we overcome fundamental issues with garbage collection and crash recovery. We argue that the NxB model is a realistic and useful assumption for settings like the ticketing platform.

Lastly, we empirically investigate performance and resilience of NxBFT in comparison to state-of-the-art proposals to pursue the following research question:

What are the trade-offs of DAG-based, hybrid fault-tolerant state machine replication in comparison to the static leader and the streamlined paradigms?

To this end, we develop the ABCperf experiment framework with a focus on a fair comparison of SMR approaches. ABCperf can emulate client behavior depending on the business application selected for the benchmark as well as omission faults and network latency. By conducting micro benchmarks, we investigate the impact of implementation level design choices like the common coin implementation and the selection of hash functions on the performance of NxBFT. We select MinBFT (static leader) [Ver+11] and Chained-Damysus (streamlined) [Dec+22a] as reference points in an extensive experiment study: We investigate the impact of the number of replicas (up to 40), the network round trip latency (up to 150 ms), the client model (NxB vs. BFT), the payload size, and crash faults on the maximum achievable throughput as well as the latency of the three paradigms. While all algorithms can benefit from the NxB client model, NxBFT achieves the highest throughput in all scenarios. When small latencies are required, MinBFT and Damysus are at an advantage with Damysus showing competitive throughput and impressively low latencies for small deployments. In contrast to leader-based approaches, NxBFT's performance is almost not impacted when actual crash faults occur.

In summary, the contributions of this dissertation are

- an SMR application for **privacy-preserving ticketing for public transport federations** with increased fairness for mobility providers and customers,
- the **design, proof, and analysis of TEE-Rider**, an asynchronous, hybrid fault-tolerant atomic broadcast algorithm,
- the **design and evaluation of NxBFT**, a pragmatic and performance-oriented SMR protocol designed for the "Not eXactly Byzantine" (NxB) operating model with automated setup, garbage collection, crash recovery, and impressive throughput, and
- a **thorough throughput-latency trade-off analysis** for state-of-the-art hybrid fault-tolerant SMR algorithms using ABCperf, a novel evaluation framework.

The remainder of this dissertation is structured as follows. We give a background on state machine replication (SMR), cryptographic primitives, fault models, timing models, and the hybrid fault model in Chapter 2 where we also systemize SMR-related agreement primitives. Please note that SMR research spans a time range of about 40 years and Chapter 2 only contains the fundamentals necessary for this dissertation. Related work is discussed in each chapter with a focus on the chapter's contribution. In Chapter 3, we discuss requirements and approaches for federated ticketing platforms and present our approach using TEEs and SMR. In Chapter 4, we design and prove TEE-Rider and NxBFT. In Chapter 5, we present ABCperf, our experiment design, and the results of our empirical analysis. We discuss method, results, future work, and draw a conclusion in Chapter 6.

1.1 Publications and Open Source Software Contributions

Parts of this dissertation have been published in previous works:

- **M. Leinweber**, N. Kannengießer, H. Hartenstein, A. Sunyaev. "Leveraging Distributed Ledger Technology for Decentralized Mobility-as-a-Service Ticket Systems". In *Towards the New Normal in Mobility: Technische und betriebswirtschaftliche Aspekte*. 2023, pp. 547–567. [Lei+23]
- **M. Leinweber**, H. Hartenstein. "Brief Announcement: Let It TEE: Asynchronous Byzantine Atomic Broadcast with $n \geq 2f + 1$". In *37th International Symposium on Distributed Computing (DISC 2023)*. 2023, 43:1–43:7. [LH23]
- T. Spannagel, **M. Leinweber**, A. Castro, H. Hartenstein. "ABCperf: Performance Evaluation of Fault Tolerant State Machine Replication Made Simple: Demo Abstract". In *24th International Middleware Conference Demos, Posters and Doctoral Symposium*. 2023, pp. 35–36. [Spa+23]
- **M. Leinweber**, H. Hartenstein. "Not eXactly Byzantine: Efficient and Resilient TEE-Based State Machine Replication". In *CoRR* abs/2501.11051. 2025, pp. 1–13. [LH25]

Moreover, I contributed to the following peer-reviewed papers (∗: equal contribution):

- M. Grundmann ∗, **M. Leinweber** ∗, H. Hartenstein. "Banklaves: Concept for a Trustworthy Decentralized Payment Service for Bitcoin". In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2019*. 2019, pp. 268–276. [GLH19]
- **M. Leinweber** ∗, M. Grundmann ∗, L. Schönborn, H. Hartenstein. "TEE-Based Distributed Watchtowers for Fraud Protection in the Lightning Network". In *Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2019 International Workshops, DPM 2019 and CBT 2019*. 2019, pp. 177–194. [Lei+19]
- J. Schiffl, M. Grundmann, **M. Leinweber**, O. Stengele, S. Friebe, B. Beckert. "Towards Correct Smart Contracts: A Case Study on Formal Verification of Access Control". In *SACMAT'21: The 26th ACM Symposium on Access Control Models and Technologies*. 2021, pp. 125–130. [Sch+21]
- L. Gebhardt, **M. Leinweber**, F. Jacob, H. Hartenstein. "Grasping the Concept of Decentralized Systems for Instant Messaging". In *WiPSCE '22: The 17th Workshop in Primary and Secondary Computing Education*. 2022, 10:1–10:6. [Geb+22]
- L. Gebhardt, **M. Leinweber**, T. Michaeli. "Investigating the Role of Computing Education for Informed Usage Decision-Making". In *WiPSCE '23: Proceedings of the 18th WiPSCE Conference on Primary and Secondary Computing Education Research*. 2023, 29:1–29:2. [GLM23]

During my time as a doctoral candidate, I founded the ABCperf project[1] and contributed to the following open source software projects:

- T. Spannagel, **M. Leinweber**, A. Castro. "ABCperf" (experiment framework for performance evaluation of SMR frameworks as described in Chapter 5)
- **M. Leinweber**, T. Spannagel. "Damysus" (Rust implementation of the Chained-Damysus SMR algorithm as described in [Dec+22a])
- T. Spannagel, A. Castro, C. Teichmann, B. Stoyan, **M. Leinweber**. "MinBFT" (Rust implementation of the MinBFT SMR algorithm as described in [Ver+11])
- **M. Leinweber**, T. Spannagel. "NxBFT" (SMR algorithm as described in Chapter 4)
- M. Haller, T. Spannagel, **M. Leinweber**, M. Raiber. "SpliTEE" (Rust library for TEE-based threshold cryptography).

1.2 Acknowledgements

This dissertation was supported by funding from the German Federal Ministry of Education and Research within the project *KASTEL_SVI*, by the topic *Engineering Secure Systems* of the Helmholtz Association (HGF), and the KASTEL Security Research Labs.

I would like to thank my advisor Prof. Dr. Hannes Hartenstein for his support and guidance and Prof. Dr. Rüdiger Kapitza for co-refereeing my dissertation. I would also like to thank Prof. Dr. Franz J. Hauck, Dr. Christian Berger, Maya Kuhlmann, and Dr. Richard von Seck for valuable discussions and feedback during my time of working on this dissertation. I must not forget to thank my colleagues of the DSN research group, especially Matthias Grundmann, Florian Jacob, Patrick Spiesberger, and Oliver Stengele, for many and fruitful discussions, my student assistants, especially Tilo Spannagel, and the bachelor and master students I advised. Moreover, I would like to thank Markus Raiber and Valerie Fetzer who helped me to understand the cryptographer's vocabulary as well as the people around Klara Mall from SCC-NET for their support in setting up our experiment cluster.

Mein größter Dank gilt meiner Familie, allen voran meiner Frau und meinem Sohn, die mich in den letzten, sehr herausfordernden Jahren unendlich unterstützt haben.

[1] `https://github.com/abcperf/abcperf`, archived at DOI: `10.5281/zenodo.16995732`

2 Fundamentals

The aim of this dissertation is to design practical, efficient, and resilient services based on state machine replication and Trusted Execution Environments (TEEs). In this chapter, we give the required fundamentals from the area of state machine replication, secure distributed systems, and agreement primitives. First, in Section 2.1, we introduce the concept of state machine replication and its connection to fault tolerance and agreement primitives. Then, in Section 2.2, we recap the cryptographic primitives used in this dissertation. In Sections 2.3 and 2.4, we describe the prevalent fault and timing models in which distributed algorithms are evaluated. In Section 2.5, we introduce TEEs and how they can be used to increase the resilience of agreement primitives. Finally, in Section 2.6, we systemize the agreement primitives used in this dissertation, reliable broadcast and atomic broadcast, their internal relation as well as their relation to consensus, and their optimal resilience levels depending on fault and timing models.

2.1 Fault-tolerant State Machine Replication

State machine replication (SMR) allows to operate a service tolerating faults of clients and replicas. **Clients** are processes relying on the service. **Replicas**[1] are processes offering the service [Lam78; Sch90; CL02]. Each replica maintains a copy of a deterministic **application server** (the "state machine") implementing the actual service of value. The application server consists of logic and state of which both may change over time; the state machine can be a Turing-complete computer program. Clients may *request* to execute an **operation** leading to a state transition in the application server. Replicas coordinate the execution of operations to keep the application servers in a consistent state.

A replica and a client can either be **correct**, meaning they are currently following the protocol, or **faulty**, meaning they are deviating in some way from the protocol. Faults that may occur are defined by a **fault model**; faults range from simple crashes to malicious behavior (see Section 2.3). Typically, the fault model limits the number of simultaneously faulty replicas t to a fix quotient of n, the total number of replicas, while the number of faulty clients is unlimited. The **timing model** describes how long processes require to compute and transmit their messages (see Section 2.4). This is of high importance

[1] The literature uses the terms replica, process or processor synonymously. We use the term replica in the context of SMR and process in the context of agreement primitives. In both cases, we use p as an identifier.

when using time itself to coordinate: While good knowledge on timings can significantly simplify algorithms, typically the impact when such assumptions break is severe.

When all correct replicas start in the same initial state and the response of the application server is solely determined by the requested operation and the operations executed before, fault tolerance is achieved by ensuring the following two properties [Sch90; CL02; Abr19a; Ale+22; HH24]:

- **Liveness:** If a correct client requests an operation, the operation will eventually be executed and the client receives a response.
- **Safety:** Every correct replica executes the exact same order of operations.

A concrete instance following the SMR paradigm is called **replicated state machine (RSM)**. We say that an RSM is *safe and live* when the assumptions of fault and timing model are not violated. The safety property leads the distributed system to appear at its system boundaries (i.e., the externally state observed by clients) like a single central system [HW90] while the liveness properties ensures that correct clients may use the promised service. Achieving safety and liveness requires to deploy some ordered broadcast mechanism between the replicas that ensures (1) that all correct replicas learn of all valid requests received by any correct replica and (2) that all correct replicas execute the exact same order of operations. Typically, this is achieved by using an **atomic broadcast** (see Section 2.6) primitive. Atomic broadcast provides guarantees only between the replicas [Abr19a].

To provide a meaningful service for the clients, the atomic broadcast is part of an **SMR framework** that is deployed to all replicas *and* clients and enforces guarantees for correct clients. The SMR framework is generic and can be combined with any application. The selected SMR framework and, thus, the deployed atomic broadcast, defines the fault tolerance of an RSM under given fault and timing models. The generic SMR framework architecture is depicted in Figure 2.1. In the following, we will describe a full request-response cycle of a client request.

1. The application composes the request, typically containing the requested operation and authorization information, and forwards the request to the SMR client module.
2. The SMR client module distributes the request to the SMR server modules of the replicas. Depending on the assumptions of the SMR framework, a client may be required to broadcast its request instead of using a unicast to a single replica.
3. The SMR server collects the request and, to increase performance, composes multiple requests to a batch (or block) that is then forwarded to the atomic broadcast. If the request was already forwarded or answered, it is not forwarded again.
4. The atomic broadcast ensures that the same block is received by all correct replicas and that all correct replicas assign the block to the same position in the history of blocks.
5. Once the atomic broadcast can guarantee that all correct replicas will handle the block in the same way, i.e., it decides or delivers the block, the block is forwarded to the SMR server module.

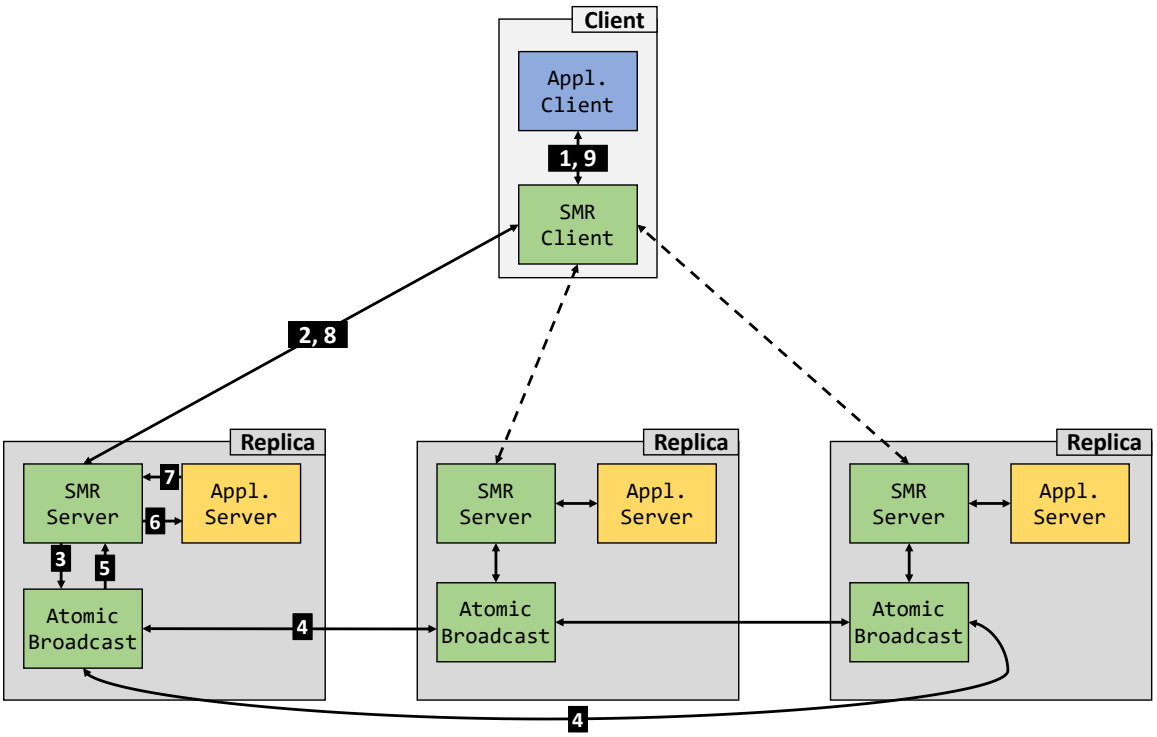


Figure 2.1: Generic state machine replication (SMR) framework architecture with three replicas and a single client. An arbitrary application (blue, yellow) is made fault tolerant using an SMR framework (green). A write request is processed as follows: After composing the request in the client application (1, blue), depending on fault model and concrete implementation, the SMR client unicasts or broadcasts (dashed lines) the request (2). The replicas then invoke the consensus layer (3) in form of an atomic broadcast. In multiple rounds of communication (4), the atomic broadcast decides on the order of the request. Once decided (5), the SMR server forwards the request to the server-side replicated application logic and state (6, yellow) actually applying the request and computing a result (7) which is then forwarded to the SMR client (8). When the SMR client received sufficient consistent responses (depends on the fault model), the response is forwarded to the client application (9).

6. The SMR server removes requests from the block that were already decided in previous blocks. It then forwards the block to the application server.
7. The application server executes all operations of the requests in a block in a deterministic order and forwards the execution results as responses to the SMR server. To increase execution performance, the application server can weaken the total order provided by the atomic broadcast and execute requests in parallel if it can be guaranteed that the requests do not interfere with each other.
8. The SMR server stores the response to be able to directly answer it if it was received a second time and sends the response over the network to the SMR client.
9. Depending on the fault model, the SMR client awaits one ore more, i.e., quorum many, consistent responses from the replicas before forwarding the response to the application.

The described cycle is, in general, required to handle any requested operation. However, depending on the application and assumptions, the request-response cycle may be simplified (e.g., for read-only operations) [Sch90; CL02].

2.2 Cryptographic Primitives

Cryptographic primitives are the fundament of any distributed system. Nowadays every SMR framework builds upon cryptography to achieve at least authenticated point-to-point links. If assuming malicious behavior, cryptography, foremost digital signatures, can be used to limit the capabilities of a possible attacker allowing to increase the fault tolerance or to simplify algorithms. The power of Trusted Execution Environments, a dedicated focus of this dissertation, comes with their ability to keep computation confidential. We give a quick recap on the cryptographic primitives used based on [CGR11; KL14; Aum17].

Cryptographically Secure Pseudorandom Number Generator (CPRNG) Randomness, or more generally speaking entropy, is the fundament of any cryptographic primitive. It is required to generate secret keys and nonces. A CPRNG is a function that produces outputs uniformly distributed on the codomain with two properties: forward and backward secrecy. Forward secrecy states that it is impossible to reconstruct previous outputs whereas backward secrecy requires that it is not possible to correctly predict future outputs with a non-negligible probability.

Hash Functions Cryptographic hash functions are a crucial building block to build authentication schemes for arbitrary-length messages. A cryptographic hash function H is a one-way function (i.e., pre-image resistant) taking a bit string x of arbitrary length and mapping it to a bit string y of fixed length (the digest). This means that H compresses x and makes it impossible to calculate x when only knowing y and H. The security-wise more relevant property is collision resistance that states that for two inputs x and x' the probability that both are mapped to the same y, i.e., $H(x) = H(x') = y$, is negligible. Widely used hash functions are the `SHA-2` [Nat15a] and `SHA-3` [Nat15b] families.

Digital Signatures A digital signature scheme ensures the authenticity and integrity of a message. The scheme consists, overly simplified, of the two functions $\mathrm{sgn}(\cdot)$ and $\mathrm{vfy}(\cdot)$. Given two parties Alice and Bob, Alice wants so send Bob a message m such that only Alice could have defined its context. Alice now signs the message by using her secret key sk (also known as private key) producing a signature $\sigma = \mathrm{sgn}(sk, m)$ and sends both signature and message to Bob. As public key cryptography is computationally expensive, messages are hashed before the signature is computed or verified. Bob knows Alice's public key pk and, thus, can check if $\mathrm{vfy}(pk, m, \sigma) \stackrel{?}{=} 1$. If the digital signature scheme used is secure there is only a negligible probability that a third party forged the signature. A digital signature provides non-repudiation and is transferable: If Alice kept her secret

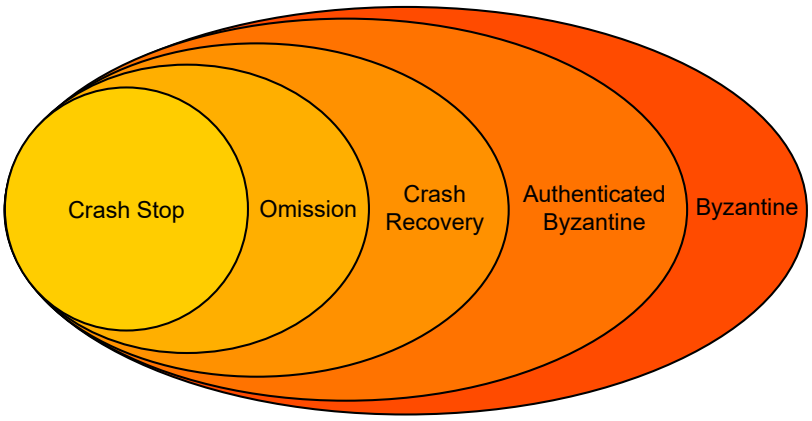


Figure 2.2: Hierarchy of fault models (extension and adaption of [CGR11, Figure 2.3]). A darker color implies that a model is a superset of models in lighter colors: Crash Stop ⊂ Omission ⊂ Crash Recovery ⊂ Authenticated Byzantine ⊂ Byzantine. An algorithm proven correct for a fault model is also correct in all subset fault models.

key private, no one but Alice could have signed the message and *every* party knowing the public key can verify the signature. Both properties are very useful when limiting Byzantine behavior. Examples for widely used signature schemes are `ECDSA` [Nat23a] and `EdDSA` [Ber+12; Bre+21]. A public key infrastructure (PKI) distributes public keys and links them to identifiers like domain names.

Authenticated Encryption and Secure Point-to-Point Links An authenticated encryption scheme encrypts a message between two parties Alice and Bob and ensures the authenticity of the ciphertext. Let Alice and Bob share a common secret key k. If now Alice wants to send an encrypted message m to Bob, she will call $(c, t) = \text{encrypt}(k, m)$ with c being the ciphertext and t the authentication tag. Bob calls $m = \text{decrypt}(k, c, t)$. If the ciphertext was altered, the function will yield an error. The standard for authenticated encryption is the block cipher Advanced Encryption Standard using the Galois counter mode (`AES-GCM`) [Nat23b]. Using the combination of digital signatures, a PKI, and a key exchange protocol like Diffie-Hellman [Bar+18], an authenticated and confidential channel, i.e., a secure point-to-point link, can be established.

2.3 Fault Models

A fault model captures the types of faults an algorithm designer assumes when designing a distributed algorithm. The correctness of agreement primitives is proven under the assumption of a fault model. The fault models can be arranged in an order (Figure 2.2). An algorithm proven correct for a fault model is also correct for all subset fault models. The fault models Crash Stop, Omission, and Crash Recovery form the group of **benign faults**. In the following, we describe the classic fault models relevant for this dissertation based on [CGR11, Section 2.2].

Crash Stop In the crash fault model, a faulty replica is assumed to potentially crash at an arbitrary point in time, i.e., it does not send anymore messages. A faulty replica can crash while being in any state according to the protocol being executed. A crashed replica cannot recover from its fault, i.e., a crashed replica is no longer part of the system. Real-world deployments, however, require a way to recover from crashes as all hardware eventually reaches its end of life [Dis21].

Omission The omission fault model addresses recoverable faults on the network layer. In the omission fault model, a faulty replica may fail to send or receive a message. In contrast to crash faults, this faulty behavior can be temporally limited. Messages between two correct replicas are not influenced.

Crash Recovery The crash recovery model is an extension to the omission fault model. Besides loosing messages in transition, faulty replicas may also loose state. A faulty replica can become correct again using a recovery procedure that enables state transfer or retransmission. Consequently, a replica may forget messages it already received or sent and an algorithm needs to be able to handle duplicate messages. Please note that a concrete implementation may not have a recovery procedure but still is crash recovery fault-tolerant.

Byzantine A replica is assumed to be Byzantine faulty when it can deviate arbitrarily from the protocol. Byzantine behavior includes any malicious behavior and collusion. What makes the assumption of Byzantine behavior especially challenging is **equivocation**: Not only can a replica actively lie, but the replica can lie in different ways to different peers [Chu+07; Cle+12]. Equivocation makes it impossible to implement any agreement primitive with a fault tolerance of $n \leq 3t$ when assuming Byzantine faults [PSL80; LSP82]. Preventing equivocation alone, however, does in general not help to increase the fault tolerance [Cle+12].

Authenticated Byzantine Authenticators, as originally defined by Pease et al. [PSL80, Section 5], can be used to limit the misbehavior of faulty replicas in the Byzantine fault model. An authenticator enforces that a faulty replica cannot alter a message when it relays it. While authenticators do not prevent equivocal statements [Cle+12], they allow to increase the fault tolerance of agreement primitives in selected cases (see Section 2.6). Authenticators can be implemented asymmetric cryptography in the form of digital signatures that offer non-repudiation and are transferable. A symmetric message authentication code (MAC) cannot be used to implement authenticators.

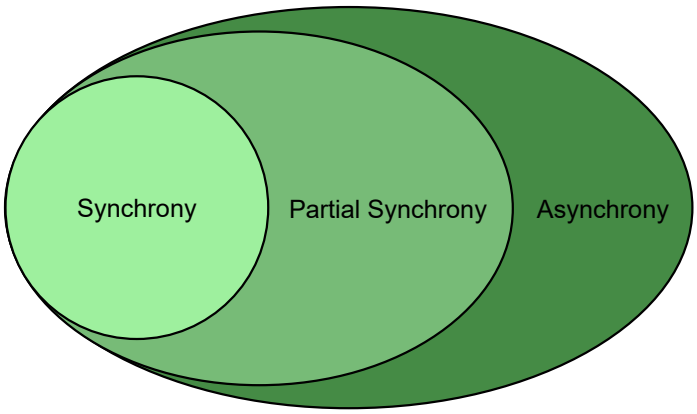


Figure 2.3: Hierarchy of timing models. A darker color implies that a model is a superset of models in lighter colors: Synchrony ⊂ Partial Synchrony ⊂ Asynchrony. An algorithm proven correct for a timing model is also correct for all subset timing models.

2.4 Timing Models

The timing assumption, i.e., the time required for messages to arrive at and be handled by replicas, is crucial knowledge when designing and proving a distributed algorithm. Good knowledge of time can be used to use time itself for coordination: If a message should arrive within a certain time span but does not, a replica can be assumed to be faulty. As it is the case for fault models, timing models can be arranged in an order as well (Figure 2.3). An algorithm proven correct for a timing model is also correct for all subset fault models. In the following, we describe the timing models relevant for this dissertation based on [CGR11, Section 2.5].

Synchrony In a synchronous system there exist two known upper time bounds Δ_T for message transmission, i.e., the maximum communication delay, and Δ_C for computation, i.e., the maximum time for processing an incoming message and computing a possible answer. Typically, the synchrony assumption is simplified to a single Δ. Synchronous algorithms use timeouts (multiples of Δ) to determine if a peer is faulty. Relying on synchrony inevitably implies that safety properties may break if the synchrony assumption is not met [Cas00] and it is hard to choose a reasonable Δ: If the chosen value is too small, rather simple network quality degradations may break the assumption. If the chosen value is too high, faults may be detected too late impacting performance negatively.

Partial Synchrony To address the issues of the synchronous timing model, the notion of partial synchrony was defined [DLS88]. Informally speaking, it is assumed that the system is only synchronous most of the time. Literature defines at least three different partially synchronous flavors: global stabilization time (GST), unknown latency (UL) and weak synchrony. In the GST model, a Δ exists and is known to all replicas but Δ only holds after an unknown stabilization time passed. Thus, replicas can never know if Δ holds. In the UL model, a fixed Δ exists but is unknown. Weak synchrony as defined

by Castro and Liskov is similar to the UL model except that Δ is not fixed but must be in polynomial dependence on the wall clock time [Cas00; Mil+16]. Algorithms in the UL model use some form of exponential back-off to increase the current timeout value whenever a timeout is triggered. Eventually, the timeout will be bigger than Δ. This behavior adapts the timeout to the worst case and potentially impacts the common case performance negatively [Mil+16; Abr24]. Modern proposals typically rely on the GST model [Abr24]. Partially synchronous algorithms only rely their liveness properties on synchrony. If chosen timeouts are too small, the system may come to a temporal halt but the safety properties are still guaranteed. In general, an algorithm for the UL model is corect in the GST model as well. The opposite is only true for so-called "time-agnostic" properties, i.e., such properties that are only proven based on the relative ordering of events but not on elapsed time [Abr24; Con+24].

Asynchrony In the asynchronous model, there is no bound on communication and computation latency. Because time cannot be used to coordinate, replicas are forced to solely rely on incoming messages to decide when and how to continue. Fischer, Lynch and Paterson [FLP83] showed with their infamous FLP impossibility result that deterministic consensus is impossible in asynchrony when having a single crash fault implying the impossibility for all fault models. The FLP impossibility states that a deterministic consensus primitive cannot be live in asynchrony as there exist traces, i.e., sequences of message transmissions and the resulting state transitions, in which a replica observes a tie vote and it has not sufficient information to break the tie with its own vote forcing the replica to wait indefinitely. Ben-Or [Ben83] and Rabin [Rab83] showed that the use of randomized protocols circumvents the FLP impossibility. A randomized protocol has only a probabilistic liveness property. If an attacker could manipulate the network between two correct replicas in a way that messages could get lost without notice, the two replicas would suffer from the "Two Generals Problem" [AEH75; Gra78]: In asynchrony, there is no way to use a finite series of acknowledgments to ensure the delivery of a message. Any asynchronous agreement protocol relies on the assumption of eventual delivery requiring that any message sent between correct replicas may be arbitrarily delayed but will eventually arrive.

Perfect Links The assumption of eventual delivery is sometimes called a fair-loss link. On a fair-loss link the probability for successful message transmission is non-zero [CGR11, Subsection 2.4.2]. Such a link can be used to implement a perfect link [CGR11, Subsection 2.4.6], a typical assumption in distributed systems [CGR11, Subsection 2.4.7]. A **perfect link** ensures reliable delivery, i.e., messages between correct processes are never lost, and no duplication, i.e., a message is never delivered twice. A **first-in first out (FIFO) perfect link** additionally ensures that messages are delivered in the order the sender hands them to the link [CGR11, Module 2.11] In practice, this FIFO perfect links can be achieved using reliable protocols like TCP [Edd22] that itself build upon adaptive routing protocols achieving fair-loss [CGR11, Subsection 2.4.7].

2.5 TEEs and the Hybrid Fault Model

While typically a process is subject to a single fault model, e.g., the Byzantine fault model, researchers are investigating hybrid fault models for at least 20 years [CNV04; Bes+23]. In a hybrid fault model, a process is equipped with a subcomponent that is subject to a different fault model than the process itself: A process may be assumed to be Byzantine faulty but a trusted subsystem – which must be used to create valid protocol messages – is assumed to be crash stop faulty. The hybrid fault model is *not* a benign fault model.

Such a trusted subsystem can be used to enforce **non-equivocation** and **transferable authentication**: A process cannot send two messages with different contents to different peers in the same context when a correct process would send the *exact same* message to all peers and *all* processes can verify the authenticity of a message [Chu+07; Cle+12]. Peering processes rely on the fact that the subcomponent is not faulty in the same way as the process itself. This dependence allows to increase the fault tolerance of an agreement primitive and to simplify agreement algorithms: BFT SMR can be implemented with a fault tolerance of $n > 2t$ when using a crash stop subcomponent [CNV04].

The advent of **Trusted Execution Environments (TEEs)** made it possible to implement practical hybrid fault model algorithms [Lev+09; Ver+11]. A TEE is rooted in a trustworthy hardware component that isolates security-critical code and data, sometimes referred to as a security kernel [JGS83], from the rest of the system. Industry has proposed several differing architectures that can be – under the loss of precision but precise enough for the focus of this dissertation – simplified into the following categories: secure boot, confidential virtual machines (VMs), and enclaved execution [SAB15; JSS20; Gep+22; Sch+22a; Li+23].

Secure boot ensures that only *expected* code can boot a system. "Expected" means that the code is either signed by a trusted authority or that a specific *code identity* is expected. Code identities are typically defined by a hash of the code (also known as measurement). Furthermore, most solutions provide a way to attest the identity of the code to a remote party. Examples for secure boot are ARM's TrustZone [PS19] and the Trusted Platform Module (TPM) [Ber+06; Tru25][2]. Secure boot can be the basis to implement a security kernel supporting higher-level functionality like attestation or encrypted persistent storage ("sealing") [SAB15]. Confidential VMs focus on the protection of VMs from the hypervisor and other VMs currently running on the same cloud host [Eis+25]. Examples are AMD's Secure Encrypted Virtualization (SEV) [KPW21] and Intel's Trust Domain Extensions (TDX) [SMF21; Akt+23]. In contrast to confidential VMs, enclaved execution aims for the protected execution of application layer software which means that all privileged software including the operating system is not trusted. The most prominent example are Intel's

[2] The TPM also provides features for storing and using cryptographic keys, e.g., for signing, and was used by the very first practical hybrid fault model agreement protocols [Chu+07; Ver+11].

Software Guard Extensions (SGX) [McK+13; CD16] that are widely used in academia (e.g., [Arn+16; Liu+19; How+23]) and industry (e.g., at Fortanix[3] and Azure[4]).

According to Anderson [And20], the **Trusted Computing Base (TCB)** "is defined as the set of components (hardware, software, human, …) whose correct functioning is sufficient to ensure that the security policy is enforced, or, more vividly, whose failure could cause a breach of the security policy." In the context of the hybrid fault model, the size and tasks of the trusted subsystem define the TCB. When instantiated with a hardware-based TEE, the TCB is the TEE itself and the code running "inside". To keep the attack surface as small as possible and increase the dependability of the trusted subsystem, a small TCB is preferred [McC+08; Lin+17]. Moreover, most TEEs have a performance overhead due to context switches, encryption tasks, or limited resources [LMR12; Mof+18; Ngo+19]. In conclusion, it is favorable to deploy only as much functionality as required to the TEE and minimize the number of context switches between the TEE and the untrusted process for security and performance reasons.

As Intel SGX allows very small TCBs while still having an acceptable performance overhead and great flexibility [Lin+17; PVC18], in this dissertation we use a TEE model derived from Intel SGX as the basis for our algorithms. Intel SGX is also used for the empirical evaluations. In the context of this dissertation a TEE is informally defined as follows:

Definition 2.1 (Trusted Execution Environment (TEE), informal)**.** A TEE is a trusted subsystem of a process and provides the ability to execute arbitrary code in an **enclave**. The TEE guarantees the following properties:

- **Enclaved Execution:**
  - **Confidentiality:** Data inside the enclave cannot be read from the outside.
  - **Integrity:** Code and data inside the enclave cannot be altered from the outside.
- **Attestation:** The TEE can certify the identity of an enclave, i.e., the code that is currently being executed, to a remote party. Attestation can be used to bind arbitrary data, e.g., a public key or a nonce, to the enclave identity.
- **Sealing:** An enclave can confidentially and authentically export/import data.

The attestation property is crucial for the use of TEEs in distributed systems. When combined with enclaved execution, peering processes can rely on the fact that the code executed inside the enclave is the same code that was attested to them. Thus, a peering process knows all possible branches of the code; enclaved execution prevents Byzantine behavior. The process hosting the TEE, however, can still be Byzantine faulty and prevent that messages sent to the enclave are actually received by the enclave and that messages created by the enclave are actually sent to the peers. Thus, when executing a complete omission fault-tolerant algorithm inside an enclave, the algorithm is trivially compiled to

[3] `https://www.fortanix.com`, accessed 2025-05-30

[4] `https://azure.microsoft.com/products/managed-ccf`, accessed 2025-05-30

withstand Byzantine faults. Minimizing the TCB is one focus of current research and this dissertation, as we will discuss in Chapter 4.

2.6 Systematization of Agreement Primitives

In the context of SMR, two agreement primitives are of particular relevance: reliable broadcast and atomic broadcast. In this section, we first give a short introduction to consensus as agreement and consensus are overlapping terms. We then define both agreement primitives and name their optimal fault tolerance depending on fault and timing model.

2.6.1 The Idea and the Term of Consensus

Consensus is a rather overloaded term; there exist multiple definitions for agreement problems all going under the name of consensus [MHS11; GK20]. Historically, **consensus** refers to the problem in which each of the n processes – out of which up to t may behave faulty – may propose a value and all correct processes must agree on the same value selected from the set of proposals:

- **Agreement:** If two correct processes decide values v_1 and v_2, then $v_1 = v_2$.
- **Termination:** All correct processes eventually decide a value.

This definition lacks a so-called validity property and can be trivially fulfilled: every process can decide on a fixed and pre-defined value v. The exact formulation of the validity property, however, defines the difficulty of the problem. Literature distinguishes between several different validity definitions in the context of *Byzantine* fault tolerance[5] of which we give three examples strictly increasing in difficulty [MHS11]:

- **Weak Unanimity (Weak Validity):** If all processes are correct and propose the same value v, all processes decide v.
- **Strong Unanimity (Validity):** If all correct processes propose the same value v, all correct processes decide v.
- **Strong Validity:** If a correct process decides value v, v was proposed by a correct process.

Strong unanimity consensus and atomic broadcast require to handle equivocation correctly to be able to tolerate Byzantine faults. The mechanisms required to achieve this fault tolerance make them equivalent to each other [MHS11][6]. Strong unanimity consensus

[5] For benign faults, the non-malicious setting allows a single validity definition: If a process decides value v, v was proposed by a process. [AAM10; MHS11]

[6] Equivalency means that an atomic broadcast algorithm can be reduced to strong unanimity consensus and vice versa.

	Synchrony	Asynchrony
Crash Stop	$n > t$ [PT86]	$n > t$ [HT94]
Omission	$n > t$ [PT86]	$n > t$ [HT94]
Hybrid	$n > t$ [CVL10]	$n > t$ [CVL10]
Auth. Byzantine	$n > t$ [LSP82]	$n > 3t$ [Cle+12]
Byzantine	$n > 3t$ [LSP82]	$n > 3t$ [BT85]

Table 2.1: Optimal fault tolerances of Reliable Broadcast for different fault and timing models.

is impossible to achieve in partial synchrony with a fault tolerance of $n \leq 3t$; the hybrid fault model cannot improve its fault tolerance since, with $n < 3t$ processes, malicious processes have the ability to outvote correct processes [Xu21, Section 2.4]. However, as we will see in Section 2.6.3, atomic broadcast has the validity requirement that a message from a correct process will eventually be delivered. As long as this property is fulfilled, i.e., faulty processes cannot mute correct processes indefinitely, partially synchronous atomic broadcast can be implemented with a fault tolerance of $n > 2t$ in the hybrid fault model [CNV04].

2.6.2 Reliable Broadcast

While consensus and atomic broadcast have n senders, reliable broadcast differs: There is only a single sender allowed to propose a value. The goal is that all correct processes eventually deliver the same value proposed by the sender. Based on [CGR11, Module 3.13], [Cac+01], and [Kei+21], we define reliable broadcast as follows:

Definition 2.2 (Reliable Broadcast)**.** A sender $p_s \in P, P := \{p_1, \ldots, p_n\}$ can **r_broadcast** tuples (c, m) where $c \in \mathbb{N}$ is a *tag* and m an arbitrary message. Correct processes **r_deliver** tuples (c, m) satisfying the following properties:

RB-Agreement: If a correct process delivers (c, m), then every other correct process eventually delivers the same (c, m).

RB-Validity: If a correct sender broadcasts message m with tag c, then every correct process eventually delivers (c, m).

RB-Integrity: For each tag $c \in \mathbb{N}$, a correct process delivers at most one message.

To be more precise, Definition 2.2 defines a reliable broadcast channel, i.e., no "one-shot" primitive as originally defined in [BT85] but closely related to the tagged version in [Bra87]; the multi-shot primitive can be composed to implement atomic broadcast and consensus [Bra87; Cac+01; LLR02; MHS11]. For simplification, we use the term reliable broadcast in the following. Since a reliable broadcast instance has only a single party that is allowed to have initial input, reliable broadcast is not equivalent to any consensus definition and is not impacted by the FLP impossibility [BT85]. It is, however, possible to have multiple instances of reliable broadcast for different senders running concurrently (i.e., each of the

n processes is a sender in its own reliable broadcast instance). The protocols investigated in this dissertation rely on this property. Table 2.1 summarizes the optimal fault tolerances of reliable broadcast for different fault and timing models. Except for the authenticated Byzantine fault model, the optimal fault tolerances are the same for synchronous and asynchronous systems. In synchrony, the correct use of digital signatures suffices to maximize the fault tolerance [LSP82] while in asynchrony non-equivocation *and* digital signatures are required to achieve the same [CVL10; Cle+12][7].

2.6.3 Atomic Broadcast and State Machine Replication

Atomic broadcast is an extension of reliable broadcast that allows multiple processes to broadcast messages and deliver them in the same order. When implementing an RSM, an atomic broadcast is deployed to all replicas. The replicas use the atomic broadcast to ensure that all replicas execute *all* validly requested operations in the *same* order. As we focus on asynchronous systems and the FLP impossibility applies to atomic broadcast (atomic broadcast is a consensus variant), we give a probabilistic definition of atomic broadcast based on [CGR11, Module 6.3], [Cac+01], and [Kei+21] as follows:

Definition 2.3 (Atomic Broadcast)**.** Processes $p \in P, P := \{p_1, \ldots, p_n\}$ can **a_broadcast** tuples (c, m) where $c \in \mathbb{N}$ is a *tag* and m an arbitrary message. Correct processes **a_deliver** tuples (p, c, m) satisfying the following properties:

AB-Agreement: If a correct process delivers (p, c, m), then every other correct process eventually delivers the same (p, c, m) with probability 1.

AB-Validity: If a correct process p broadcasts m with tag c, then every correct process eventually delivers (p, c, m) with probability 1.

AB-Integrity: For each process $p \in P$ and for each tag $c \in \mathbb{N}$, a correct process delivers (p, c, m) at most once.

Total Order: Let t_1 and t_2 be any two valid tuples that are delivered by any two correct processes p_i, p_j. If p_i delivers t_1 before t_2, then p_j delivers t_1 before t_2.

Table 2.2 summarizes the optimal fault tolerances of atomic broadcast for different fault and timing models. For each combination, we try to give the oldest applicable result. Note that we try to not use results for consensus primitives (e.g., [DLS88] for partial synchrony) as we want to ensure that we give a correct result for atomic broadcast and not a stronger or weaker primitive. Typically, the fault tolerance of atomic broadcast is also the fault tolerance of SMR building on it. In the synchronous model, however, when assuming either omission faults, hybrid faults, or authenticated Byzantine faults (marked

[7] Please note that [CVL10, Algorithm 1] does not use the muteness failure detectors described in [CVL10, Sec. 2.1] and, thus, does not rely on any timing assumption. The wormhole described by the authors is the trusted subcomponent. The correctness of [CVL10, Algorithm 1] is proven in the extended version of the paper [CVL09, Sec. A.1]

	Synchrony	Partial Synchrony	Asynchrony
Crash Stop	$n > t$ [Ray02] ○	$n > 2t$ [Abr23]	$n > 2t$ [AAM10] ○
Omission	$n > t$ [Ray02] ▹$\not\$$	$n > 2t$ [Lam98]	$n > 2t$ [AAM10] ▹
Hybrid	$n > t$ [Ray02] ⊕$\not\$$	$n > 2t$ [CNV04]	$n > 2t$ **[this work]**
Auth. Byzantine	$n > t$ [PSL80] ◇$\not\$$	$n > 3t$ [CL99]	$n > 3t$ [Cac+01]
Byzantine	$n > 3t$ [PSL80] ◇	$n > 3t$ [Cas00]	$n > 3t$ [CNV06]

Table 2.2: Optimal fault tolerances of Atomic Broadcast for different fault and timing models. Cells marked with $\not\$$ do not directly apply for SMR. Other symbols mark results originally not for atomic broadcast but applicable (explanations in Section 2.6.3): ○ = crash stop fault-tolerant consensus, ▹ = omission fault-tolerant consensus, ⊕ = fault model compiler, ◇ = interactive consistency.

with $\not\$$), SMR requires at least a fault tolerance of $n > 2t$ [Abr19b] which is not the case for atomic broadcast. If we are not aware of a result for atomic broadcast, we use results that also apply for atomic broadcast for different reasons. Those results are marked with the following symbols:

- ○ = crash stop fault-tolerant consensus. Crash fault-tolerant atomic broadcast can be reduced to crash fault-tolerant consensus [MHS11].
- ▹ = omission fault-tolerant consensus. Omission fault-tolerant atomic broadcast can be reduced to omission fault-tolerant consensus [MHS11].
- ⊕ = fault model compiler. Clement et al. [Cle+12] and Ben-David et al. [BCS22] showed that any crash fault-tolerant agreement protocol can be compiled to withstand faults in the hybrid fault model without a change of the fault tolerance.
- ◇ = interactive consistency. Byzantine fault-tolerant atomic broadcast can be reduced to Byzantine fault-tolerant interactive consistency [CNV06].

We can see that except for synchrony and crash stop failures, SMR always relies on quorum-based voting to prevent inconsistencies. In synchrony, the use of digital signatures suffices to achieve a fault tolerance of $n > t$ for atomic broadcast and $n > 2t$ for SMR which is not the case if faults cannot be reliably detected using timing assumptions (i.e., in partial synchrony or asynchrony). In those cases, non-equivocation *and* digital signatures (i.e., the hybrid fault model) are required to achieve the maximum fault tolerance of $n > 2t$ for atomic broadcast and SMR.

3 Use Case: Mobility-as-a-Service Ticketing

Mobility-as-a-Service (MaaS) ticketing platforms are a promising use case for a consortium-operated application. They require the integration of and computation on data from multiple providers, such as public transport providers, ride-sharing services, and infrastructure providers, to offer customers a seamless travel experience. These systems face significant challenges regarding privacy and fairness. Personal data, e.g., travel patterns, and trade secrets, e.g., service utilization, must be protected. Furthermore, centralized approaches can lead to unfair advantages for certain providers. In the following, we analyze how a TEE-based SMR application can address these challenges.

In Section 3.1, we introduce the concept of Mobility-as-a-Service. In Section 3.2, we motivate issues concerning privacy, fairness, and scalability, and define the objectives for a privacy-preserving and fair MaaS ticketing platform. In Section 3.3, we discuss and analyze different architectural approaches for privacy-preserving solutions by reviewing related work from the domains of confidential databases, location-based services, confidential SMR, and secure multi party computation. In Section 3.4, we present our TEE-based SMR application design for an MaaS ticketing platform. Finally, in Section 3.5, we discuss the achieved properties, opportunities for optimization, the characteristics of our design and its potential for other application, and the impact of broken TEEs. We conclude that (1) that our findings generalize to other use cases than MaaS ticketing and (2), as long as secure multiparty computation is not competitive performance-wise, TEEs are an important building block for modern, confidentiality-aware distributed applications.

This chapter is an extension to previous work by Leinweber, Kannengießer, Hartenstein, and Sunyaev [Lei+23].

3.1 Introduction to Mobility-as-a-Service

As the name suggests, Mobility-as-a-Service (MaaS) is in line with the cloud-enabled "as-a-service" paradigm [Jit+17; Cal+18]. MaaS provides a platform that integrates various mobility services by different **mobility service providers (MSP)**, such as public transport (e.g., busses, trams, trains), micromobility sharing (e.g., bikes, scooters), car sharing, and ride hailing (i.e., taxis and taxi-like services), into a single service offering[1]. By increasing

[1] MaaS is a so-called multi-modal service as it combines different modes and means of transportation.

the convenience, and thus attractiveness, of shared mobility services, MaaS is expected to be a key enabler for the transition to sustainable and carbon-neutral mobility [Jit+17; Eck+18; CS20b]. This integration allows **customers** to plan, book, and pay for their entire journey through a single application, enhancing the user experience and promoting the use of public transport and shared mobility services. Examples are Gaiyo[2] in the Netherlands, Jelbi[3] in Berlin, Floya in Brussels[4], and Regiomove[5] in Karlsruhe.

Typically, MaaS breaks with established advance payment tariff schemes and, instead, offers subscription-based and pay-as-you-go tariffs with features like fare capping. This paradigm shift is deemed to be more flexible and user-friendly than classic tariff schemes [Jit+17; CS20b; PBS22]. An MaaS platform may combine subscription-based tariffs and pay-as-you-go tariffs: The subscription-based tariff is used for the basic mobility services, an example is the "Deutschlandticket"[6], while the pay-as-you-go tariff is used for additional services, like sharing services or ride hailing. Karlsruhe's public transport authority "Karlsruher Verkehrsverbund" follows this scheme within Regiomove[7].

According to the International Association of Public Transport (UITP) and the Smart Ticketing Alliance, the *number one challenge* for MaaS is a functional, interoperable, and trusted ticketing platform [PBS22, p. 28]. In fact, a well-designed ticketing platform is "a great enabler of MaaS" [PBS22, p. 11]. Pay-as-you-go tariffs rely on the ability to record travel information, i.e., the start, end, and type of a trip, to calculate the fare. While this information theoretically could be recorded on the side of the customer, MaaS typically follows the pattern of account-based ticketing [LV17; PBS22; Ack24]. Account-based ticketing is based on an extensive **back office** [PBS22, p. 16]. In computer science terms, the back office is a backend database system that records customer and travel-related information for the purpose of physical access control to vehicles, fare calculation, invoicing, revenue distribution, and analytics. The back office is of important value for the MSPs: As the back office logically centralizes all travel-related information, MSPs have real-time access to service utilization and customer behavior which can be used to (automatically) optimize their services, pricing, and passenger routing [CS20b; Ack24]. The back office is part of the MaaS platform (also including front-end software for customers and MSPs) typically operated by an "MaaS operator" [PBS22].

3.2 Objectives

In the following, we derive the objectives for a privacy-preserving and fair MaaS back office that still follows the needs of the MSPs in terms of utility and scalability.

[2] `https://gaiyo.com/?lang=en`, accessed 2025-06-09
[3] `https://www.jelbi.de/`, accessed 2025-06-09
[4] `https://www.floya.com/`, accessed 2025-06-09
[5] `https://www.kvv.de/mobilitaet/regiomove.html`, accessed 2025-06-09
[6] `https://en.wikipedia.org/wiki/Deutschlandticket`, accessed 2025-06-10
[7] `https://www.kvv.de/service/kvv-webshops/deutschlandticket-shop.html`, accessed 2025-06-10

Utility and Privacy Account-based ticketing is a location-based service known for processing and storing potentially highly sensitive information: People's movement trajectories, i.e., sequences of spatiotemporal information, are rather unique and can be used to derive information on wealth and income, health, sexual orientation, and political and religious views [GKP11; Mon+13; Dra+19; Kap22]. In 2019, the "Berliner Beauftragte für Datenschutz und Informationsfreiheit" (in English approximately Data Protection and Freedom of Information Officer of Berlin) has officially objected to the Jelbi platform due to a lack of technical data protection [Dat19, Chapter 4.1]. This demands the back office of an account-based ticketing approach to respect and maintain a high level of privacy. However, the back office also has to be useful for the MSPs. The MSPs need to be able to access the data in the back office to optimize service, pricing, and passenger routing [Kap22; Ack24]. Due to the need for revenue distribution in an MaaS context, MSPs inevitably have to share data with each other in an approachable and efficient way, demanding high interoperability of involved technology stacks [CS20b; PBS22]. While this data exchange is paramount for MaaS, it poses a significant risk for business secret leakage (e.g., financial gain through industry espionage) [Cal+18; Gar+23]. Mechanisms that protect the privacy of customers and MSPs may severely limit the utility in terms of analytics and may lead MSPs to reject the system [Kap22]. Potential customers, however, may reject the system when strong privacy guarantees cannot be made [PPT20]. In conclusion, an MaaS approach deployed within the European Union must comply with the General Data Protection Regulation (GDPR) [EC16] not solely by policy but also by technical measures following the "privacy by design" principle [Dat19; Cot20] and "create an ecosystem for data sharing accepted by all parties involved" [PBS22, p. 30], i.e., *bridge the gap between protection and utility*. The back office must hide spatiotemporal information of customers and business secrets of MSPs from each other while still allowing for useful analytics. Metadata tracing, e.g., using IP addresses, is an orthogonal problem which is not in scope of this work.

Fairness and Dependability Due to the nature of their business in terms of investment risk (e.g., for infrastructure and vehicles), operation cost, and customer-expected dependability, MSPs are typically large companies with significant market power (if not monopolies) or publicly owned entities [FFF07; PPT20]. This centralization can lead to unfair advantages of established MSPs over smaller MSPs [Bun22; Bun23]. Additionally, the centralization may increase the market entrance barrier for new MSPs. If now MaaS platforms are operated or controlled by such central parties, which is to be expected [LV17; PPT20], the required success of the MaaS paradigm for modal shift and sustainable transport may be hindered. A successful adoption of the MaaS paradigm comes with the risk of insider threats, i.e., federation members and customers may manipulate or spoof input data for their own benefit [Che+17; Cal+18]. Finally, federation members may refuse to have their computing infrastructure available and accessible which comes with the risk of denial of service attacks. We require the back office to be *fair* by preventing central and powerful parties, by being resilient against insider threats, and by technically enforcing the governance of the MaaS platform ("law is code" [FH16]). Moreover, the back office must be *dependable* by avoiding single points of failure.

Scalability and Interoperability If MSPs were required to implement one-to-one custom logic, interfaces, and contracts for each MaaS platform or peering MSP, this again would increase the market entrance barrier significantly and, as customers would be required to have accounts on multiple platforms, also lower customer convenience [Cal+17; PPT20]. The "The European Roadmap 2025 for Mobility as a Service" by the "Mobility as a Service for Linking Europe" project of the Conference of European Directors of Roads (CEDR) states that "national and international interoperability is needed" [Eck+18]. Consequently, there must not be a plethora of MaaS platforms or MaaS standards. Instead, there should be a single standard defining a single, *commonly operated* MaaS platform (which directly limits the impact of central parties). Thus, an MaaS platform must be able to *seamlessly integrate* with existing and future MSPs in Europe. Statista Market Insights [Sta25a] prognoses approximately 303 million users of public transport for 2029 in the EU-27, i.e., the 27 states that are forming the European Union in 2025. If we assume that all users use a single MaaS platform for their daily commute, this leads to 606 million daily interactions with the MaaS platform. If we further assume that half of these interactions are equally distributed over a morning rush hour of two hours for the journey to work or education, this results in 42 000 interactions per second for the rush hour[8]. If we add a margin for staff requests (e.g., ticket inspections) and to prevent the system from operating at its limits, we require the back office to be able to handle at least *50 000 requests per second* when scaling to the European Union. The scalability requirement is not only on the number of requests but also on the number of MSPs that can be integrated into the MaaS platform. This number is hard to predict, but for the area of responsibility of a single regional transport authority, the number is easily well into the two-digit range [VBB25].

3.3 Architectural Considerations and Related Work

In the context of location-based services, researchers have been investigating the utility-privacy trade-off and the dependability of such systems for nearly two decades [Jia+22]. In this section, we analyze related work from the domains of confidential databases, location-based services, mobility services, confidential SMR, and secure multi party computation regarding their ability to fulfill the objectives derived in the previous section. We first start by reviewing centralized approaches, then we analyze decentralized approaches, and finally we conclude with a summary explaining why we deem a permissioned, TEE-based SMR application to be currently the best fit for an MaaS ticketing platform.

[8] Please note that this is a clear overestimate, as, according to Statista Market Insights [Sta25b], only around 45% of shared mobility journeys are billed via online sales channels in 2023 and, according to Eurostat [Eur21], only 27% to 45% of all distances traveled are due to commuting.

3.3.1 Centralized Approaches

In centralized approaches, a single party, the MaaS operator, controls and provides the back office service and potentially the full software stack. While it is relatively straightforward to implement the required functionality and to scale the system, e.g., through cloud-based horizontal scaling [LML14], it is trivial that a naive centralized approach can neither enforce fairness nor guarantee privacy[9]. To prevent the MaaS operator from learning movement patterns of customers, research has investigated the use of obfuscation techniques like cloaking [XC07; Al-+18], differential privacy [Kap22], and federated learning [CG24]. Obfuscation techniques work with noise or inaccuracies to hide the actual data and require that potential attackers have limited additional information (e.g., metadata traces) they could use to reverse the obfuscation. Because of the noise, obfuscation techniques may have a significant impact on the service quality. If used in the context of MaaS, all data is generated by the customers and obfuscation is applied to this data. MSP service utilization cannot be obfuscated and, thus, business secrets are not protected. Recent research [Mir+23; Buc+24] has shown that differential privacy, a technology that is currently being heavily researched, is not sufficient to meet customers' privacy demands.

Customer privacy can also be protected by means of sophisticated cryptographic protocols securing the confidentiality of transferred and stored data. Besides tailored approaches for location-based services (e.g., [SLL14]), also generic approaches like confidential databases [KJH15], e.g., based on oblivious RAM [JSS14] exist. Those approaches have the problem that, in general, complex queries for analytics cannot be executed on side of the storage provider, i.e., the MaaS operator, without revealing the data yielding additional load on the client side, possibly limited utility, and impeded scalability. TEE-based confidential databases (see, e.g, [Sar+18; PVC18; MMS25]) can circumvent this issue. Additionally, the attestation feature of the TEE can be used to enforce the governance and ensure that the MaaS operator and MSPs do not learn more than agreed. But if the assumptions for the TEE are not met, e.g., due to flaws in the TEE's hardware isolation (e.g., [Bul+18]) all confidentiality and dependability guarantees are lost. Anonymous credential systems have been proposed to hide the identity of customers when interacting with a backend system [Nab+21].

All centralized approaches have the problem that the central service is a single point of failure. Any fault on the side of the MaaS operator – no matter if it is a technical failure, a security incident, or a denial of service attack – may compromise the availability of the back office. Obviously, the MaaS operator can apply distributed computing mechanisms, i.e., redundancy, fail-over mechanisms, and fault tolerance, to increase the availability of the back office. However, this has only small impact when the MaaS operator itself is being compromised and, importantly, does not solve the requirement for fairness as the federation has only limited control on the MaaS operator's actions.

[9] This is without relying on regulation or contracts.

3.3.2 Decentralized Approaches

One key focus of decentralization in general and distributed ledgers in particular is to prevent a single point of failure and to increase the autonomy of the involved parties. The use of a distributed ledger to increase the fairness and the dependability is evident: Information systems and economics literature has shown that ledger technology can be used to enforce rules and regulations in a technical system by implementing the governance model as part of the technical agreement process (i.e., following the paradigm "law is code") [FH16; PFR20; LWS21; Roz+21; Pet22; Gre+24]. Consequently, several authors proposed the use of distributed ledgers for improved and automated governance and for ensuring functional correctness of mobility services (e.g., [KH18; NPP19; Bot+19; WZ21]), ticketing (e.g., [PE19; Lam+19; YZ22; PME24]), and location-based services (e.g., [Amo+18; Nos+20; She+20; Amo+22; Rüs+22; Guo+23]). It has to be noted that a permissioned ledger inherently is less inclusive than a permissionless ledger; consortia can be discriminatory. When powerful parties are to be prevented at all, a permissionless ledger has to be used.

Scalability is in conflict with the coordination and replication required to operate a distributed ledger. It seams to be unfeasible for every MSP to run a state machine replica due to (1) cost and required technical expertise and, more importantly, (2) due to the limited scalability of distributed ledgers: Distributed ledgers suffer from the **blockchain trilemma** (simplified) stating that it is impossible to maximize throughput without sacrificing either decentralization (i.e., high number of independent replicas) or security (i.e., fault model or fault tolerance) [MPP20; Nak+23; MK25]. If a permissionless ledger was chosen, the choice would certainly not match the requirements of a public transport infrastructure (for Europe): Permissionless ledgers are typically associated with a rather strong attacker model but throughput is significantly limited and response latency ranges from a minimum of several seconds to, in the worst case, several minutes or longer [Cro+16; EC25; Hil+23]. Fast response times come with high cost [JM25] and requests suffer from potential censorship, i.e., they are either not processed at all or only with delays, for example due to external policies [Wah+24]. Additional to the throughput and latency issues, permissionless ledgers have a non-negligible, ever-growing demand for storage space as it is typically required to store the full request history permanently [FNL22]. Research investigates the use of so-called second layer solutions [Gud+20] and sharding [ZMR18; Wan+19; Yu+20] to improve the scalability of permissionless ledgers. Second layer solutions use an overlay network formed by the clients on top of a distributed ledger. To minimize the use the of the first layer, the distributed ledger is only used as a settlement and conflict resolution layer. When using sharding, the ledger state is split into multiple shards, each of which is processed by a subset of the replicas. Typically, those approaches come with a trade-off between scalability and security [Wan+19; Gud+20; Yu+20; AMW25] or added requirements on side of the clients (e.g., being always online [Gud+20; BG22]). Permissioned ledgers based on SMR achieve sub-second latency and impressive throughput while being able to scale to a few hundred replicas and having strong security guarantees [Spi+22; Gir+24; Aru+25]. Spanner [Cor+13] and CockroachDB [Taf+20] are examples for permissioned ledgers using sharding for improved scalability.

Confidentiality and privacy, outlined above as a key objective for an MaaS back office, are even more in conflict with the nature of a distributed ledger: every replica holds a full copy of the data and, thus, every replica can access *all* data [Bes+08]. There is a vast amount of literature investigating confidentiality and privacy aspects of permissionless and permissioned ledgers respectively. Moreover, the whole area of secure multi party computation (MPC) [Has+19], tightly related to the idea of a permissioned ledger in particular and to agreement primitives in general [GL05] but coming from the cryptography research community, focuses on the protection of data from insider threats. In the following, we try to give an overview of the most relevant[10] general purpose approaches and approaches that are tailored to location-based services and mobility use cases.

Permissionless ledgers are accessible by everyone making the confidentiality issue even more severe as everyone can access the data and data cannot be deleted [FNL22; Pol+22; Bel+23]. There exist solutions extending permissionless ledgers with functionality providing privacy-preserving decentralized computation using self-sovereign identities [FSZ18; Feu+22], TEEs [Yua+18; Che+19; GLH19; Xia+20; Wu+25], homomorphic encryption [WK18], secret sharing [Ben+20; Ste+21], and payment channel networks [GZH22] but, in general, scalability issues and unfavorable trade-offs for second layer approaches still apply. Moreover, a multiparty computation, i.e., a federation computes a common public output on private inputs of the federation members, typically requires to make the clear-text data available to distinguished computation nodes or TEEs or to rely on MPC. Nevertheless, Ernstberger et al. [Ern+23] promote infrastructures rooted in permissionless ledgers to foster data sovereignty. Please note that, in general, these approaches can also be combined with permissioned ledgers. We argue that if choosing a permissioned ledger, e.g., for scalability reasons, the additional complexity of a second layer solution is not justified as confidentiality mechanisms can directly be integrated into the permissioned ledger (see below).

Permissioned ledgers limit the access to the data to the members of the federation which is a first line of defense against external threats [XXZ18; PP24]. Furthermore, permissioned ledgers allow for garbage collection [Dis21] which allows to keep data only as long as it is required for the business logic. But if the federation is formed by the MSPs, the MSPs can access all data and, thus, neither business secrets nor customer locations are protected from insider threats. Early work in the context of SMR investigated the gains of separating ordering and execution [Yin+03] (improved by [DZ16]). A recent work, Qanaat [Ami+22], follows and extends this idea for cooperating companies in a supply chain: Data is clustered and companies replicate only those clusters relevant to them. These ideas are related to sharding and aim at maximizing the number of parties that need to be corrupted to compromise the confidentiality. Sharding approaches do not protect from honest-but-curious parties, i.e., parties that do not deviate from the protocol but try to learn as much as possible about the data. In particular, the approaches do not fit the data sharing model of MaaS where it is a priori unknown which customers use which MSPs

[10] Searches for “privacy blockchain” and “multiparty computation” on `https://dblp.org` yield 1 879 and 593 matches respectively (accessed 2025-06-22).

and, thus, which data is relevant to which MSP. Bessani et al. [Bes+08] use secret sharing – also a main building block of MPC – to achieve confidentiality. The approach only supports set and read operations of client-side encrypted information. More sophisticated state transitions have to be handled by a client itself and the replicas cannot validate client-provided information which conflicts with the objective of dependability. Wang and Zhang [WZ21] propose a proxy re-encryption scheme for ride-sharing services that uses the ledger as data storage and for conflict resolution. Computations on user data on the ledger are not possible as location data is handled off-chain. In P-CFT [Li+21], clients send zero-knowledge proofs instead of the clear text data that are ordered in a crash fault-tolerant manner. Vehicloak [Guo+23] is a privacy-preserving payment scheme for location-based vehicular services based on zero-knowledge proofs. Computation is limited to money transfer; the ledger is only used for dependability reasons (there are no federating parties). While zero-knowledge proofs can be validated and invalid data is not stored, similar to [Bes+08] replicas cannot compute on the data.

This is where MPC comes into play. MPC is a cryptographic primitive that allows a set of parties to jointly compute a function on their private inputs while keeping those inputs secret from each other. MPC is a well-established field of research with a large body of literature [Has+19]. While MPC can, in theory, compute any function, some operations come with a significant performance penalty limiting the scalability both in terms of throughput and replicas [Has+19; Kel20]. POBA [Fau+25], a follow-up to [Fet+22], is a tailored MPC solution for use cases like federated ticketing. They use anonymous credentials to hide user identities when interacting with the back office. They achieve a check-in latency of ~ 3 s for 1 million customers per day and 9 replicas limiting their use to city-scale deployments. Similar to MPC, TEEs can provide high confidentiality guarantees while still allowing for complex queries and analytics. TEEs have been used to implement general purpose, confidential, and replicated databases [Bra+19; Wan+20; Yan+20] with recent advances focussing on feature-rich governance and practicality [How+23]. EventChain [Sch+22b] is an example for a tailored TEE-based replicated database for location-based services. In all works, the consensus is that the additional performance overhead is negligible compared to a native permissioned ledger. As in the centralized case, broken TEEs lead to a loss of confidentiality and dependability. MPC does not make any hardware assumptions and, thus, is considered more robust.

3.3.3 Conclusion

Table 3.1 summarizes the analysis. The analysis above and the findings of related work rule out any centralized approach as it cannot meet the objectives of fairness and dependability and proposed obfuscation and encryption techniques do not meet the requirements of customers and MSPs. Consequently, we require the use of a distributed ledger. While a permissionless ledger would be most inclusive, we find that permissionless ledgers are not a good fit for MaaS ticketing due to their limited scalability, high latency, high cost, and the risk of censorship. The permissioned model is a first line of defense against outside threats and allows for garbage collection and efficient algorithms limiting the resource

	Customer privacy	Business secret protection	Attacker model	Utility	Fairness	Dependability	Scalability
Centralized: naive	×	×	–	✓	×	◯	✓
Centralized: client-side obfuscation	◯	×	◯	◯	×	◯	✓
Centralized: client-side encryption	✓	×	✓	×	×	◯	◯
Centralized: TEE	✓	✓	◯	✓	◯	◯	✓
Permissioned: naive	×	×	–	✓	✓	✓	◯
Permissioned: sharding	◯	×	×	✓	✓	✓	✓
Permissioned: client-side encryption	✓	×	✓	×	✓	✓	×
Permissioned: multi party computation	✓	✓	✓	✓	✓	✓	×
Permissioned: TEE	✓	✓	◯	✓	✓	✓	◯

Table 3.1: Design space analysis for federated Mobility-as-a-Service ticketing applications comparing variants of centralized solutions and decentralized permissioned ledgers. Permissionless ledgers (and their possible extensions) are left out as permissioned ledgers are equal or superior in all listed categories. As long as MPC is not competitive performance-wise, the choice of TEEs offers the best trade-off.
Classification: ✓ = good, ◯ = medium, × = insufficient, – = does not apply.

requirements of the system. Furthermore, permissioned ledgers achieve strong security guarantees and sub-second latency while being able to scale to a few hundred replicas. While MPC is clearly the best fit in terms of achieved security guarantees, it is simply not feasible (yet) to scale MPC to the required number of requests per second and the number of replicas in an MaaS federation. We conclude to use a TEE-based application layer for a permissioned ledger as the back office of an MaaS federation. To maximize efficiency and minimize the trusted computing base (general purpose databases come with a full-featured SQL interface), a TEE-based database tailored to the requirements of an MaaS ticketing back office should be used. It needs to be ensured that any observable communication with the back office does not leak information about the customers' movement patterns. To this end, anonymous credentials should be considered.

3.4 SMR Application for Anonymous and Fair Federated Ticketing

We focus on the realization of a federated combined pay-as-you-go and subscription-based ticketing platform that supports check-in and check-out, invoicing and fare distribution. Our focus is on the back office and the interaction with the same which we define as an SMR application layer protocol. We first define the scenario and the system model. Then, we define the functional and security requirements of the resulting RSM followed by an

overview of the protocol. Finally, we describe the protocol in detail and analyze if and how it meets the security requirements.

3.4.1 Scenario Definition

We design an SMR application layer implementing the following scenario: A customer registers with a single MSP, called **home provider (HP)**, and creates an MaaS account. If customers want to use a mobility service, they must check in before using it. The check-in is performed using a smartphone application that interacts with a check-in/check-out **terminal**. This can happen at the vehicle (e.g., for buses at the entrance door or for rental bikes at a device on the bike) or at an entrance control (e.g., at a turnstile). After use has ended, customers must perform a check-out. For services where check-in can be bypassed (e.g., buses or trams), ticket inspection is supported. A ticket inspector can ask to see proof of check-in and check its validity.

The back office is responsible for the recording of **proofs-of-interaction (PoI)** that are check-ins and check-outs of customers. In regular pre-defined intervals (**billing period**), e.g., every month, **invoice generation and revenue distribution** is performed: The back office calculates for each customer the total fare based on the recorded PoIs and possible subscriptions according to a pre-defined billing function. Additionally, the back office creates a list of transfers to be made between MSPs in order to distribute the revenue. Each HP is responsible for collecting the fare from its registered customers including those fares that accrue for using mobility services of peering MSPs. Then, each HP distributes revenues across all federation members according to the usage of their mobility services. We define trip planning, payment, and analytics beyond invoicing and fare distribution to be out of scope.

3.4.2 Requirements

Based on the scenario definition above and the objectives derived in Section 3.2, we define functional and security requirements for the SMR application layer protocol as follows.

Functional Requirements

- **Check-in and check-out.** Customers must be able to check-in and check-out a mobility service at arbitrary valid locations. Valid locations are defined by the provider of the mobility service.
- **Ticket inspection.** Ticket inspectors must be able to verify that a customer has checked-in to the mobility service currently used.
- **Invoice generation.** The back office must be able to calculate the total fare due for a customer for a billing period. The fare must be available to a customer's HP.

- **Verifiable fares.** Customers must be able to verify the correctness of calculated fares and invoices.
- **Revenue distribution.** The back office must be able to calculate revenue shares and resulting money transfers for MSPs based on their actual service utilization. Each MSP must be able to receive the information about the amounts it owes to or receives from other MSPs.

Security Requirements

- **Confidentiality of customer identifiers.** Any MSP, including the HP of a customer, must not learn who the customer is or where the customer is registered when recording a PoI or when inspecting a ticket for invoicing, the back office may link the identity of a customer to its total fare and present it to the corresponding HP.
- **Confidentiality of spatiotemporal information.** The back office and communication with the same must not leak spatiotemporal data of customers. Ticket inspectors must not be able to re-identify customers in different inspections.
- **Confidentiality of mobility service utilization.** Any MSP, except the MSP providing the mobility service, must not learn which mobility service is being used when recording a PoI, when inspecting a ticket, or during revenue distribution. Revenue distribution may only leak revenue shares between two peering MSPs but not what actual services or customers generated those shares.
- **Correctness of PoIs.** The back office must not accept PoIs that are not valid, i.e., check-ins and check-outs must be at valid locations by a customer actually owning the identity used. Thus, customers must not be able to spoof identity or location information.
- **Correctness of fares and revenue shares.** The back office must calculate fares and revenue shares according to a pre-defined billing function.
- **Availability.** The back office must be available for check-ins, check-outs, ticket inspections, and invoicing at all times as long as a majority of replicas is operational.

3.4.3 Protocol Overview

In this subsection, we give an overview of the application layer protocol. The protocol is based on a permissioned ledger that is replicated using SMR.

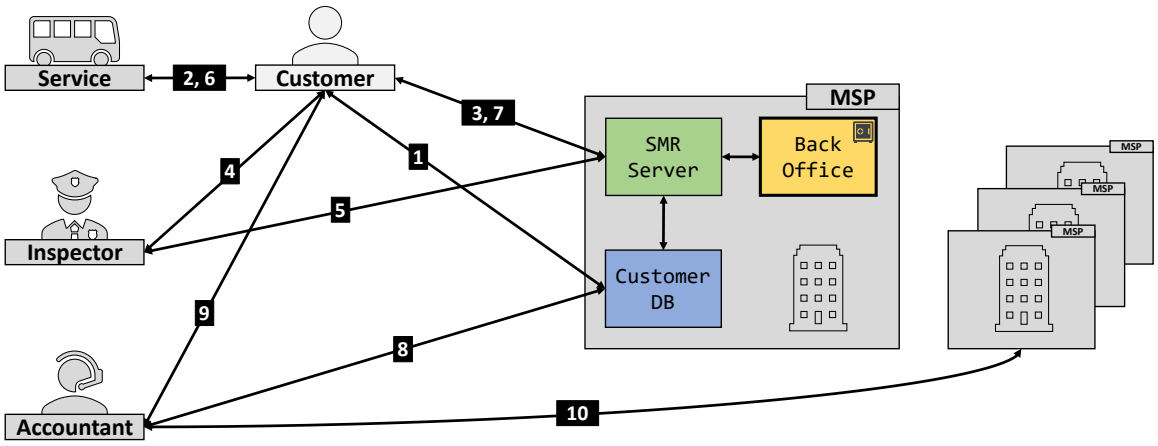


Figure 3.1: Mobility-as-a-Service ticketing architecture. Each MSP hosts a local customer database (blue) and a ticketing back office (yellow) encrypted using TEEs (safe icon, thick border) and replicated using SMR (green). The customers registers with its home provider (1) which creates an entry in its customer database. Now, the customer can use mobility services by checking in: the customer interacts with the service using their smartphone and a terminal (2) which creates a proof-of-interaction (PoI) that is sent to the replicated back office (3). On ticket inspection, the inspector asks the customer for a proof of check-in (4) which is verified by the back office (5). Check-out is equivalent to check-in (6, 7). At the end of a billing period, an accountant initializes the billing process via the customer database (8) which is executed by the back office. The back office responds with revenue distribution information and pseudonymous total fares; the pseudonyms can be mapped by the customer database. The accountant bills the customer (9). The MSPs distribute the revenue according to the revenue distribution information (10). To simplify, replica-to-replica SMR communication is left out in this figure.

System Model We assume a federation of n MSPs, each of which is a legal entity with a unique identifier. Each MSP provides a set of mobility services, e.g., public transport or sharing services, and operates a replica of the MaaS platform. MSPs may behave Byzantine (either $n > 3t$ or $n > 2t$). Each replica hosts a TEE (see Definition 2.1) that is assumed to be crash stop faulty. When a TEE crashes, it looses its state. Customers are SMR clients and may be Byzantine; correct customers behave according to the defined application layer protocol and the SMR framework. The timing assumption depends on the SMR framework. We assume a PKI that allows for the secure identification of customers, MSPs and their employees, and check-in/check-out terminals (pk identifies a public key, sk a secret key). Replicas and clients communicate via perfect and secure point-to-point links (see Section 2.2). Communication in which one endpoint is hosted inside an enclave is established using the TEE's attestation feature; the encryption is terminated inside the enclave. We propose to use near-field communication (NFC) between customers and terminals.

System Architecture and Basic Protocol Flow The architecture of the ticketing platform and the basic protocol flow is shown in Figure 3.1. Each MSP hosts a local customer database (not replicated) and an SMR-based ticketing back office. The customer database is used to store customer billing information, e.g., clear-text customer identities, addresses, and payment information. The back office implements encrypted and replicated PoI (**`PoIDB`**) and authentication (**`AuthDB`**) databases and the business logic, i.e., access control,

validity checks, the billing function for fare calculation and revenue distribution, and the governance model, inside an enclave. A concrete enclave is associated with the MSP it belongs to and carries information to authenticate as an enclave operated by the respective MSP. The basic protocol flow is as follows:

1. Customers register with their HP which creates an entry in the HP's customer database.
2. A registered customer can use the mobility services of all MSPs by checking in: the customer interacts with the service, e.g., a turnstile or other interface, which creates a PoI.
3. The check-in PoI is sent to the replicated back office which validates the PoI and stores it on success.
4. On ticket inspection, the ticket inspector asks the customer for a proof of check-in.
5. The ticket inspector uses the presented information to query the back office for the validity of the check-in.
6. Check-out is equivalent to check-in: the customer interacts with the service which creates a PoI.
7. The check-out PoI is sent to the replicated back office, validated, and stored.
8. At the end of a billing period, the accountants of all MSPs initialize the billing process which is executed by the back office. The back office calculates the total fare due for all customers of all MSPs for a billing period and the revenue shares for each MSP. The back office sends as a result to each MSP's accountant a list of pseudonymous total fares due and a list of transfers to be made between the accountant's MSP and the other MSPs. The pseudonyms can be mapped to the clear-text customer identities by the customer database.
9. The accountant bills all customers of the MSP according to their total fares due.
10. The MSPs distribute the revenue according to the revenue distribution information.

3.4.4 Protocol Description

In this subsection, we describe the protocol in detail. We assume proper authorization of MSP staff (i.e., accountants, ticket inspectors).

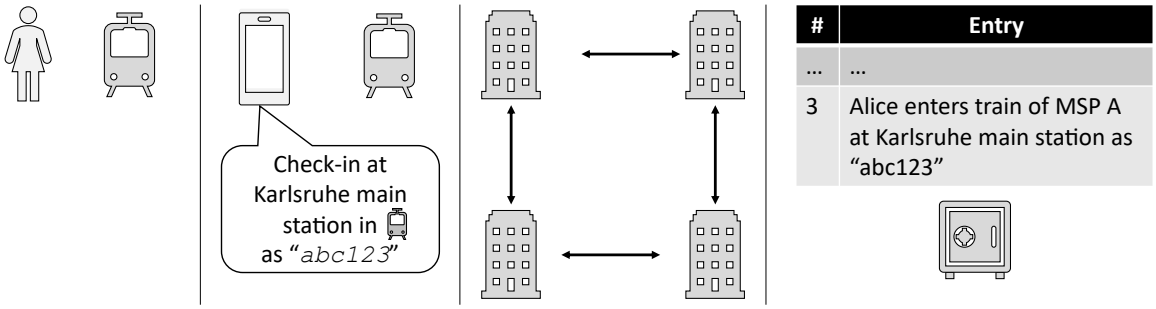


Figure 3.2: Exemplary Mobility-as-a-Service check-in protocol flow. Alice wants to use a train of MSP "A" and enters the train in Karlsruhe. She checks in using her smartphone and the terminal in the train. The terminal creates a proof-of-interaction (PoI) that Alice verifies and sends to the back office as a check-in request. The back office validates the PoI and replicates it using the SMR framework. On successful validation, the check-in becomes a new entry in the enclaved `PoIDB`.

Registration, Customer Database and Authentication Database If a person wants to use the MaaS platform, they must first register with an MSP (selected at the person's discretion). This registration makes the person a customer of the MSP and the MSP becomes the customer's HP. Each MSP maintains a local customer database that contains their customers' contract information, e.g., name, address, and payment information. The customer database is not replicated and is only accessible by the responsible MSP. Additional to the customer database entry, the registration process creates an entry in `AuthDB`: $\langle customerID, mspID, authToken, interactionToken \rangle$. The *customerID* is a unique identifier for the customer, i.e., a UUID [DPL24], that is chosen by the HP. The *mspID* is the unique identifier of the MSP where the customer registered. Both *authToken* and *interactionToken* are randomly generated and cryptographically secure secrets that are used to authenticate the customer. They are generated by the customer. The back office rejects the registration if any of the three values is not unique or does not conform to the expected format (i.e., in terms of length and character set to ensure sufficient entropy). While the *authToken* must *not* be used outside the secure channel to the back office, the *interactionToken* is designed to be transmitted in clear text and be read by MSP employees and devices, i.e., for check-in, check-out, and ticket inspection. To this end, the *interactionToken* is a one-time token that is exchanged after every use.

Check-In and Check-Out Protocol A customer *c* must check-in to a mobility service *s* in the ticket system to start a trip. Customer *c* needs to check-out when the use of service *s* ended to communicate the ticket system that their trip has finished. Both, check-in and check-out, are write operations to `PoIDB` and `AuthDB`[11]. The procedure is as follows:

1. Customer *c* sends their current *interactionToken* to terminal *s*.

[11] Please recall that both databases are managed in the enclave and, thus, benefit from the security guarantees provided by the TEE.

2. Terminal s creates a signature σ_s over the *interactionToken*, the current time *ts*, and the current location *gps* and sends the following message to customer c:

$$interaction = \langle \text{INTERACTION}, pk_s, interactionToken, ts, gps, \sigma_s \rangle$$

3. Customer c checks if they agree with current location and time and chooses a new interaction token $interactionToken'$.
4. Customer c sends $\langle \text{CHECKIN}, interaction, authToken, interactionToken' \rangle$ as a check-in request to the back office using its SMR client.
5. Each back office replica performs the following validation checks:
   a) *authToken* and *interactionToken* belong to the same customer c (lookup in `AuthDB`, yields *customerID* on success)
   b) $interactionToken'$ is unique, i.e., not used before (lookup in `AuthDB`)
   c) Customer c is currently not checked-in (lookup in `PoIDB` using *customerID*)
   d) *interaction* contains a valid signature by s
6. If the checks are successful, the back office replicas order the check-in request using the SMR framework. Otherwise the request is rejected and customer c receives a negative response.
7. On successful replication, each back office replica repeats the validation checks, adds the check-in to `PoIDB`, sets $interactionToken \leftarrow interactionToken'$ in `AuthDB`, and sends a positive response to the customer using the SMR framework.
8. In case of physical access control, e.g., for rental bikes, terminal s unlocks the service when receiving positive information from the back office.
9. Customer c collects the responses using the SMR client, uses the rules of the SMR framework to determine the result of the operation, and, on success, updates their local state with the new *interactionToken* and the information that they are currently checked-in to service s.

For check-out, the procedure is equivalent to check-in except that the message type is $\langle \text{CHECKOUT}, \cdot \rangle$ and the the validation check requires a valid check-in as the current customer state (instead of validating that the customer is currently not checked-in). Figure 3.2 shows an exemplary check-in protocol flow.

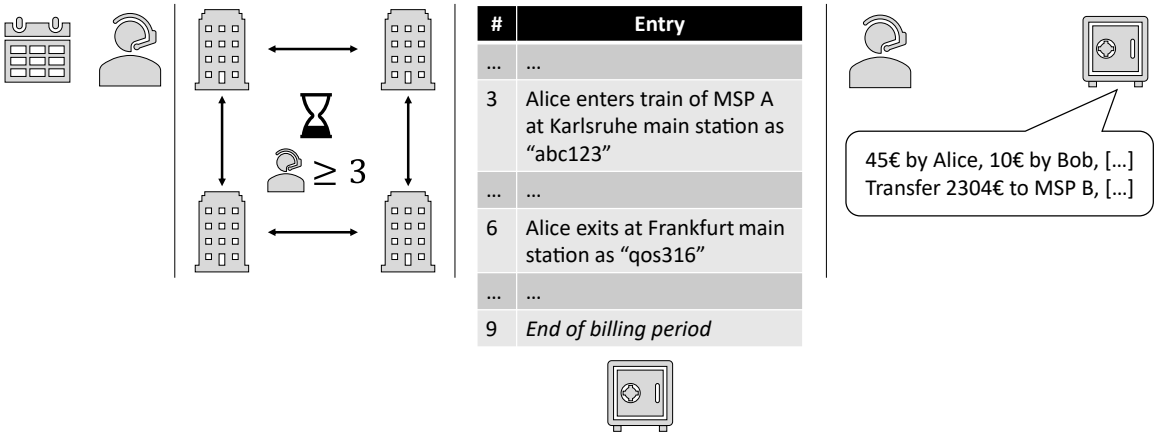


Figure 3.3: Exemplary Mobility-as-a-Service billing protocol flow with four MSPs. MSP "A" initializes the billing process by sending a conforming request to the back office. The back office waits until $n - t$, i.e., a majority of MSPs (three in the example), have sent their requests. Then, the back office calculates the total fare due for each customer and the revenue shares. Each MSP receives a list of pseudonymous total fares due and a list of transfers to be made between the MSP and the other MSPs, i.e., it neither contains fares of customers where the MSP is not the HP nor transfers where the MSP is not involved.

Ticket Inspection The ticket inspection allows a ticket inspector t to verify that a customer c has checked-in to the mobility service s currently used. The procedure is as follows:

1. Customer c provides the current *interactionToken*, i.e., the one that was established during check-in, to ticket inspector t.
2. Ticket inspector t sends request $inspection = \langle \text{INSPECTION}, interactionToken, s \rangle$ using their SMR client to the back office.
3. Each back office replica maps the *interactionToken* to the corresponding customer c identified by *customerID* using `AuthDB`.
4. Each back office replica identifies the last entry in `PoIDB` for customer *customerID* and service s identified by pk_s.
5. Each back office replica reports success if the last entry is a check-in to s and failure otherwise.
6. Ticket inspector t receives the responses using the SMR client and uses the rules of the SMR framework to determine the result of the inspection.

Invoicing and Revenue Distribution After the end of a billing period (e.g., monthly), the back office needs to calculate the total fare due for each customer and distribute the revenue across the MSPs according to the usage of their mobility services. The TEEs itself have no access to a trusted time source and, thus, the initiation of the billing process has to come from outside the back office. To prevent faulty billing processes, the back office waits for a majority of MSPs to initiate the billing process. In the following, MSP_A is an arbitrary MSP and pk_A its public key. The procedure is as follows:

1. When the untrusted part of the back office replica of MSP_A observes that the current billing period is over, the replica issues a request $billing = \langle \text{BILLING}, b_t, pk_A, \sigma_{MSP_A} \rangle$ where b_t is an identifier for the current billing period and σ_{MSP_A} a signature over the request by MSP_A.
2. The SMR framework orders the *billing* request.
3. As soon as $n - t$ validly signed billing requests for the same billing period b_t and issued by different MSPs have been ordered, the back office starts the billing process: The back office inserts a special entry into `PoIDB` that marks the end of the billing period. Subsequently, the back office collects all PoIs between the markers for b_{t-1} and b_t and inputs them to a predefined billing function.
4. The result of the billing function is unique for every MSP. The result consists of two lists: First, a billing list with entries of the form $\langle customerID, totalFare \rangle$. All *customerID*s listed must be registered with MSP_A. Second, a revenue distribution list with entries of the form $\langle fromMSP, toMSP, amount \rangle$. In the revenue distribution list, either *fromMSP* or *toMSP* has to be equal to MSP_A.

Figure 3.3 shows an exemplary billing protocol flow with four MSPs.

3.4.5 Security Analysis

In this subsection, we analyze how the protocol described above meets the security requirements defined in Section 3.4.2.

Confidentiality of Customer Identifiers The back office handles three different types of identifiers: (1) the *customerID* that is used to identify a customer in the back office, (2) the *authToken* that is used to authenticate operations of a customer, and (3) the *interactionToken* that is used to identify a customer in the context of a PoI. Two of those identifiers, *customerID* and *authToken* are static while the *interactionToken* is a one-time token. The *customerID* is static and leaves the enclave. However, it is only used (in clear text) to link a customer to their total fare when the back office sends the billing information to the customer's HP. As a HP knows its customers, this does not leak any additional information. The *authToken* is only used inside secure channels and known in clear text only by the customer and the enclaved back office. The *interactionToken* must not be secret and must be read by MSP employees and terminals. As the *interactionToken* is a one-time token, two different tokens can only be linked inside the enclaved back office.

Confidentiality of Spatiotemporal Information Spatiotemporal information is generated when a customer checks-in or checks-out to a mobility service. The terminal of the mobility service creates a PoI by signing the request of the customer. The PoI is then confidentially transmitted to and stored by the enclaved `PoIDB` of the back office. The PoIs of a customer are aggregated to a total fare due for a billing period that is revealed to the HP. The PoI

contains the one-time *interactionToken*, which, as discussed above, is not secret and can only be linked by the enclaved back office. In terms of customer privacy, each PoI is public information. During ticket inspection, the ticket inspector receives the *interactionToken*. Thus, a ticket inspector is only able to re-identify a customer[12] if the customer is inspected more than once between check-in and check-out, i.e., on the same trip.

Confidentiality of Mobility Service Utilization As it is the case for spatiotemporal information, the utilization information is generated when customers check-in or check-out to a mobility service. The PoI contains the public key of the mobility service, i.e., the identifier of the service or terminal. While the PoI is public information in terms of customer privacy, it is confidential in terms of the MSPs' business secrets. As stated before, the PoI is confidentially transmitted and stored. The only time a PoI is handled in clear text is during check-in, check-out, and ticket inspection. Those clear-text interactions are with the MSP providing the service. This MSP is allowed to learn which of its services are used how often. Revenue distribution presents aggregated information. While this hides the actual services used, it reveals how many customers of a HP used the services of a peering MSP. Please see Section 3.5 for a discussion of the impact of this information leak.

Correctness of PoIs The correctness of PoIs is crucial for the correctness of the billing process. Correct PoIs ensure that the correct customers are charged the correct fare for the mobility services used: for this, customers must not be able to spoof their identity or location information. The location information (i.e., which station or terminal, where in terms of GPS coordinates) is provided by the terminal and signed by the same. The signature ensures that customers cannot modify this information. Thus, the only entity capable of spoofing location information is the terminal itself. However, this only produces damage (in terms of lost revenue) for the MSP operating the terminal and no peering MSPs. The identity information is provided by the customer in the form of the *interactionToken*. We require the *interactionToken* to be a cryptographically secure random string. Thus, if the customer's smartphone is not compromised, an attacker can only guess the *interactionToken* with negligible probability. Furthermore, the *interactionToken* is only accepted if it as accompanied by a valid *authToken* that is unique for the customer. The *authToken* is a secret that is only known to the customer and the enclaved back office. Thus, an attacker can only present a valid *authToken* with negligible probability. Finally, a check-in will only be accepted if currently checked-out and vice-versa for a check-out. This ensures that only valid input data is provided to the billing function.

Correctness of Fares and Revenue Shares The correctness of fares and revenue shares relies on the correctness of the PoIs and the correctness of the billing function. As discussed above, the PoIs are correct. The billing function is implemented inside the enclaved back office and is only executed after a majority of MSPs have initiated the billing process. The

[12] We speak of re-identification in a technical sense. Ticket inspectors who are good at remembering faces obviously will re-identify customers especially when they use the same service regularly

voting ensures that the billing function is only executed when a majority of MSPs agree that a billing period is over. As the back office is an SMR application, the enclaved back office executes the billing function deterministically, i.e., the same input always produces the same output. The enclaved back office is assumed to be crash stop faulty. Combined with the attestation feature of the TEE that allows customers and peering MSPs to ensure code integrity, this ensures that the enclaved back office is not compromised and that the billing function is executed correctly. In conclusion, the total fare due for each customer and the revenue shares are correct at all MSPs.

Availability The back office is operated as an SMR application. Thus, it is available as long as $n - t$ of MSPs operate their replicas.

3.5 Discussion on Approach and Its Generalization

With this section, we discuss the proposal, the features achieved by using an TEE-based SMR application, limitations concerning scalability, and the impact of broken TEEs. We discuss the applicability of our approach to similar use cases and conclude the chapter.

3.5.1 Achieved Privacy, Fairness, and Dependability

As discussed above (see Section 3.3), trajectories can be used to identify a person, and, more crucially, also a person's habits. Current obfuscation techniques do not suffice to prevent this [Mir+23; Buc+24]. Instead, the idea of the presented protocol is to prevent that trajectories are accessible. To this end, spatiotemporal information is not transmitted in clear text and only stored and processed inside of an enclave. The MaaS ticketing use case, however, requires that some aggregated information, i.e., total fares due, can be learnt by an MSP. Related work [Jol+24], however, has shown that aggregated information may still leak sensitive information. The possible leakage heavily depends on the parametrization of the tariff system. If a sufficient amount of different trajectories lead to the same fare, the leakage can be minimized [Fet24, Section 3.6]. Thus, the achieved privacy depends on the actual tariff system with fare capping and subscription models increasing the privacy.

For business secrets, the possibilities for privacy-improving parametrization are more limited. The presented protocol leaks to an MSP how many service it provided to which MSP's customers. Especially, when shares are compared over time, this may allow conclusions about customer movements. Moreover, if an MSP has only a small number of services and service also change over time, this may allow conclusions on the rentability of the services. Again, the parametrization of the tariff system and the internal structure of the MSPs can help to mitigate this. The protocol does not leak any information about the customers themselves or which services are used. The revenue distribution information is an aggregation of an aggregation – a transfer amount is a sum of sums – and, thus, an added layer of indirection that makes it harder to draw conclusions.

Substantial fairness gains are achieved by the use of an SMR application layer: The protocol ensures that all MSPs and customers are treated equally. The protocol does not allow for any preferential treatment of customers or MSPs. However, the use of a permissioned model is a clear caveat: a majority within the federation can push the federation to be discriminatory. A permissionless model would allow for more inclusiveness but, if not relying on second layer solutions, would not be able to provide the same level of privacy and scalability. In conclusion, the choice between a permissionless and permissioned ledger is a trade-off between fairness/inclusiveness and scalability/privacy with the permissioned ledger being superior in all but the fairness/inclusiveness category.

The use of TEEs and SMR ensures the correct operation. Finally, the application is available as long as a majority of replicas is operational and the request load can be handled. However, SMR comes with a performance overhead which we will investigate in Chapter 5.

3.5.2 Extending Functionality

Our proposal only contains the basic functionality for a federated MaaS ticketing platform. A "full" MaaS ticketing platform, however, would require additional functionality: The analysis in Section 3.2 identified a clear demand for a solution that can be adapted (over time) to the needs of the federation. The lack of support functions limits the usability and, thus, the adoption of the system design.

Comprehensive analytics is relatively straightforward to achieve. The federation needs to agree on a set of analytics functions (e.g., service utilization by date and time or by location/station). Based on this agreement, the back office software can be extended. The attestation feature of the TEE ensures that only the agreed functions are executed. A similar case is the inclusion or exclusion of an MSP: As long as the federation supports a change, it can be deployed. Recent related work [How+23] proposed an automated change of the federation structure and even the federation rules.

The protocol depends on the customers to manage their secrets. However, in practice, customers may lose their secrets, e.g., when they lose their smartphone. The protocol can be extended to allow a customer to request a new *authToken* from their HP. The HP can then allow the customer to create new secrets. Another very common issue is that customers may forget to check-out. While, obviously, the protocol cannot prevent this, it can support the MSPs in correcting location data (e.g., after the customer authorized the MSP to do so). While such features give the HP the possibility to correct mistakes, they also introduce a risk of abuse. Such trade-offs are not unique to the presented approach but do also apply for other federated approaches, e.g., based on MPC.

Finally, the systems requires a way to recover crashed replicas to be practical as, eventually, every hardware will reach its end of life [CL02; Dis21; How+23]. This feature depends on the SMR framework used. For classic BFT SMR, checkpoints and state transfer can be used to recover crashed replicas [CL02; Dis21]. We investigate the recovery of crashed replicas in TEE-based SMR in Chapter 4.

3.5.3 Protocol Optimizations

Scaling the proposed protocol to a wide area, e.g., the European Union, comes with two rather obvious questions: Why should an MSP in Malmö replicate the data of travels in Porto? And how can SMR scale to such a high number of involved MSPs? In summary, the question is if the described approach can be optimized to fit the goal of Europe-wide MaaS ticketing. Obviously, if all MSPs in a Europe-wide MaaS federation would run a replica, the number of replicas would be too high. As discussed in the design space analysis (see Section 3.3), sharding is a possible solution to this problem. Sharding allows to partition the data and the replicas into smaller groups, i.e., shards, that can be replicated independently. For the mobility use case primarily designed for regional services, a possible solution could be to shard the data by region, e.g., by country or by state. This would limit the interaction between shards to cases where customers travel between regions. However, this approach has implications for the privacy goals (cross-region travels can be distinguished from intra-region travels) and also complicates the billing process.

Consequently, the question is if immediate ordering of all requests is necessary or if a delayed and batched ordering, e.g., only directly before billing, is sufficient. Ticketing is a form of asset transfer: the customer pays for the right to use a mobility service. Related work has shown that asset transfer does not require global consensus but only agreement between the involved parties [Gue+19]. Limiting the agreement to the involved parties comes with two downsides: (1) the billing subprotocol would be significantly more complex and (2) the privacy goals would be impacted as for every trip the assets would have to be transferred separately (Grundmann et al. [GZH22] follow such an approach). Faut et al. [Fau+25] propose to minimize the number of consensus invocations. MSPs are responsible for collecting all PoIs that accrue from the use of their mobility services. Before billing, the PoIs of all MSPs are anonymously collected and ordered. A downside of reduced consensus invocation is that analytics functions, e.g., for utilization analysis, cannot be performed in real-time.

We conclude that a consensus-based approach can simplify the billing process and improve the privacy at the cost of full replication. The number of consensus invocations is a trade-off between utility and scalability. For real deployments, future work should consider to use an approach that minimizes the number of consensus invocations while still allowing for sufficient analytic capabilities. As this significantly reduces the load on the consensus layer (e.g., one invocation every 20 minutes), it can also be used to scale the system to a very large number of MSPs. The optimization of read-only operations (like analytics functions) is not impacted by the number of consensus invocations but further reduces the number of consensus invocations [BRB21].

3.5.4 On the Impact of Broken TEEs

TEEs are used as they promise efficient and secure federated applications. Related work and the work at hand show that they can be used to build convincing SMR frameworks and SMR applications. Such a design heavily relies on the correct operation of the TEE. However,

for all major TEE implementations exist known attacks [Bul+18; Cer+20; Buh+21]. A broken TEE assumption can have a significant impact. Typically, those attacks can be mitigated by updating the TEE firmware but the risk of future vulnerabilities is not negligible. On the application layer, a broken TEE leads to the leakage of all PoIs stored and, thus, to the loss of all confidentiality guarantees. In terms of fairness, the customer can use their request history to prove their correct behavior, i.e., that they checked-in and checked-out correctly. Thus, while a broken TEE can lead to unfair treatment of customers, such conflicts can be resolved. On the coordination layer, the TEE is used to prevent equivocation. A broken TEE may lead to equivocation which can lead to a safety violation: Different replicas transition to conflicting states which leads to different output of application layer operations. Consequently, the goal for future work should be to use TEEs to boost efficiency while limiting the impact of broken TEEs to liveness violations or making the compromising of a single TEE not sufficient for an attack to be successful. CoVault [Vit+25], a very recent approach, follows this goal and combines MPC with TEEs to increase efficiency while limiting the impact of broken TEEs.

3.5.5 Characteristics and Related Use Cases

The MaaS ticketing use case has some characteristics – in terms of required features and in terms of the deployment scenario – that allow to conclude a generalization of the proposed protocol to other use cases. Those characteristics are:

Management of Shared Resources The SMR application is used to manage shared resources: the customers share the mobility services, the MSPs share the back office and, thus, the customers. The use of a distributed ledger allows for a shared view of the state of the system and for an equal treatment of all customers and MSPs.

High Demand for Privacy *and* Utility On the one hand, the SMR application handles highly sensitive data. On the other hand, this data has to be of value for the MSPs to be accepted. As the design space analysis (see Section 3.3) and the discussion above highlighted, this is a clear conflict and requires a careful design of the SMR application.

Asset Transfer and Auditability The core of the MaaS ticketing use case is the transfer of assets: the right to use the service is exchanged for a fare. The asset transfer characteristics enforces a customer to participate in all state transitions concerning their state stored on the ledger. As such, a customer is always able to know its current ledger state without contacting the ledger and the customer's request history can be used for conflict resolution as a proof of correct behavior. We conclude that, while not strictly necessary for asset transfer, a global consensus can be beneficial for utility and privacy.

Regulated Environment and Involvement of Public Parties In Europe, public transportation is regarded as critical infrastructure [EC22] and, thus, heavily regulated (e.g., [Bun22; Bun23]). As such, the operation of public transport typically involves (semi-)public parties, e.g., transportation authorities, municipalities, governments, or publicly owned companies. This setting significantly lowers the risk of a peering party to attack their TEE.

There are several endeavors by researchers and government agencies proposing or investigating the use of ledger technology for use cases sharing the characteristics above:

- **Central Bank Digital Currencies (CBDCs).** CBDCs aim at replacing or complementing cash transactions with a digital currency. CBDCs are issued or at least regulated by a central bank. The flexibility of our proposal in terms of governance decisions [Wüs+22, Section 4] as well as the achieved privacy guarantees [Eur24, Section 5.3] make it a good candidate for a CBDC. Ledger technology is considered as a possibility for the basic infrastructure of the digital euro [Eur23, Recital 64]. Microsoft promotes to use their Confidential Consortium Framework (CCF) [How+23], a TEE-based, confidentiality-preserving distributed ledger, for CBDCs [WJP24].
- **Smart Grids and EV Charging Federations.** In smart grids, energy is traded for money but, in contrast to classic power grids, the end users, i.e., private households or organizations, are consumers and producers of energy at the same time ("prosumer"). Such a system is inherently decentralized [Bun25]. With electric vehicles (EVs), the consumption is no longer bound to buildings. Instead, customers want to buy the energy wherever they need it – as it has been for combustion engine cars ever since. The modern smart grid – that is (1) reactive to current energy demands and (2) offers a seamless service for prosumers, consumers, and producers – can benefit from privacy-preserving and efficient distributed ledgers [Bao+21; Mol+21; FCV25].
- **(General Purpose) Statistics.** Many states operate agencies that collect and analyze data for statistical purposes. Eurostat pursues a project to use privacy-enhancing technologies to improve the privacy in statistical data sharing and analytics [Ric24].
- **Public Administration and E-Government.** The public administration often relies on data exchange between different agencies. A citizen's or organization's data is only of value if it is up-to-date and correct. However, stored data must only be accessible as long as an agency has the right (e.g., by law or consent) to access the data. Fatz et al. [FHF20] show the value of confidentiality-preserving data sharing on a distributed ledger at the example of tax documents. Other examples could be any publicly managed registry (e.g., civil registries) or censuses.
- **Healthcare and Medical Data Sharing.** Medical doctors rely on a comprehensive medical record to find the best treatment. However, due to strict regulations, sharing between doctors is either not feasible or cumbersome. There is a vast amount of work investigating the benefits of using distributed ledgers for healthcare data management, access control, and sharing (e.g., [Hal+21; Ami+25]).

3.5.6 Conclusion

With MaaS ticketing, we analyzed a use case of a federated application. We identified privacy, utility, fairness, dependability, scalability, and interoperability as key objectives. We found that the use of a distributed ledger is beneficial for the use case as it increases the fault tolerance and allows for a shared view of the state of the system and for an equal treatment of all customers and MSPs. The discussion and analysis of different architectural approaches showed that when not relying on TEEs, either utility, privacy, or scalability is sacrificed. We were able to show that the use of TEEs allows to achieve all three objectives and comes with the necessary flexibility to adapt the system to the needs of the federation. While the use of TEEs is not without risk, we conclude that the benefits outweigh the risks, especially when (1) operating in a regulated environment with public parties involved and (2) clients are in the position to prove their correct behavior at any time.

4 Theory of Asynchronous TEE-Based State Machine Replication

Historically, SMR protocols rely on a leader to implement an efficient agreement process [Lam98; CL02]. The introduction of a leader simplifies the fault-free agreement process in which the leader is responsible for coordination of the same. There are, however, three major drawbacks caused by the leader being a **bottleneck** (or single point of failure):

1. The leader is on the performance-critical path of every request that is to be processed [Gra22]. The leader must maintain connections to all clients, handle every request, and coordinate the ordering the requests. For the latter, the leader communicates with every replica. Hence, the leader is the bottleneck for scalability and throughput [Bie+12; VG19; NMR21].
2. When the leader becomes faulty, the system must initiate a so-called **view change** to elect a new leader [CL02]. A view change is typically a complex and expensive operation [Cra+18; Yin+19] that requires multiple communication rounds and the participation of a majority of replicas. The view change is crucial for the system's safety and liveness; its correct implementation is, however, non-trivial [BSA14; Din+17; WKM24]. Moreover, the additional overhead caused by the view change protocol limits the resilience of the protocol even to simple faults like crashes or omissions [Dan+22; Ant+23] and can, when operating close to the peak performance, lead to a metastable failure from which the system is not able to recover [Hua+22; LD24].
3. In leader-based SMR, timing information is used to identify faulty replicas (e.g., [CGR11, Section 2.6.4]). If a replica does not receive a response from the leader within a certain timeframe, it assumes the leader is faulty and triggers a view change. If this timeouts are too small, this can lead to unnecessary view changes. If timeouts are too big, this can lead to prolonged periods of unresponsiveness and degraded performance [Mil+16; Dan+22].

The focus of this chapter is to lay the *theoretical foundations for asynchronous, leaderless, hybrid fault-tolerant BFT SMR* protocols. Asynchronous BFT SMR protocols do neither rely on a leader nor on timing assumptions [Mil+16; Kei+21], thereby circumventing the leader bottleneck and making them more independent from the network quality. Related work using asynchronous, DAG-based atomic broadcast as the basis for SMR has shown tremendous throughput improvements and competitive performance under faults [Dan+22] which leads to the question whether the hybrid fault model can further

improve performance and resilience. However, while it is known that any crash fault-tolerant atomic broadcast protocol can be transformed into a BFT atomic broadcast protocol with polynomial overhead [BCS22], a *thorough* investigation of the implications of using asynchrony – positive and negative ones – when aiming at a practical, production-ready protocol with minimal overhead is missing for two reasons:

1. We observe that the combination of asynchrony and TEEs makes setup, garbage collection, and crash recovery impossible.
2. Existing hybrid fault-tolerant, asynchronous proposals [Fu+22; Xie+25] fail short in providing a thorough analysis and proper solutions for the aforementioned issues.

This chapter aims to fill this gap by providing a theoretical foundation for asynchronous BFT SMR protocols based on a TEE-based atomic broadcast protocol inspired by DAG-Rider [Kei+21]. We contribute the following:

- Design, proof, and analysis of a **communication-efficient, TEE-based reliable broadcast** protocol (Section 4.2).
- Design and analysis of two **TEE-based common coin** protocols (Section 4.3).
- Design, proof, and analysis of the **TEE-Rider asynchronous atomic broadcast** (Section 4.4).
- The finding of three **impossibility results** (Section 4.5) stating that
  - the setup of a TEE-based agreement primitive requires consensus on the enclave identities and, thus, is impossible to achieve in partial synchrony with a fault tolerance of $n < 3t$ when assuming Byzantine faults,
  - backfilling-based reliable broadcast and garbage collection are incompatible when operating in asynchrony, and
  - TEE reinitialization, i.e., the establishment of a new enclave identity for a sending process, in partially synchronous, TEE-based reliable broadcast requires active participation of *all* processes.
- The **NxBFT SMR framework** based on TEE-Rider addressing scalability issues, request duplication, setup, garbage collection, reconfiguration, and crash recovery (Section 4.6).

We give an overview on related work in Section 4.1.

The proof of TEE-Rider was in parts published by Leinweber and Hartenstein [LH23]. The PRNG-based common coin, the optimized broadcast, the impossibility result for recovery, and NxBFT were previously published by Leinweber and Hartenstein [LH25]. The secret sharing-based common coin stems from the master's thesis by Marius Haller [Hal25].

4.1 Background and Related Work

In this section, we discuss related work in the field of SMR that addresses the leader bottleneck and work that uses the hybrid fault model to increase efficiency. In addition, we provide a background and discuss related work on checkpoint protocols and garbage collection, as well as recovery and reconfiguration.

4.1.1 Addressing the Leader Bottleneck

Typically, approaches aiming at increasing the efficiency in BFT SMR focus in some way on reducing the impact of the leader bottleneck. To this end, approaches either (1) optimistically reduce the communication, (2) enable pipelining techniques, (3) follow a multi-leader approach allowing load balancing between leaders, or (4) circumvent a leader at all. In this subsection, we discuss related work from each direction.

Speculative and Optimistic Protocols Speculative and optimistic protocols aim at reducing the communication overhead on side of the leader to reduce the impact of the leader bottleneck. In speculative protocols, the replicas execute and answer requests based on the leader's proposal without waiting for the agreement process to finish. It is the task of the client to detect inconsistencies in the responses. If the client observes a mismatch, it can re-issue the request to the replicas triggering the agreement process. In such a case, replicas have to invalidate/roll back their state to the point before the speculative execution. Prominent examples are Zyzzyva [Kot+07] and its hybrid fault-tolerant variant MinZyzzyva [Ver+11]. The possibility to roll back is in contradiction with the atomic broadcast's agreement property (AB-Agreement, Definition 2.3), i.e., such approaches do not implement an atomic broadcast between the replicas. As the client can detect inconsistencies and have them corrected, safety from the client's point of view is not endangered. Instead of speculative execution, Aublin and Vogel [AV25] propose speculative ordering: The leader executes validation and ordering of requests in parallel. In optimistic protocols, e.g., ReBFT [DCK16], not all replicas participate in the agreement process to minimize the communication overhead. Passive replicas are only activated in the case of a view change. While reducing the communication overhead on side of the leader, neither speculative nor optimistic protocols are leaderless and fault handling is at least as expensive as in classic approaches.

Pipelining and Streamlined Protocols Another approach to increase performance is to use pipelining. PBFT [CL02] already contained a pipelining mechanism: The leader does not wait for the agreement process to finish before proposing the next request. The number of requests that can be pipelined is limited by a sliding window. This approach was later adopted by many protocols (e.g., MinBFT [Ver+11], JPaxos [SS12], μBFT [Agu+23], or Bandle [Wan+24]) and can be combined with other techniques (e.g., alternative fault models or speculation). Santos and Schiper [SS13] describe that pipelining can amplify

overloading effects on side of the leader, leading to inverse performance effects. Antoniadis et al. [Ant+18] show that the size of the pipeline can have a negative effect on the latency observed by clients. Tendermint [Buc16] and the HotStuff family [Yin+19; MN23; Kan+25a; Gup+25] promoted to simplify the leader regime in partially synchronous BFT SMR in comparison to PBFT-inspired approaches. Their goal is to reduce the complexity of the view change. In contrast to the **static leader** in PBFT which is only re-elected when faults are assumed, the leader is exchanged after every decision (**rotating leader**). Moreover, Chained HotStuff streamlines the protocol by reducing the number of message to a single generic message and by arranging decisions in a four-step pipeline [Yin+19; CS20a; Dec+22a; Gup+25]. As a result, there is one decision per network round trip and a linear number of messages per round. Due to the missing implicit synchronization of the replicas that PBFT achieves by all-to-all communication, rotating leader protocols need an orthogonal consensus primitive to ensure that sufficient enough replicas are at the same time in the same view. This protocol is called pacemaker [Yin+19; Lew22; MN22]. The rotating leader approach, however, still requires a leader and faulty replicas have a significant impact on the performance of the protocol [Ant+23; Ami+24].

Load Balancing and Multi-Leader Protocols Biely et al. [Bie+12] find that the necessary client communication has a significant impact on the performance of the SMR protocol. They investigate for the omission fault-tolerant Paxos protocol how the client requests can be evenly distributed among the replicas to reduce the load on the leader. In BFT SMR, it is not sufficient to receive a response from a single replica; the client has to collect responses from a quorum of replicas to ensure safety. Depending on the implementation, these are either $t + 1$ or $2t + 1$ consistent responses [BRB21; Bes+23]. If the client would not collect responses from a quorum, it could be tricked by a faulty leader (e.g., the leader did not order the request at all or it manipulated the response of the application layer). To allow equal distribution of clients among the replicas, SBFT [Gol+19] and CART [HHM24] use threshold signatures to reduce the number of responses the client has to collect. A valid aggregated signature signals that sufficient replicas agree on the response. Troxy [Li+18] uses a TEE instead of threshold signatures to achieve the same. In multi-leader protocols, multiple agreement processes are executed in parallel allowing clients to select one of the possible leaders to issue their requests. Kauri [NMR21] arranges the replicas of Chained HotStuff in a tree topology to reduce the communication overhead of the leader to a constant and to execute a constant number of agreement processes in parallel thereby increasing the pipelining effect. ISS [SPV22] is a BFT generic wrapper for leader-based protocols that uses request partitioning, a form of sharding, to instantiate multiple independent consensus instances in parallel. Moreover, they explicitly address the issue of request duplication in multi-leader protocols (inspired by Mir-BFT [Sta+22]). Hybster [BDK17] is a hybrid fault-tolerant protocol that allows to execute multiple independent consensus instances in parallel. For NxBFT, limit replicas to omission faults when interacting with clients. This adaption allows the client to collect responses from a single replica. To handle omission faults, a NxBFT client contacts a new replica after a timeout and repeats this until it receives a response from a replica. In fault-free cases, this allows every replica to propose a disjunct block of requests thereby increasing throughput and reducing latency. We discuss

for which uses cases this decision is reasonable and how NxBFT could incorporate CART or Troxy to maintain the assumption of Byzantine faults.

Leaderless Protocols Leaderless, also known as egalitarian [MAK13], consensus protocols do not rely on a leader. They allow every replica to propose requests without the need to task a leader with the ordering, hence eliminating the leader bottleneck and having the potential to significantly improve throughput in comparison to the previously discussed techniques [Dan+22]. This, however, comes at the cost of all-to-all communication, i.e., every replica has to send and receive messages from/to every other replica and, thus, at least a communication complexity of $O(n^2)$ [Zha+24a]. Modern BFT leaderless protocols can be split in DAG-based/inspired (e.g., Hashgraph [BL20], the DAG-Rider family [Kei+21], or Autobahn [Gir+24]) and "non-DAG" protocols (e.g., DBFT [Cra+18], ISOS [ED21], or the HoneyBadgerBFT family [Mil+16]). Both categories can further be split in partially synchronous and asynchronous protocols. As a rule of thumb, DAG-based protocols achieve better throughput and scalability due to the inherent parallel request pipelining [Zha+24a]. In a single round, DAG-based protocols typically exchange messages that disseminate n blocks in parallel. Typically, partially synchronous DAG protocols have lower latencies but are less resilient to crash faults or degraded network quality than asynchronous protocols [Spi+22; Gir+24; Wan+24; Ton+25]. Concurrent works lower this resilience gap significantly [Aru+25; Bab+25; Ton+25]. Lewis-Pye et al. [LNS25] provide a theoretical model to compare leader-based and leaderless approaches in terms of latency and throughput finding that DAG-based protocols achieve better throughput (by a factor of n) at the cost of latency.

In summary, we consider the empirical results of the related work on DAG-based protocols to be groundbreaking and therefore rely on DAG-based protocols as the basis for researching efficient and resilient hybrid fault-tolerant SMR for the following reasons:

1. The DAG-based approach promises the best throughput and scalability. In contrast to DAG-Rider, Hashgraph [BL20] builds the graph following a non-deterministic rule making consensus derivation more complex and denying garbage collection [Spi+22]. Moreover, Hashgraph has worst-case exponential time complexity [SBK22].
2. Missing timing assumptions make the protocols more resilient.
3. Finally, related work [Spi+22; Aru+25; Bab+25] has shown that efficient asynchronous protocols are a perfect base to be later extended to the partially synchronous model for improved latencies.

The work at hand delineates itself from the existing leaderless protocols as follows:

- We prove the omission fault tolerance of the DAG-Rider protocol with a quorum size of $\lfloor \frac{n}{2} \rfloor + 1$ and otherwise unchanged assumptions.
- We instantiate DAG-Rider with a TEE-based common coin and a TEE-based reliable broadcast in the hybrid fault model with a fault tolerance of $n > 2t$.

- We adapt Narwahl's backfilling approach [Dan+22] for hybrid faults to improve the constant communication overhead (common case n^2 for TEE-Rider, $3n^2$ for Narwhal; worst case $3n^2$ for TEE-Rider, $5n^2$ for Narwhal).
- We investigate ways to better exploit parallelism in DAG-based protocols and propose limiting replicas to omission faults when communicating with clients. We discuss under which circumstances this assumption is not compromising security.

4.1.2 The Hybrid Fault Model: Limiting Potential Faults

Adapting the fault model such that a potential attacker is assumed to have less possibilities for malicious behavior typically comes with less complex protocols and, hence, better performance. There are numerous proposals for benign fault-tolerant SMR protocols (e.g., [MAK13; Cor+13; OO14; Taf+20; Wan+24])[1]. Those protocols assume that the replicas are non-malicious but may crash or omit messages (e.g., due to network partitions). Clients, however, may behave Byzantine. Typically, Byzantine clients can be handled without additional communication (e.g., by using cryptography and stateful protocols) which is not possible for Byzantine replicas. If now a replica shows a Byzantine fault, e.g., caused by a compromised host system, a bug, or a misconfiguration, safety and liveness of the protocol are at risk [Agu+23]. We argue that in use cases as categorized in Chapter 3, i.e., where the system is not operated by a single entity, the risk that the replicas are not operated with the same necessary diligence by everyone is high; thus, amplifying the risk of (unintended) Byzantine faults.

If Byzantine faults still should be tolerated, the hybrid fault model (see Section 2.5) is a valid choice to reduce the complexity of the protocol. Ben-David et al. [BCS22] extend the seminal work by Clement et al. [Cle+12] and show that any crash fault-tolerant atomic broadcast protocol can be compiled into a BFT atomic broadcast protocol with polynomial overhead when non-equivocation and transferable authentication can be assumed. In general, however, the compiler may produce non-ideal results in terms of overhead: The compiler reliably broadcasts *every* message and *every* message is processed by the trusted subsystem. As an example, if an omission fault-tolerant protocol is used as base protocol, *at least* the additional reliable broadcast added by the compiler is not required.

Independent from the generic compiler approach, there exists a variety of proposals that transform a BFT agreement protocol to the hybrid fault model by adding just as much functionality to the trusted subsystem as required. Prominent examples are MinBFT [Ver+11] and FastBFT [Liu+19] for PBFT and Damysus [Dec+22a], OneShot [Dec+24], and Achilles [Niu+25] for HotStuff. Recent work [Zha+24c] investigates leaderless, hybrid fault-tolerant consensus in partial synchrony. All proposals share a signature service that combines a digital signature with a monotonic counter and are designed for the partially synchronous model. This idea goes back to TrInc [Lev+09] and MinBFT's USIG (Unique

[1] Please note that in publications authors often speak of the crash fault model without specifying the fault model in more detail.

Sequential Identifier Generator) [Ver+11]. Such a device can be used to implement a BFT reliable broadcast – a crucial building block for consensus primitives in general and atomic broadcast in particular (see Section 2.6) – with a fault tolerance of $n > t$ (see Section 4.2). Bessani et al. [Bes+23] discuss such classic hybrid fault-tolerant approaches and point out that it is important for TEE-based SMR to ensure the agreement property of the atomic broadcast is correctly implemented; otherwise, clients may loose their liveness guarantee. Moreover, they point out that concurrent/parallel ordering of a multitude of proposals, e.g., as in leaderless protocols, is possible and required for accelerated performance.

Xu et al. [Xu+18] and Amoussou-Guenou et al. [AHP25] propose an asynchronous, hybrid fault-tolerant (binary) consensus protocol that relies on a simple signature service with a monotonic counter as described above. Both have non-constant expected time complexity and use local randomness to break possible ties, but Mostéfaoui et al. [MPW24] showed that for $n < 3t$ a common coin achieves better results. Moreover, a direct implementation of atomic broadcast instead of a composition of consensus instances may be more efficient [Mil+16; Kei+21]. To the best of our knowledge, Teegraph [Fu+22] and Fides [Xie+25] are the only proposals for asynchronous, TEE-based atomic broadcast. Since Teegraph builds upon Hashgraph, it inherits the problems of Hashgraph, i.e., the lack of a deterministic construction rule and possible exponential time complexity (see above). Fides is concurrent work to this dissertation and proposes a transformation of DAG-Rider by using a TEE-based reliable broadcast, a TEE-based common coin, and a TEE-based transaction disclosure. The TEE-based common coin is similar to one of the two we propose and uses a PRNG inside the enclave to minimize the cryptographic overhead of otherwise required threshold cryptography. To the best of our knowledge, Fides is the only work proposing an asynchronous, TEE-based common coin. The transaction disclosure mechanism keeps requests confidential until they are to be executed and aims at reducing the probability for censorship. This mechanism is related to HoneyBadgerBFT's dissemination technique [Mil+16] and uses a TEE to minimize the cryptographic overhead. In contrast to Fides, we show that the reliable broadcast can be implemented with a $O(n^2)$ communication complexity. Finally, we show that Fides' proposal to reduce the wave length of DAG-Rider breaks the common core property required for liveness, regardless of the adoption of omission, hybrid, or Byzantine faults.

Note that due to the impossibility of strong validity consensus with $n < 3t$ (see Section 2.6), not every BFT atomic broadcast can be transformed using non-equivocation and transferable authentication. A relevant example is the HoneyBadgerBFT protocol [Mil+16] which relies on a binary strong validity consensus. In contrast to most proposals, CCF [How+23] deploys the full SMR stack, based on Raft [OO14], to the enclave. The CCF approach enforces more than only non-equivocation and transferable authentication: Instead of the untrusted host system, the enclave itself produces the consensus messages. Hence, there are no malicious parties participating in the consensus protocol. This pragmatic fault model compiler would allow to compile protocols like HoneyBadgerBFT with $n < 3t$ as there are only benign faults for which there is no need for different validity notions (see Section 2.6). The downside is that the enclave has to process *all* messages limiting the performance (see Sections 2.5 and 3.3) in comparison to approaches reducing the use of

the enclave. We deploy only as much functionality to the enclave as required to implement hybrid fault-tolerant reliable broadcast and common coin.

In summary, the work at hand delineates itself from existing work in the following ways:

- For TEE-Rider, we design, similar to Fides [Xie+25], a small TEE to implement a hybrid fault-tolerant atomic broadcast protocol based on DAG-Rider. We improve the communication complexity of the atomic broadcast to $O(n^2)$ ($O(n^3)$ for Fides).
- We show that, contrary to the intuitive believe, the hybrid fault model does not allow to shorten the wave length in DAG-Rider-like protocols.
- We propose two TEE-based common coins, one based on a PRNG and one based on secret sharing, that can be used in the hybrid fault model and compare the impact of a broken TEE on both approaches.
- We propose the first mostly asynchronous, hybrid fault-tolerant SMR protocol and investigate the implications of the combination of asynchrony and TEEs on setup, garbage collection, and crash recovery.

4.1.3 Checkpoints, State Transfer, and Garbage Collection

Replicas cannot store the entire history of the ledger indefinitely; storage is limited [Dis21]. If replicas would delete without coordination, at least liveness properties and the success of recovery and backfilling mechanisms would be at non-negligible risk[2]. To this end, practical SMR systems rely on the combination of checkpoints and garbage collection to limit the required storage. The checkpoint protocol proposed for PBFT [CL02], which remains highly influential to this day [Dis21], works as follows: A checkpoint is achieved in a voting process that operates independently from the protocol ordering client requests. Whenever a replica reaches a checkpoint, i.e., it successfully committed and executed a pre-defined number of requests, it broadcasts a checkpoint message to the other replicas. The checkpoint message contains information to identify the checkpoint (e.g., a sequence and view number) and the current state of the replicated application in form of a digest. As soon as sufficient consistent checkpoint messages, i.e., $2t + 1$, are received, the checkpoint is stable. A **stable checkpoint** is a proof that at least $t+1$ correct replicas had the same state when reaching the checkpoint. Correct replicas can delete all data belonging to requests older than the stable checkpoint.

Distler [Dis21, Section 7] points out that garbage collection must not violate SMR liveness: Faulty replicas can help to stabilize a checkpoint but commit omission faults towards clients, e.g., by not sending responses. Assume a client request is part of a garbage collected decision but the client did not receive a quorum of responses yet. If now a correct replica has fallen behind, e.g., due to the partially synchronous or asynchronous channel, and other correct replicas already deleted the messages that led to the request being ordered,

[2] Please note that a synchrony assumption is also a form of coordination.

the delayed replica cannot perform all computations that led to the response anymore. To close the gap to the other correct replicas, the replica must initiate a state transfer to apply the missing requests. In a **state transfer protocol**, a replica receives the state of a stable checkpoint allowing to safely fast forward the receiving replica's local state. Due to the garbage collection, the state transfer typically does not transfer all requests in the order they have to be applied. Instead, the end result after all requests of a checkpoint interval have been applied is transferred. In particular, the replica cannot execute the request and, thus, cannot send a response to the client. To circumvent this problem, the state transfer must contain all responses sent to clients such that replicas that received a state transfer can send responses to clients if required to ensure liveness. MinBFT [Ver+11] adopts the mechanism of PBFT with an altered quorum size of $t + 1$. However, an explanation how expected counter values – which are required to validate USIG-signed messages – are updated during a state transfer is missing. Stathakopoulou et al. [SPV22] implement a wrapper around PBFT, HotStuff, and Raft that provides checkpoint-based garbage collection. Hence, the checkpoint-based garbage collection is also applicable to the rotating leader paradigm.

With asynchrony, the garbage collection problem is more difficult to solve than with partial synchrony. This is especially true for DAG-based protocols [Dan+22; Spi+22]. Spiegelman et al. [Spi+22] find that garbage collection in asynchronous, DAG-based protocols in conflict with the validity property of atomic broadcast (AB-Validity, Definition 2.3). A replica must continue the protocol whenever it receives $n - t$ messages for the current protocol stage. From those $n - t$ messages, t messages may be from faulty replicas, and, in the worst case, t messages of correct replicas may still be missing. Consequently, a replica can only be sure that it has received all messages from other correct replicas when it received n messages. In all other cases, the replica cannot distinguish whether the missing messages are from (very) slow correct replicas or from faulty ones [Spi+22, Section 7]. Danezis et al. [Dan+22] and Spiegelman et al. [Spi+22] propose to delete parts of the DAG that are already ordered. If now old parts of the DAG are deleted, the replica cannot make sense of messages from correct replicas that were heavily delayed. While the DAG structure ensures that AB-Agreement is not violated, AB-Validity is violated as the message must be dropped. The only known workaround is for replicas that observe that their proposal for a garbage-collected part of the DAG was not yet ordered to re-issue their proposal. Raikwar et al. [RGV25] claim that garbage collection is a current research gap in DAG-based protocols. They propose to investigate checkpoint-based garbage collection for DAG-based protocols as well.

TEE-Rider uses a backfilling-based reliable broadcast to achieve $O(n^2)$ communication complexity. We show that the backfilling is in conflict with the garbage collection as proposed by Danezis et al. A single correct replica may be the only correct replica that received a certain message and, thus, the only one that can backfill the message for others. If this replica deletes the message after it used it in a decision, the replica cannot backfill the message for other replicas which breaks AB-Agreement and, thus, SMR safety of NxBFT. We propose a checkpoint-based garbage collection with state transfer to ensure that SMR safety is not violated. Optimizations to minimize the communication overhead of the state transfer are possible (see, e.g., [CL02; Dis21]) but are out of scope of this dissertation.

4.1.4 Recovery and Reconfiguration

For practical systems, recovery and reconfiguration mechanisms are crucial [Bes+13; BSA14; Dis21]. Eventually, every system will experience faults, e.g., due to hardware failures. If faulty replicas are not recovered, the system may not be able to provide liveness guarantees anymore. Moreover, recovery mechanism are required to be able to perform maintenance work, e.g., for software updates or hardware replacements. Reconfiguration is tightly coupled with recovery as it allows to change the set of replicas, i.e., to add or remove replicas. A new replica must be able to synchronize with the current state of the system to be able to participate as a correct replica. Administrative commands, e.g., for recovery or reconfiguration, are typically special client operations that are ordered like any other request. State synchronization of crashed or new replicas is achieved by state transfer that is also used to ensure that garbage collection does not violate liveness (see above) [CL02; Dis21]. Replicas in need of state transfer issue a recovery request to other replicas to send their current state. When now the recovery request is ordered, the stable checkpoint following the ordered recovery request is the recovery checkpoint that will be transferred to the recovering replica. The recovering replica can then be sure that any consensus message following the recovery checkpoint (1) can be handled by the recovering replica and (2) is consistent with the state it received. We are not aware of any work that investigates recovery of DAG-based protocols.

There are several works that outline that rollbacks, i.e., the TEE is started on a previously sealed, potentially outdated state, are a significant problem for TEE-based SMR [Mat+17; Bra+19; Niu+22; Ang+23; KCG25; Niu+25; Wil+25]. To circumvent this problem, either (1) very slow performing rollback prevention mechanisms have to be deployed or (2) for parts of the TEE enforcing non-equivocation there must be no possibility to seal state making replay of old state impossible in the first place. Because of the performance impact, the first solution is typically not feasible in practice [Niu+25]. The second solution, however, makes TEEs lose all their state when being restarted, e.g., after a power outage, a crash, or simple maintenance work requiring a reboot. Recovery must now bring such parts of the TEE back into a useful state; otherwise, the replica is not able to participate in the consensus protocol anymore. As the TEE prevents equivocation, recovery can, if not done carefully, enable equivocation and, thus, become an attack vector.

Messadi et al. [Mes+25] investigate recovery of replicated confidential VMs (see Section 2.5); they do not support a fault tolerance of $n \leq 3t$ and, thus, cannot be used for hybrid fault-tolerant SMR. To the best of our knowledge, CCF [How+23] and Achilles [Niu+25] are the only TEE-based SMR protocols with a fault tolerance of $n > 2t$ that support recovery; both protocols operate in partial synchrony. The recovery in CCF is rather straightforward: Whenever a replica crashes, it is removed from the set of replicas and a new replica is added, i.e., a reconfiguration step is performed. This new replica receives a state transfer from the incumbent leader. As the complete SMR logic is implemented in the TEE, this single state transfer is sufficient to bring the new replica into a consistent state. Achilles, in contrast, does not support reconfiguration. The authors argue that reconfiguration can lead crashed nodes to contact replicas that are not part of the current replica set because

said set may have changed while the replica was down. A recovering node in Achilles sends a recovery request to the replicas and waits for a quorum of consistent recovery replies which are used to re-initialize the TEE.

We will show that the recovery of TEE-Rider is more complex than in CCF and Achilles. This is due to the asynchronous nature of TEE-Rider and the fact that the *only* interactive coordination mechanism is the TEE-based reliable broadcast. In particular, we show that Achilles-like recovery breaks TEE-Rider's properties if the quorum size is smaller than n. The impossibility result does not apply to Achilles because in Achilles, the enclaves manage a common counter that is equal at every replicas' enclave. Similar to CCF, NxBFT replicas recover through reconfiguration to circumvent the impossibility. This reconfiguration builds upon the checkpoint-based state transfer mechanism that is also used for garbage collection.

4.2 TEE-Based Reliable Broadcast

As outlined in Section 2.6, reliable broadcast (Definition 2.2) is a crucial building block for atomic broadcast and, thus, for state machine replication. Typical reliable broadcast implementations rely on echoing (single echo for hybrid faults, double echo for Byzantine faults) to ensure RB-Agreement. In the common case, i.e., when the sender is not faulty, this echoing step induces significant overhead. Danezis et al. [Dan+22] observed that, when having n senders, a causal relationship between broadcast messages can be used to leverage backfilling to implement a reliable broadcast. Whenever a message in the causal history is missing, a process can request the missing message from the peering processes. As a result, the echoing step can be avoided and the communication complexity of the reliable broadcast is reduced from $O(n^2)$ to $O(n)$ for single process and from $O(n^3)$ to $O(n^2)$ for the distributed system respectively. In this section, we use TEEs to improve on the result by Danezis et al. [Dan+22].

The section is structured as follows. First, we give a brief overview of the history of reliable broadcasts and the seminal work by Correia et al. [CVL10] and Veronese et al. [Ver+11] that introduced hybrid fault-tolerant reliable broadcast using the Unique Sequential Identifier Generator (USIG). Then, we present the USIG-based hybrid fault-tolerant causal order broadcast protocol, show its correctness, and compare it to the original DAG-Rider protocol and the Narwhal protocol. The protocol was, except the formal proofs, previously published by Leinweber and Hartenstein [LH25].

4.2.1 Background

To date, Bracha's reliable broadcast [Bra87] is the most influential, asynchronous reliable broadcast protocol. The protocol is divided in three phases: init, echo, and ready. In the init phase, the sending process broadcasts its message to all other processes. The echo phase ensures that all correct processes receive *a* message. The ready phase ensures that

all correct processes deliver the *same* message. A process will continue to the next phase if there was a quorum in the previous phase. Bracha's reliable broadcast requires $n > 3t$ processes to tolerate t Byzantine faults. Digital signatures do not suffice to increase the fault tolerance (see Section 2.6). Due to the echo phase, Bracha's reliable broadcast has a communication complexity of $O(|m|n^2)$ bit for a message m of size $|m|$. Cachin and Tessaro [CT05] showed that erasure codes can be used to decrease the communication complexity to $O(|m|n + cn^2 \log n)$ bits where c is a small constant referring to the size of the erasure code. Narwhal [Dan+22], a follow-up to DAG-Rider, uses backfilling to achieve linear communication complexity in the common case: A sender broadcasts a message m to all other processes. Every process that receives the message stores it and sends an acknowledgement in form of a signature to the sender. When the sender received $n - t$ acknowledgements, the sender can accumulate the signatures to a certificate which is then broadcast to all other processes. The certificate is a proof that the previous message was received by at least $t + 1$ correct replicas, and replicas that did not receive the message can receive it from said $t + 1$ replicas. In the common case, the communication complexity is $|m|n + |\sigma|n + |\Sigma|n$ bits where $|\sigma|$ is the size of a signature and $|\Sigma|$ is the size of the certificate. In the case of a faulty sender, one more round of communication is added, yielding a communication complexity of roughly $5n$ protocol messages in the worst case. We observe that, when using a TEE, the construction of certificates can be omitted and the communication complexity is reduced to $|m|n$ bits in the common case and $3n$ protocol messages in the worst case. The reason is that the USIG signature suffices to prove that any correct replica will accept the message if it is received. In Narwhal, however, the certificate is required to prove that at least $t + 1$ correct replicas received the exact same message. In summary, the Narwhal certificate mechanism certifies a successful voting step which is not required when using a USIG.

Correia et al. [CVL10] were the first to show that reliable broadcast can be implemented in the hybrid fault model with a fault tolerance of $n > t$. Veronese et al. [Ver+11] describe the Unique Sequential Identifier Generator (USIG) that can be used to practically achieve the result by Correia et al. The idea is rather simple: the sender is equipped with the USIG that assigns every message with a unique, increment-only counter (or identifier) and a digital signature. The sender broadcasts the message together with the counter and the signature. The combination of the counter and the signature is a proof that the message is the only message with this counter, i.e., non-equivocation is guaranteed. Consequently, a receiver can directly deliver a message after it received the message together with a valid signature for a counter value for the first time. The result is a *single echo reliable broadcast* with communication complexity $O(n^2)$ but one phase, i.e., half a network round trip, less than Bracha's reliable broadcast.

Algorithm 1 shows the USIG module of the sender. It is initialized with an asymmetric key pair (sk, pk) and a counter c. The sender can sign a message m by calling the $\textsc{sign}(m)$ function. The function computes a signature σ for the pair (c, m) using the secret key sk and the current counter c. The counter is then incremented. To prevent equivocation through rollbacks, the module must not support sealing and re-import of its state.

Algorithm 1 Unique Sequential Identifier Generator (USIG) [Ver+11]

```
1: state (sk, pk): asymmetric key pair
2: state c: ℕ, initialized with 1
3: function SIGN(m)
4:     σ ← sgn(sk, (c, m))
5:     c ← c + 1
6:     return (c − 1, σ)
```

Algorithm 2 USIG-Based FIFO Reliable Broadcast for Process p_i [CVL10; Ver+11]

```
1: state p_s: ℕ identifier of sender p_s
2: state pk_s: public key of sender's USIG
3: state P: set of all processes
4: state expectedCounter: ℕ, initialized with 1
5: state received: addressable priority queue of messages, initialized empty
6: state usig: USIG instance                                   // Only for sender p_s
7: // The function r_broadcast is only available for the sender p_s
8: function R_BROADCAST(m)
9:     (c, σ) ← usig.sign(m)
10:    send ⟨BCAST, c, m, σ⟩ to p_s
11: upon BCAST(p_k, c, m, σ)
12:    if ¬vfy(pk_s, (c, m), σ) then
13:        abort
14:    if c < expectedCounter ∨ received.containsKey(c) then
15:        abort
16:    for all p_j ∈ P \ {p_s, p_k, p_i} do
17:        send ⟨BCAST, c, m, σ⟩ to p_j
18:    received.insert(c, m)
19:    while ¬received.isEmpty() do
20:        (c′, m′) ← received.peek()
21:        if c′ ≠ expectedCounter then
22:            abort
23:        expectedCounter ← expectedCounter + 1
24:        (c, m) ← received.pop()
25:        r_deliver (c, m)
```

Algorithm 2 shows the USIG-based single echo reliable broadcast protocol. Only the sender is equipped with a USIG. Formally, it implements a first-in first-out (FIFO) reliable broadcast [CGR11, Module 3.8] to prevent indefinite memory growth. A naive, non-FIFO solution would rely on an ever-growing set of already delivered counters. The sender p_s has a function R_BROADCAST(m) that invokes the sender's USIG to create a signature over m and the USIG's current counter value. The function then sends the BCAST message to p_s itself triggering the BCAST message handler. The BCAST handler is triggered whenever a process p_i receives a message. The handler first verifies that the message is validly signed

and that a message for the counter value c was not already received. The message would have been already received if the current expected counter value is higher than c or if there is a message in the priority queue of received but not yet delivered messages (*received*) that has the counter value c as its sorting key. If the message is valid, it is broadcast to all but the sender p_s, the relayer of the message p_k, and the current process p_i. Note that it may $p_s = p_k = p_i$. In a second step, the messages are delivered in the order of their counter values using the *received* priority queue; gaps in the counter values are not allowed.

The correctness of the protocol crucially relies on the USIG and is easy to show: Due to the echoing of messages (ll. 16–17), the protocol ensures that, if a correct process receives a message, the same message is received by every correct process. As the USIG enforces non-equivocation for assigned counters, RB-Agreement follows. If the sender is correct, it will send the message to itself (l. 10) and, subsequently, forward the message to all processes. Moreover, a correct sender will send all messages it created a USIG signature for. If the sender is faulty and no correct process receives the message m or a message with a lower counter from the same sender, no correct process will deliver m. By the argument for RB-Agreement, RB-Validity follows as well. RB-Integrity is ensured by the USIG and the conditional aborts in ll. 12–15.

4.2.2 USIG-Based Causal Order Reliable Broadcast Protocol

We build upon the USIG-based reliable broadcast to propose a USIG-based causal order reliable broadcast [CGR11, Module 3.9]. The causal order of messages allows to achieve linear communication complexity in the common case. The idea of the protocol is relatively simple. When a process broadcasts a message, it includes the history of already received messages. Correct processes do not echo messages by default. Instead, if an ancestor is unknown, i.e., it was neither already delivered nor received, a request for the missing message is sent to all processes. Due to asynchrony, correct processes cache message requests if they cannot answer them directly: It is possible that the corresponding message arrives delayed and the process receiving the message request is the only correct process that receives the requested message due to faults by the sender.

The protocol is listed in Algorithm 3. When a process aims to broadcast a message, it invokes the USIG to receive a counter and a signature for the message (l. 9). The process then creates a history array that contains the expected counters of all processes (ll. 12–14). If no message was received from a process yet, i.e., the expected counter is 1, the history entry is set to $\perp$. The process then sends the message to all processes (l. 15). When a process receives a broadcast message, it first verifies the signature (l. 17) and that a message for the counter was not yet received (l. 18). If the message is valid, it is added to the received messages and, to prevent indefinite growth of set of requested messages, removed from the set of requested messages (l. 19). The process then checks if it has received requests for the message which are answered by sending the message to the requesting processes (ll. 20–21). Then, the process iterates over all processes and checks if it can deliver messages from the received messages (ll. 22–36). For a message to be delivered, it must be the next expected counter for the process (l. 25), and the ancestry of the message

Algorithm 3 USIG-Based Causal Order Reliable Broadcast for Process p_i

```
1: state P: set of all processes
2: state PK: array of public keys
3: state expectedCounters: array of ℕ, initialized with 1
4: state received: array of addressable priority queues, initialized empty
5: state delivered: array of array of messages, initialized empty
6: state requested: set of tuples (ℕ, ℕ), initialized empty
7: state receivedRequests: map of key (ℕ, ℕ) to list of processes, initialized empty
8: state usig: USIG instance
9: function R_BROADCAST(m)
10:     (c, σ) ← usig.sign(m)
11:     history ← array of size n initialized with ⊥
12:     for all p_j ∈ P do
13:         if expectedCounters[p_j] > 1 then
14:             history[p_j] ← expectedCounters[p_j] − 1
15:     for all p_j ∈ P do send ⟨BCAST, p_i, c, m, σ, history⟩ to p_j
16: upon BCAST(p_k, p_s, c, m, σ, history)
17:     if ¬vfy(PK[p_s], (c, m), σ) then abort
18:     if c < expectedCounters[p_s] ∨ received[p_s].containsKey(c) then abort
19:     received[p_s].insert(c, (m, σ, history)); requested.remove((p_s, c))
20:     for all p_j ∈ receivedRequests.remove((p_s, c)) do
21:         send ⟨BCAST, p_s, c, m, σ, history⟩ to p_j
22:     for all p_j ∈ P do
23:         while ¬received[p_j].isEmpty() do
24:             (c′, (m′, σ′, history)) ← received[p_j].peek()
25:             if c′ ≠ expectedCounters[p_j] then continue
26:             ancestryComplete ← True
27:             for all (p_h, h) ∈ history do
28:                 if h ≥ expectedCounters[p_h] then
29:                     ancestryComplete ← False
30:                     if (p_h, h) ∉ requested then
31:                         for all p_j ∈ P do send ⟨REQ, p_h, h⟩ to p_j
32:                         requested.add((p_h, h))
33:             if ¬ancestryComplete then continue
34:             expectedCounters[p_j] ← expectedCounters[p_j] + 1
35:             (c, m, σ, history) ← received[p_j].pop()
36:             r_deliver (c, m); delivered[p_j][c] ← (m, σ, history)
37: upon REQ(p_k, p_s, c)
38:     m ← ⊥; σ ← ⊥
39:     if delivered[p_s][c] ≠ ⊥ then (m, σ, history) ← delivered[p_s][c]
40:     if m = ⊥ ∧ received[p_s].containsKey(c) then
41:         (m, σ, history) ← received[p_s].get(c)
42:     if m ≠ ⊥ then send ⟨BCAST, p_s, c, m, σ, history⟩ to p_k
43:     else receivedRequests[(p_s, c)].add(p_k)
```

must be complete, i.e., all expected ancestors must be known (ll. 26–33). If the ancestry is not complete, the process checks if it already requested the missing ancestor (l. 30), and, if not, sends a request to all processes (l. 31). If the ancestry is complete, the process increments its expected counter (l. 34), pops the message from the received messages (l. 37), and delivers the message (l. 36). When a process receives a request for a message, it checks if it has the message in its delivered (l. 39) or received messages (ll. 40–41). If it has the message, it sends the message to the requesting process (l. 42); otherwise it adds the requesting process to a list of processes that requested the message (l. 43).

4.2.3 Correctness

The algorithm still resembles a USIG-based FIFO broadcast. Thus, it leans on arguments for the FIFO reliable broadcast presented above.

Theorem 4.1 (RB-Integrity property of Algorithm 3)**.** *Under the assumption of perfect point-to-point links and a TEE-based USIG, the causal order broadcast protocol in Algorithm 3 fulfills RB-Integrity with a fault tolerance of $n > t$: For each tag $c \in \mathbb{N}$, a correct process delivers at most one message.*

Proof. By the check in l. 17, processes are forced to combine their message with a valid USIG signature. The USIG increments the counter after each signature (Algorithm 1, l. 5), thus, under the assumption of a correct TEE, enforces the uniqueness of the combination of counter and message. When a correct process receives a message for the first time, it will be added to the received messages (l. 19). Moreover, a correct process will reject any message for a counter value that it already received (l. 18). As soon as the message can be delivered, the process increases the expected counter value (l. 34) and removes the message from the received messages (l. 35). Thus, if a correct process delivers a message for a counter value c, it will only do so once. □

Theorem 4.2 (RB-Agreement property of Algorithm 3)**.** *Under the assumption of perfect point-to-point links and a TEE-based USIG, the causal order broadcast protocol in Algorithm 3 fulfills RB-Agreement with a fault tolerance of $n > t$: If a correct process delivers (c, m), then all correct processes deliver the same message m for tag c.*

Proof. If a correct process p_i delivers a message m, it was able to deliver its complete ancestry. This follows from the recursive application of the ancestry condition in ll. 27–32. If another correct process p_j receives this message m and it does not know one of its ancestors, this is due to two reasons: (1) the ancestor is from a correct process and the corresponding message was not yet delivered by the asynchronous channel or (2) the ancestor is from a faulty process that committed a send omission fault. In case (1), by the assumption of perfect point-to-point links, p_j will receive the ancestor eventually. In case (2), p_i will be able to answer any ancestor request: Since p_i delivered m, it added it to its delivered messages (l. 36). In both cases, by Theorem 4.1 and the USIG-enforced binding

of message and counter (l. 17), all correct processes will deliver the same message as there is only one message for each tag c. □

Lemma 4.1. *When a correct process broadcasts a message m, it will immediately deliver m, i.e., it does not require additional communication to be able to deliver.*

Proof. Correct processes will only use counter values as history information that they already delivered (ll. 14 and 34). From the recursive application of this behavior follows that each of those counter values is the highest of a set of already delivered counter values. Consequently, when now the BCAST handler is invoked at the sender, the checks in ll. 17–18, 25, and 26–33 will pass and the sender will be able to deliver the message directly. □

Theorem 4.3 (RB-Validity property of Algorithm 3)**.** *Under the assumption of perfect point-to-point links and a TEE-based USIG, the causal order broadcast protocol in Algorithm 3 fulfills RB-Validity with a fault tolerance of $n > t$: If a correct sender broadcasts message m, then every correct process eventually delivers m.*

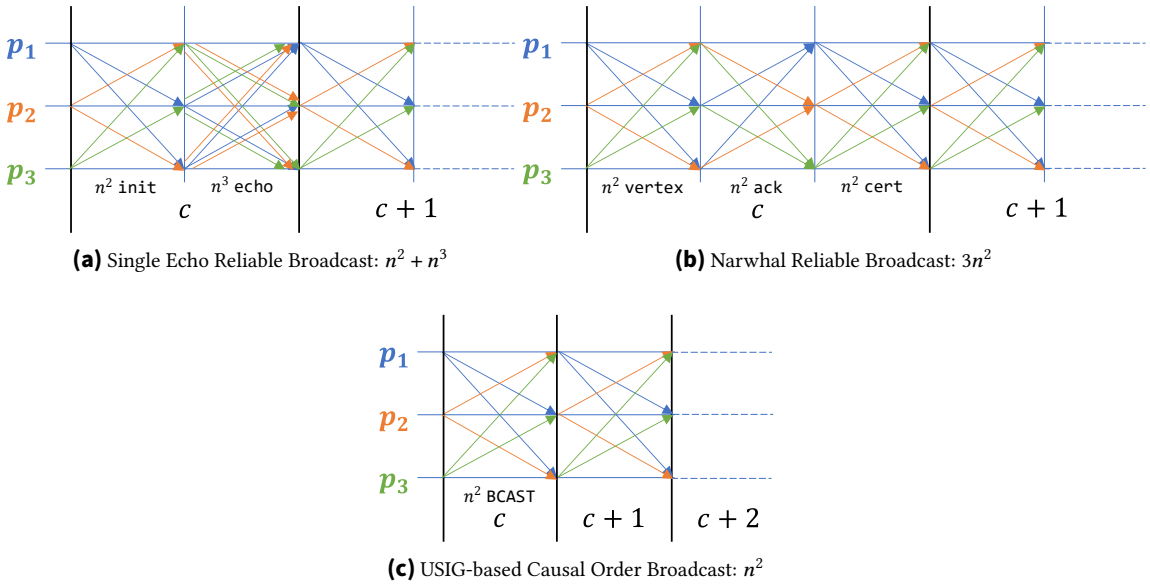
Proof. By Lemma 4.1, the correct sender p_s will deliver m immediately. By Theorem 4.2, all correct processes will eventually do alike. □

4.2.4 Analysis and Comparison

We compare the communication complexity of the USIG-based causal order broadcast protocol in Algorithm 3 to the single echo reliable broadcast protocol by Correia et al. [CVL10] and the Narwhal reliable broadcast [Dan+22]. Figure 4.1 shows the common case communication patterns for the three protocols.

In the single echo reliable broadcast, each process echos each message it receives for the first time. Hence, the single echo reliable broadcast requires in the common and in the worst case $n + n^2$ messages. For n senders, we get a communication complexity of $n^2 + n^3$ messages.

The pattern in Narwhal [Dan+22] is as follows: The sender sends the message to all processes (i.e., n messages). Each process that receives the message sends an acknowledgement to the sender (i.e., another n messages). When the sender collected $n - t \geq 2t + 1$ acknowledgements, it sends a certificate to all processes (i.e., another n messages). When a process received a valid certificate for a message, it can deliver the message. Thus, for n senders, the communication complexity is $3n^2$ messages in the common case. A process must include $n - t$ certificates to be allowed to broadcast a new message. In the case of faults or a slow sender, the certificates in received messages can be used to backfill their ancestry. In this case, a process requests the missing message at the $n - t$ processes that are listed in the certificate giving $n - t$ responses. Thus, in the worst case, the communication



(a) Single Echo Reliable Broadcast: $n^2 + n^3$

(b) Narwhal Reliable Broadcast: $3n^2$

(c) USIG-based Causal Order Broadcast: n^2

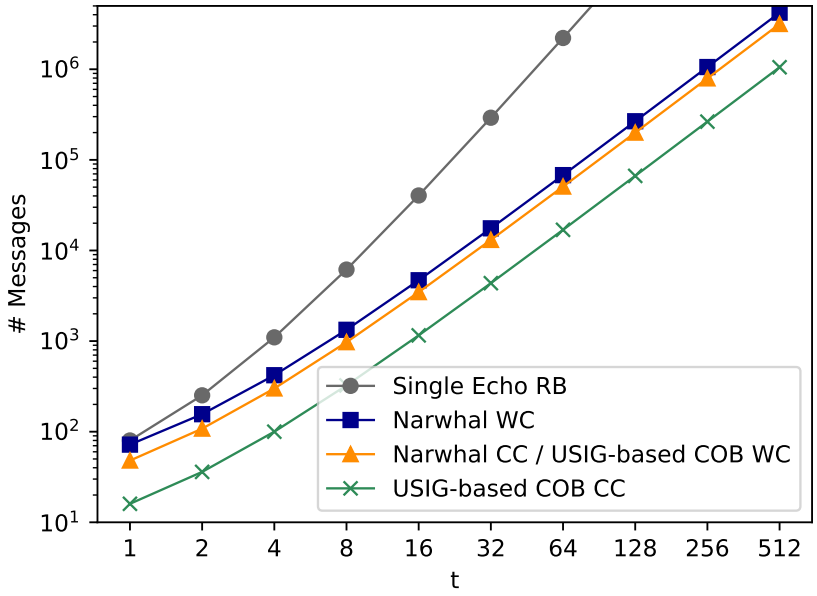
Figure 4.1: Common case communication patterns for single echo reliable broadcast (e.g., [CVL10]), Narwhal reliable broadcast [Dan+22] and USIG-based causal order broadcast (this work) with n senders. Blue vertical lines delimit communication rounds, black vertical lines delimit broadcast tags. The colors of the arrows signify the payloads' originator. The USIG-based causal order broadcast proceeds with one round of communication per broadcast round.

complexity is $3n + 2(n - t) = 5n - 2t$ messages for a single sender, i.e., $5n^2 - 2tn \simeq 5n^2$ messages for n senders.

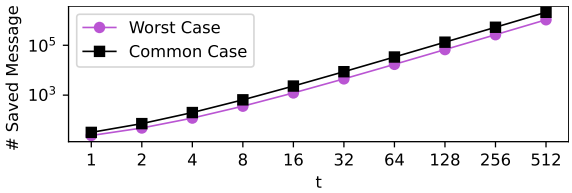
The USIG-based causal order broadcast as presented in Algorithm 3 proceeds with one broadcast message in the common case: The sender sends the message to all processes (i.e., n messages). Thus, for n senders, the communication complexity is n^2 messages in the common case. In the case of faults or a slow sender, a correct process will learn about the existence of a message through the history information in other received messages. In this case, the process must request the message from all processes yielding n requests and n responses. Thus, in the worst case, the communication complexity is $3n^2$ messages for n senders.

In Figure 4.2, we instantiate the communication complexity for $n = 2t + 1$ senders and $t = 2^i, i \in [0, 9] \subset \mathbb{N}$. It is clearly visible that the single echo reliable broadcast is dominated by the cubic complexity of the echo phase (Figure 4.2a). In the common case, the USIG-based causal order broadcast (denoted as COB in the figure) saves $\frac{2}{3}$ of the messages (Figure 4.2c) which are roughly 30k messages (Figure 4.2b) for $t = 64$. In the worst case, the savings are reduced to at most $\sim 30\%$ for $t = 1$. Here, Narwhal shows a benefit of the certificates: a process is not required to contact all processes to retrieve the missing message. The number of saved messages of the USIG-based causal order broadcast declines with increasing t.

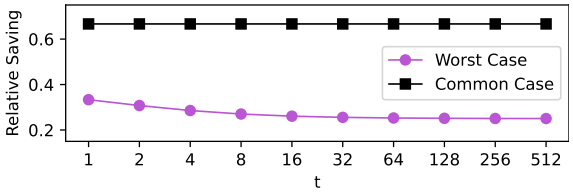
In summary, both causal order broadcast protocols, Narwhal and the USIG-based causal order broadcast, are more efficient than the single echo reliable broadcast. They trade a lower communication complexity in the common and in the worst case for a higher



(a) Total number of messages



(b) Absolute savings of USIG-based COB



(c) Relative savings of USIG-based COB

Figure 4.2: Number of exchanged message for single echo reliable broadcast (RB, e.g., [CVL10]), Narwhal reliable broadcast [Dan+22] and USIG-based causal order broadcast (COB, this work) with $n = 2t + 1$ senders and $t = 2^i, i \in [0, 9] \subset \mathbb{N}$. In Figure 4.2a, CC denotes the common case and WC denotes the worst case. The small plots show the absolute (Figure 4.2b) and the relative (Figure 4.2c) number of saved messages when using the USIG-based causal order broadcast instead of Narwhal.

algorithmic complexity and memory consumption. The single echo reliable broadcast protocol is not required to store delivered messages for backfilling requests, leading to a memory consumption of $O(n)$ in the common case. In contrast, the Narwhal and the USIG-based causal order broadcast protocol require to store all delivered messages for backfilling requests (i.e., $\Omega(n + m)$ where m is the number of messages broadcast). Thus, consensus-based garbage collection is required.

4.3 TEE-Based Common Coins

The FLP impossibility [FLP83] shows that for a deterministic consensus protocol in an asynchronous system it is impossible to tolerate a single crash fault. Ben-Or [Ben83] and Rabin [Rab83] were the first to use randomization to circumvent the impossibility. Ben-Or relies on local randomness. Whenever a process has to decide but it observes a tie, the

process invokes a local random function to break the tie. In contrast, Rabin uses a common coin that is shared among all processes. In a common coin, all processes receive the same random value for the ith invocation of the coin. Asynchronous, DAG-based approaches rely on a common coin; in current designs, the common coin cannot be replaced by a local coin. Thus, in this dissertation, we solely focus on common coins.

We define a common coin (or global perfect coin) based on [CKS05; Kei+21] as follows:

Definition 4.1 (Common Coin)**.** Each process $p \in P$, $P := \{p_1, \ldots, p_n\}$, can start the coin protocol for arbitrary instances $c \in \mathbb{N}$ by calling createCoinShare(c) that returns a coin share s_p. Let $t \in \mathbb{N}, 0 \leq t < n$, be the reconstruction threshold and ϵ a security parameter. The function computeCoin(c, S) takes a coin instance c and a set of coin shares S and returns a value $v \in V$ where $V := [1, n] \subset \mathbb{N}$ or $\perp$. The functions createCoinShare(c) and computeCoin(c, S) can be combined to a blocking function toss(c) that returns a value $v \in V$ or $\perp$ and abstracts the coin internals. A valid implementation satisfies the following properties:

CC-Agreement: If two correct processes call computeCoin($c, \cdot$) for the same instance c with return values v_1 and v_2, then $v_1 = v_2$.

CC-Termination: If at least $t + 1$ correct processes call createCoinshare(c) for the same instance c, then every call to computeCoin($c, \cdot$) eventually returns a value $v \neq \perp$.

CC-Unpredictability: As long as less than $t + 1$ process call createCoinshare(c) for the same instance c, the return value of computeCoin($c, \cdot$) is indistinguishable from a uniformly random value except with negligible probability ϵ.

CC-Fairness: The probability for all instances c to output any value $v \in V$ is $\frac{1}{n}$.

Typically, common coins are implemented using some form of threshold cryptography. State of the art for implementing a common coin is the "Cachin coin" by Cachin et al. [CKS05, Section 6] which is based on the Diffie-Hellman problem and a lightweight zero-knowledge proof. The coin originates in the work by Naor et al. [NPR99]. The second famous variant stems as well from Cachin et al. and is based on threshold signatures [CKS05, Section 4.3]. Both coins require a distributed key generation (DKG) during setup to generate a threshold-shared key pair with threshold t. A recent empirical evaluation [Bar+25] shows that the Cachin coin is the most efficient known common coin that does not rely on a TEE. Still, the use of threshold cryptography is costly. In most cases, the cost stems from the verification steps that ensure that only valid coin shares are combined to retrieve the coin value. In contrast, a TEE-based common coin implementation can leverage the properties of trusted execution environments to reduce the cost of these verification steps. We propose and compare two TEE-based common coin implementations: a "naive" approach deploying a cryptographically secure pseudorandom number generator (CPRNG) to the TEE and the TEE-based Cachin coin.

4.3.1 Naive TEE-Based Coin

This coin implementation differs slightly from the above definition. The coin instances must be tossed in a total order starting with coin instance $c = 1$. It is not possible to skip coin instances. This means that it is not possible to start the protocol at any time for arbitrary coin instance. The implementation assumes a cryptographically secure pseudorandom number generator (CPRNG), a digital signature scheme (for both see Section 2.2), and a TEE (Definition 2.1). This coin implementation was previously published by Leinweber and Hartenstein [LH25]; Xie et al. [Xie+25] proposed an equivalent approach in concurrent work.

Protocol The TEE maintains a counter c that is incremented after each successful invocation of the common coin. Additionally, the TEE maintains a list of public keys of all processes and a CPRNG state. Moreover, each process maintains a secret key. During setup, the processes must reach agreement on the TEE identities of all processes, all TEEs must be initialized with the same set of public keys, and all TEEs must establish a common seed which is used to initialize the CPRNG. The counter c is initialized with 1 and is incremented after each successful invocation of the common coin. When a process p_i invokes CREATECOINSHARE(c), the process creates a signature σ over c as the coin share $s_{p_i} := (c, \sigma)$ using its secret key. As soon as p_i collected $t + 1$ valid coin shares for instance c, it can invoke COMPUTECOIN($c, \cdot$), a function hosted by the TEE, to retrieve the coin value v. The TEE verifies that it received $t + 1$ validly signed coin shares created by $t + 1$ processes and that they all belong to the same coin instance c expected to be next by the TEE. If these conditions are met, the TEE invokes the CPRNG to retrieve a random value v which is returned. Moreover, c is incremented by 1. If the conditions are not met, the TEE aborts and returns $\perp$.

Correctness Note that CREATECOINSHARE($\cdot$) does not require to be executed inside the TEE. The enclave will not reveal a toss unless at least sufficient processes requests a toss. However, as soon as at least $t + 1$ processes request a toss for an instance and distribute the generated coin share, the TEE will return a value v that is, due to the CPRNG properties and the TEE assumption, indistinguishable from a uniformly random value and fair. Under the assumption that all correct processes initialize the CPRNG with the same seed, the TEE will return the same value v for all correct processes and coin instances.

4.3.2 TEE-Based Cachin Coin

This coin implementation allows to generate coin shares for arbitrary instances c. This implementation is based on [CKS05, Section 6] and [NPR99]. It is based on the Diffie-Hellman problem, i.e., the discrete logarithm is hard to compute, and additionally assumes a TEE (see Definition 2.1), cryptographic hash functions, and a digital signature scheme

(see for both Section 2.2). This coin implementation was developed in the master thesis by Marius Haller [Hal25].

Protocol The TEE's state is limited to a secret key and a so-called result key share. The result key share is a threshold-shared secret and used to generate the coin shares. The secret key is used to sign coin shares. The coin share is created and signed within the enclave; a valid signature signals the authenticity and the correctness of the coin share. During setup, the processes must reach agreement on the TEE identities and the TEE's public keys of all processes. Moreover, in a distributed key generation, the TEEs must generate a result key pair (rk, PK) where $rk \xrightarrow{(t,n)} (rk_1, \ldots, rk_n)$ is threshold-shared between all n parties with threshold t and PK is the global verification key using Shamir's secret sharing scheme [Sha79]. When a process p_i invokes CREATECOINSHARE(c), the TEE maps $g := H_1(c)$ to the prime field of the result key pair using a hash function H_1 and computes the coin share $s_{p_i} := (c, \tilde{g}_{p_i}, \sigma)$ where $\tilde{g}_{p_i} := g^{rk_i}$ and σ is a signature over $(c, \tilde{g}_{p_i})$ using the TEE's secret key. As soon as p_i collected $t + 1$ validly signed coin shares for instance c, it can invoke COMPUTECOIN($c, \cdot$). The COMPUTECOIN(c, S) function can be executed outside the TEE and computes

$$v = H_2(\prod_{p_i \in T} \tilde{g}_{p_i}^{l_i}).$$

This is the polynomial interpolation of the coin shares where T is the set of process identifiers contributing to the collected shares S, H_2 is a hash function mapping from the prime field to the coin's domain and l_{p_i} the adequate Lagrange coefficient. Similar to modern signature schemes (e.g, ECDSA and EdDSA, see Section 2.2), instead of a finite field with prime order, implementations should use elliptic curves to decrease the size of the coin shares and improve computation speed [Bar+25]. In this case, all exponentiations translate to scalar multiplications on the elliptic curve.

Correctness This coin implementation resembles the Cachin coin with the difference that the zero-knowledge proof is replaced by the TEE constructing the coin share and signing it. Instead of verifying the zero-knowledge proof, receiving processes verify the signature. For CC-Agreement, it is crucial that only valid coin shares are combined to retrieve the coin value. Hence, if the TEE assumption holds, CC-Agreement, CC-Unpredictability, and CC-Fairness follow from Cachin et al.'s original proof [CKS05, Section 6.2]. If at least $t + 1$ correct processes invoke CREATECOINSHARE(c) and distribute the coin share for the same instance c, every correct process will eventually receive at least $t + 1$ valid coin shares and be able to calculate the interpolation.

4.3.3 Comparison

Table 4.1 summarizes the cost of the common coin operations for the Cachin coin, the naive TEE-based coin, and the TEE-based Cachin coin. The costly operations are the context switches (CS) to the enclave [CD16; Li+18; WAK18] and the scalar multiplication (MUL)

Protocol	createCoinShare()	Validate coin share	computeCoin()
Cachin coin [CKS05]	3MUL	$4n$MUL	$(t+1)$MUL
Naive TEE coin	1MUL (+1CS USIG)	$2n$MUL	1CS + $(2t+2)$MUL
Cachin TEE coin	1CS + 2MUL	$2n$MUL	$(t+1)$MUL

Table 4.1: Analytical evaluation of common coin operation cost. The table lists for the Cachin coin, the naive TEE-based coin, and the TEE-based Cachin coin the cost for the operations createCoinShare(·), coin share validation, and computeCoin() the cost in terms of the number of context switches to the TEE (CS) and the number of scalar multiplications with elliptic curve points (MUL). The dominant operations are the coin share validation and computeCoin() for which the TEE-based Cachin coin is the most efficient. If the naive coin is combined with a USIG, one additional context switch during coin share creation is required. The Cachin coin requires a zero-knowledge proof for coin share generation and verification, which is not required for the TEE-based Cachin coin.

with elliptic curve points [BL07] required for signature generation (one multiplication), signature verification (two multiplications), and, for the Cachin variant, interpolation (one multiplication per coin share). In addition to this operations, the Cachin coin requires two multiplications for the generation of the zero-knowledge proof and four multiplications for the verification of the zero-knowledge proof. While the TEE-based Cachin coin outperforms the naive TEE-based coin, the naive TEE-based coin solely relies on standard cryptography that is typically available in a TEE. We provide an empirical analysis in Section 5.4.3. Both implementations can be combined with a USIG signature service as described in Section 4.2 or any round-based protocol that produces signatures per process and round (e.g., as DAG-Rider does, see Section 4.4.1). For the naive coin, the signature has to be generated inside the TEE, causing an additional context switch. The USIG signatures serve as coin shares and no additional coin share generation is required. For the TEE-based Cachin coin, the combination with a USIG causes no additional overhead: The coin share is created within the TEE anyways and the USIG signature is then on the coin share, the counter, and the message, thus, serving as validation for the coin share.

If the TEE assumption does not hold, the naive TEE coin sacrifices CC-Unpredictability. An attacker could simply generate coin values as it gains access to the CPRNG state and seed. CC-Agreement, however, is still guaranteed as the attacker cannot manipulate the CPRNG state at processes it does not control. In contrast, the TEE-based Cachin coin sacrifices CC-Agreement while CC-Unpredictability is still guaranteed: The attacker can generate invalid coin shares but sign them correctly. Correct processes will now combine the invalid coin share with valid coin shares and receive a wrong coin result. If the attacker disseminates different invalid coin shares to different processes, the processes will receive different coin values. If the coin shares are reliably broadcast, which may incur overhead, the attacker cannot disseminate different coin shares for the same coin instance but there is no guarantee which shares a correct process uses (if $t+1 < n$). Hence, reliable broadcast of shares does not ensure CC-Agreement in the light of broken TEEs.

DAG-based protocols use the common coin in the context of their consensus mechanism. If CC-Unpredictability is broken, an attacker can foresee coin values and potentially bias consensus decisions. If CC-Agreement is broken, different processes may use different

coin values, leading to differing consensus decisions which typically is a safety violation. In conclusion, for the use cases in this dissertation, the naive TEE-based coin is easier to implement, more robust, and for coin share generation and validation equally efficient as the TEE-based Cachin coin. For coin computation, it is roughly twice as costly as the TEE-based Cachin coin.

4.4 TEE-Rider: TEE-Based Asynchronous Atomic Broadcast

In this section, we propose TEE-Rider, an asynchronous, DAG-based, hybrid fault-tolerant atomic broadcast protocol. TEE-Rider builds upon DAG-Rider, a seminal proposal by Keidar et al. [Kei+21], and optimizes the algorithm for the hybrid fault model. First, we introduce and explain the DAG-Rider protocol as the basis for TEE-Rider. Next, we define the system model and the assumptions we make for TEE-Rider. Then, we present the following contributions:

1. We change the quorum size to $\lfloor \frac{n}{2} \rfloor + 1$ to increase the fault tolerance of the protocol. In this step, we show that DAG-Rider itself can withstand t Byzantine faults with $n > 2t$ processes if the used reliable broadcast has a fault tolerance of at least $n > 2t$. It follows that DAG-Rider withstands omission faults with a fault tolerance of $n > 2t$.
2. DAG-Rider distinguishes two edge types: strong and weak edges. We show that it is safe to not rely on weak edges if a fair network scheduler can be assumed thereby reducing the computational complexity.
3. DAG-Rider operates in rounds; rounds are grouped in waves. We show that it is not possible to reduce the wave size to less than four rounds without harming the proof of the AB-Agreement property.
4. Finally, we list and describe the full protocol instantiated with the USIG-based causal order broadcast (see Section 4.2) to improve the communication complexity. The DAG can be used to enforce the causal history which reduces the overhead of the required reliable broadcast data structures.

The proofs were, except for the removed weak edges and the claims on expected latencies, previously published by Leinweber and Hartenstein [LH23]. The counter example for the wave size was previously published by Leinweber and Hartenstein [LH25].

4.4.1 Background: DAG-Rider

While there were proposals for asynchronous atomic broadcast, e.g., by Cachin et al. [Cac+01; CP02; CT05], until the advent of the HoneyBadgerBFT protocol family [Mil+16], asynchronous protocols were considered impractical and practical protocols were based on the partially synchronous model. HoneyBadgerBFT built upon the work by Cachin et al. and introduced a practical asynchronous atomic broadcast protocol based on reliable

broadcast, binary agreement, and a common coin. The team around Alexander Spiegelman invented the DAG-Rider protocol family [Kei+21; Dan+22; Spi+22; Spi+24; Aru+25; Ton+25] that became the state of the art for high throughput asynchronous and, later, partially synchronous atomic broadcast.

DAG-Rider [Kei+21] was the first asynchronous atomic broadcast protocol that builds upon a deterministically constructed directed acyclic graph (DAG) that is structured in rounds. DAG-Rider assumes an asynchronous reliable broadcast primitive (e.g., Bracha's reliable broadcast) and a common coin. The unique feature of the DAG-Rider atomic broadcast is that the reliable broadcast is the only communication-based building block. In particular, DAG-Rider does not rely on a strong validity consensus building block. Consensus is achieved by interpreting the state of the DAG whenever the DAG construction made sufficient progress. In the following, we will first give an overview on DAG-Rider and introduce the DAG-Rider vocabulary. Then, we explain the DAG-Rider protocol consisting of *DAG construction* and the *consensus logic*. Finally, we will provide the intuition behind the correctness of the protocol. For the description we assume, without loss of generality, that $n = 3t + 1$, making $2t + 1$ the quorum size.

Overview and Vocabulary

In the following, we introduce the core concepts of DAG-Rider.

Round-Structured DAG The DAG captures the communication history of all processes. Each process maintains a local copy of the DAG. Processes disseminate protocol messages by using the reliable broadcast primitive. Hence, each process participates in n reliable broadcast instances, one for each process. The DAG is structured in rounds and each round contains maximally one vertex – that is one protocol message – per process. The vertex data structure carries a broadcast message ($v.block$), a round number ($v.r$), a process identifier naming the process who created the vertex ($v.p$), and a binary flag indicating if the broadcast message was delivered ($v.delivered$). A vertex can only be added to the graph if all its outgoing edges are present in the local DAG.

Edges and Paths The vertices of the DAG are connected via directed edges. Edges always point from a vertex of a higher round to a vertex of a lower round. DAG-Rider distinguishes strong and weak edges. A **strong edge** connects a vertex of round r to a vertex of round $r - 1$. A **weak edge** connects a vertex of round r to a vertex of round $r' \leq r - 2$. A correct process will only select vertices as edges if they are already part of its local DAG. For a vertex to be valid, it requires to have at least $2t + 1$ outgoing strong edges. Outgoing weak edges are optional. If a vertex v of round r reaches a vertex v' of round $r' < r$ by only using strong edges, the resulting path is called a **strong path**. A **path** instead contains at least one weak edge.

Waves and Commit Rules Every disjoint four consecutive broadcast rounds are grouped into a **wave** w. A wave w is completed, when its last round is completed. When a process completes a wave, it checks if it can commit a wave directly: The process selects a **wave root** using a common coin (Definition 4.1) from the first round of the wave. The **direct commit rule** requires that the wave root is reachable via a strong path from at least $2t + 1$ vertices of the last round of the wave. If the direct commit rule is not fulfilled, the wave cannot be committed and the consensus logic is aborted. There is, however, the guarantee that a correct process will eventually complete a wave that can be committed directly. The process will then check for uncommitted waves if the **retrospective commit rule** is fulfilled: The process selects the wave root of the uncommitted wave using the common coin and checks if the wave root has a single strong path to the wave root of the next wave which can be committed.

DAG Construction

In Algorithm 4, we present the original DAG-Rider protocol based on [Kei+21, Algorithms 1–3]. Algorithm 5 lists the utility functions used in Algorithm 4. Each process p_i maintains a local round counter r that is incremented whenever the process can start a new round. Furthermore, each process p_i maintains a local message buffer that contains messages that are to be atomically broadcast by p_i and a local vertex buffer that contains vertices that were received by a reliable broadcast but are not yet part of the DAG. When process p_i invokes A_BROADCAST(m) to atomically broadcast a message m, the message is stored in the message buffer (l. 21). DAG-Rider has a "main loop" (ll. 7–19) that is endlessly repeated and consists of two parts: adding received vertices to the DAG (ll. 8–10) and the round transition logic including the construction of new vertices and triggering the consensus logic (ll. 11–19). First, we explain how a vertex is created. Then, we explain how vertices are received and handled and then how rounds can be completed.

Vertex Creation Whenever a process p_i can transition to a new round r, it constructs a new vertex v (ll. 13–19). The first message of the message buffer is selected as the vertex' payload and stored in $v.block$. The strong edges of the vertex are those vertices of round $r - 1$ that p_i already added to its DAG; p_i must select at least $2t + 1$ vertices as strong edges (l. 25). Weak edges are selected as follows: p_i searches all vertices v' of rounds $r' \leq r - 2$ that have no *path*, i.e., no sequence of strong and weak edges, from v to v'. Those vertices are added as weak edges to v (ll. 16–18). The number of weak edges is not limited. Subsequently, p_i reliably broadcasts the vertex v to all other processes using the reliable broadcast instance in which p_i is the sender (l. 19).

Vertex Reception and Vertex Validation DAG-Rider operates in asynchrony. This implies that, while process p_i currently being in round r, vertices for rounds $v.r < r$ or $v.r > r$ may be received any time. To this end, received vertices are not added directly to the DAG but buffered in a vertex buffer. When a process p_j receives a vertex v created by $v.p$, it verifies that the vertex is valid. As DAG-Rider operates above the reliable broadcast,

Algorithm 4 DAG-Rider Atomic Broadcast: Main Loop for Process p_i [Kei+21, Algs. 2–3]

state DAG: array of array of vertices, $DAG[0]$ initialized with genesis vertices
state r: $\mathbb{N}$, initialized with 1
state $decidedWave$: $\mathbb{N}_0$, initialized with 0
state $messageBuffer$: queue of messages, initialized empty
state $vertexBuffer$: set of vertices, initialized empty
state $coin$: common coin instance with TOSS(·)
while true **do**
    for $v \in vertexBuffer$ **do**
        if $v.r \leq r \wedge \forall u \in v.strong \cup v.weak\colon u \in \cup_{r' \geq 0} DAG[r']$ **then**
            $DAG[v.r][v.p] \leftarrow v$; $vertexBuffer$.remove(v)
    if $|DAG[r]| < 2t + 1$ **then continue**
    if $r \mod 4 = 0$ **then** waveReady($\frac{r}{4}$)
    $r \leftarrow r + 1$
    wait until $\neg messageBuffer$.isEmpty()
    $v \leftarrow$ new vertex; $v.block \leftarrow messageBuffer$.pop(); $v.strong \leftarrow DAG[r-1]$
    for $r' \leftarrow r - 2$ down to 1 **do**
        for $u \in DAG[r']$ **do**
            if $\neg$path(v, u) **then** $v.weak$.add(u)
    r_broadcast(r, v)
function A_BROADCAST(m)
    $messageBuffer$.insert(m)
// The r_deliver *handler is triggered whenever a reliable broadcast instance (of n) delivers.*
upon R_DELIVER(p_k, r', v)
    $v.p \leftarrow p_k$; $v.r \leftarrow r'$; $v.delivered \leftarrow$ False
    if $|v.strong| \geq 2t + 1$ **then** $vertexBuffer$.add(v)
function WAVEREADY(w)
    $v_w \leftarrow DAG[\text{round}(w, 1)][coin.\text{toss}(w)]$
    if $v_w = \perp \vee |\{u \mid u \in DAG[\text{round}(w, 4)]\colon \text{strongPath}(u, v_w)\}| < 2t + 1$ **then**
        return
    $committedRoots \leftarrow$ new stack; $committedRoots$.push(v_w)
    for $w' \leftarrow w - 1$ down to $decidedWave + 1$ **do**
        $v_{w'} \leftarrow DAG[\text{round}(w', 1)][coin.\text{toss}(w')]$
        if $v_{w'} \neq \perp \wedge \text{strongPath}(v_w, v_{w'})$ **then**
            $committedRoots$.push(u); $v_w \leftarrow v_{w'}$
    while $\neg committedRoots$.isEmpty() **do**
        $v_w \leftarrow committedRoots$.pop()
        $verticesToDeliver \leftarrow \{u \mid u \in \cup_{r' > 0} DAG[r']\colon \text{path}(v_w, u) \wedge \neg u.delivered\}$
        for $u \in verticesToDeliver$ in deterministic order **do**
            $u.delivered \leftarrow$ True
            a_deliver $(u.p, u.r, u.block)$
    $decidedWave \leftarrow w$

Algorithm 5 DAG-Rider Atomic Broadcast: Utility Functions [Kei+21, Alg. 1]

```
1: function PATH(v, u)
2:     return exists a sequence of vertices (v_1, v_2, ..., v_k) ∈ ∪_{r'≥0} DAG[r'] such that
3:     v_1 = v ∧ v_k = u ∧ ∀i ∈ [2, k] : v_i ∈ v_{i−1}.strong ∪ v_{i−1}.weak
4: function STRONGPATH(v, u)
5:     return exists a sequence of vertices (v_1, v_2, ..., v_k) ∈ ∪_{r'≥0} DAG[r'] such that
6:     v_1 = v ∧ v_k = u ∧ ∀i ∈ [2, k] : v_i ∈ v_{i−1}.strong
7: function ROUND(w, i)
8:     if i < 1 ∨ i > 4 ∨ w < 1 then abort
9:     return 4(w − 1) + i
```

the only validity check is that v has at least $2t + 1$ strong edges. In particular, it is not necessary perform authenticity checks or to check if a vertex for the DAG slot, which is the combination of round and vertex creator, was already received. If this check is successful, p_i adds v to its vertex buffer (ll. 23–25). Otherwise, the vertex is discarded.

Vertex Handling and Round Completion As part of the main loop, DAG-Rider regularly performs the "buffer walk" (ll. 8–10): Process p_i checks for every vertex in the buffer if it can be added to the DAG. This is possible if p_i has already reached the round of the vertex, i.e., $r \geq v.r$. Moreover, it is required that all vertices that v references as strong and weak edges are part of p_i's local DAG (l. 9). By transitively applying this rule, the *complete* ancestry of a newly added vertex is guaranteed to be part of the DAG. The combination of DAG construction rules and reliable broadcast of vertices yields a causal order reliable broadcast of vertices; the DAG encodes the causal order of messages. When p_j was able to add $2t + 1$ vertices to round r of its local DAG, it *completes* round r and transitions to round $r + 1$ and starts the round with the creation of a new vertex (ll. 11–19).

Consensus Logic

Every disjoint four consecutive broadcast rounds are grouped into a wave. Waves are indexed starting from 1; round r belongs to wave $w = \lceil \frac{r}{4} \rceil$. Rounds within a wave can be addressed using ROUND(w, i) (see Algorithm 5, ll. 7–9). In the consensus logic, process p_i tries to derive a total order of vertices from rounds $\leq$ round$(w - 1, 4)$ and, thus, of messages as vertex payloads. In the following, we first describe when a wave is completed and how wave roots are selected. Then, we describe when and how waves are committed. Finally, we describe how the total order is derived.

Wave Completion and Wave Roots When process p_i completes round$(w, 4)$ of some wave w, it completes wave w and can start the consensus logic (call to WAVEREADY$(\cdot)$ in l. 12). Each wave w has exactly one wave root v_w which is selected from round$(w, 1)$ using the common coin (l. 27). Wave roots play a crucial role in ensuring the safety of DAG-Rider:

Wave roots are used as start points for the graph traversal that defines the total order of vertices (see below).

Wave Commit: Direct and Retrospective Commits If the wave root v_w of wave w that is to be committed was not yet received or there are no $2t+1$ vertices in round$(w, 4)$ that reach v_w by using a strong path, the direct commit rule is not fulfilled and the consensus logic is aborted (l. 28). If the direct commit rule is fulfilled, process p_i can continue. Process p_i will then check if there are uncommitted waves. Uncommitted waves are such waves for which the direct commit rule was not fulfilled when the wave was completed. Process p_i will then iterate over all uncommitted waves w' from the most recent to the oldest, i.e., in descending order of wave index (ll. 31–34). For the retrospective commit rule to be fulfilled the wave root $v_{w'}$ of wave w' must be part of the local DAG. Moreover, wave root $v_{w'}$ must have a strong path to the wave root v_w of the wave $w > w'$ which p_i could either directly or retrospectively commit before verifying the retrospective commit rule for w' (l. 33). All wave roots which can be committed are added to the *committedRoots* stack. The retrospective commit rule is only checked once for each uncommitted wave. In future commits, the search for uncommitted waves stops before the wave that could last be committed directly (variable *decidedWave*, ll. 31 and 41). If the retrospective commit rule is not fulfilled for an uncommitted wave w' when the check is carried out, the corresponding wave root $v_{w'}$ is *never* committed.

Deriving a Total Order The last part of the consensus logic is to determine the total order of vertices. To this end, process p_i will perform a deterministic graph traversal, e.g., a depth-first search, starting from the wave roots in the *committedRoots* stack (ll. 35–40). Because waves are traversed in descending order when checking the retrospective commit rule and wave roots are added to a stack, the wave root of the wave with the smallest index, i.e., the oldest uncommitted wave, is the top-most element and serves as the starting point for the very first graph traversal. The graph traversal uses both strong and weak edges when moving through the DAG. DAG-Rider `a_delivers` the messages as the payload of the vertices in $v.block$ in the order the vertices are traversed[3]. The "most recent" message, i.e., the message that is the payload of the vertex with highest round identifier, is the payload of the wave root v_w of wave w for which the direct commit rule is fulfilled. Finally, p_i sets $decidedWave := w$ (l. 41).

Correctness

The atomic broadcast properties (Definition 2.3) AB-Agreement and Total Order rely on the DAG structure and the consensus logic. The properties AB-Validity and AB-Integrity

[3] As vertices are traversed "backwards" in the graph, the messages are delivered in reverse order. Moreover, vertices of the same round have no relation to each other. The graph search algorithm must define an order for these vertices (e.g., sorted by process identifiers). This may seem strange but is not in conflict with the atomic broadcast properties.

follow, more or less directly, from the reliable broadcast primitive. In the following, we will provide an intuition for the correctness of the DAG-Rider protocol.

AB-Agreement For AB-Agreement, we have to make sure that, if a correct process delivers a message (which is the payload of a vertex), eventually every other correct process does alike and that messages are equal. A correct process delivers a message as part of the consensus logic which is invoked whenever a wave is completed. As there are at least $2t+1$ correct processes that propose a vertex for a round, by the reliable broadcast properties, all correct processes will receive at least $2t + 1$ vertices for a round. Hence, under the assumption of indefinite broadcast requests at each correct process, a correct process will eventually complete the same rounds as every other correct process. Consequently, it only remains to argue that a correct process, when it terminates a wave, has the possibility to start the consensus logic. A correct process can start the consensus logic if the direct commit rule is fulfilled. For the direct commit rule to be fulfilled, the wave root v_w must be part of the DAG and at least $2t + 1$ vertices in round$(w, 4)$ must reach v_w using a strong path. Using the **common core**, get-core, or gather argument [AW04, Section 14.3], we can show that there are at least $2t + 1$ vertices from round$(w, 1)$ that are reachable from a set of $2t + 1$ vertices in round$(w, 4)$ using a strong path in every wave w. Hence, when the common coin is fair, the probability that the direct commit rule is fulfilled is at least $\frac{2t+1}{3t+1}$. Applying the Bernoulli distribution, it follows that a correct replica will be able to directly commit every 1.5 waves in expectation and we can conclude that a correct process will eventually commit with probability 1. At this point, we know that eventually every correct process will be able to commit a wave directly. This, however, only allows to conclude that a correct process will deliver *some* messages. For AB-Agreement, we need to ensure that correct processes deliver the *same* messages. When a correct process commits, it will deliver all payloads, i.e., messages, of vertices in its DAG that were not yet marked as delivered. As vertices are reliably broadcast and vertices can only be added when their ancestry is part of the DAG, all correct processes will eventually know the ancestry of all received vertices, thus, allowing them to add all vertices delivered by the reliable broadcast to the DAG. In particular, it is guaranteed that the local DAG of every correct process contains the complete ancestry of a vertex v when v is added to the DAG. The RB-Agreement property ensures that every DAG slot – the combination of round and vertex creator, i.e., process – contains the exact same vertex for all correct processes. The combination of the RB-Agreement property and the knowledge of the full ancestry implies that when a correct process p_i is able to directly commit a wave root v_w, process p_i reaches the *exact same* vertices, i.e., v_w's ancestry, when it starts a graph traversal from v_w. Please note that the retrospective commit rule is only relevant for the Total Order property (see below). For AB-Agreement, it is sufficient to show that the same vertices are traversed; the traversal order is not relevant. Consequently, it is irrelevant whether a process starts the graph traversal as part of a direct or a retrospective commit. In any case, the exact same vertices will be traversed. We can conclude that the AB-Agreement property holds.

AB-Validity For AB-Validity, we have to show that if a correct process broadcasts a message, every other correct process will eventually deliver that message. A correct process can only deliver a message if it was part of a vertex that was added to the DAG. Correct processes can only add vertices to the DAG if they received the vertex via the reliable broadcast primitive and know the vertex' ancestry. When a correct process p_i creates a vertex v, it will create a valid vertex for which it knows the ancestry. Thus, said process p_i will be able to add the vertex to its own DAG. By the reliable broadcast properties, all correct processes will receive the vertex and add it to their DAG. Any correct process p_j will now add v, if possible, as a strong edge to its next vertex. In any case, p_j will add v as a weak edge to its next vertex if v is not already reachable from the new vertex. Because of the weak edges, v will eventually be in the ancestry of *any* vertex created by a correct process. Figure 4.5 in Section 4.4.4 shows in an example the importance of weak edges in DAG-Rider. In particular, v will eventually be in the ancestry of a wave root v_w created by a correct process. Following the argument for agreement, v_w will eventually be committed which implies that v will be delivered.

AB-Integrity For AB-Integrity, we have to show that a correct process delivers a tuple (p, r, m), where p is the process identifier, r is the round, and m is the message, at most once. DAG-Rider uses the round as the tag of the reliable broadcast. Thus, by RB-Integrity, any process can only broadcast a single vertex for a round. In the consensus logic, a process marks a vertex as delivered when it delivers the message and it will not traverse vertices that were marked twice. Thus, a process will never deliver the same tuple twice.

Total Order From the agreement argument, we know that all correct processes will eventually deliver the same messages. It remains to be shown that all correct processes deliver the messages in the *same order*. The order is primarily determined by the start points of the graph traversals. Thus, to ensure total order, all correct processes must use the *exact same* start points. Put differently, if a correct process uses a wave root v_w as a start point, every other correct process must use the same vertex v_w as a start point. If all correct processes observe the direct commit rule to be fulfilled for the same waves, this is trivially the case. Asynchrony, however, may lead to a situation where a correct process p_i observes the direct commit rule to be fulfilled for wave w while another correct process p_j does not: The asynchronous link may have delayed v_w, so that the DAG of p_j does not contain v_w at the completion of wave w. In this case, p_i will use v_w as a start point while p_j will not. The retrospective commit mechanism must ensure that p_j will eventually use v_w as a start point before p_j commits any other "younger" wave, i.e., a wave with a higher wave index. For uncommitted waves $w' > \mathit{decidedWave}$, a correct process will check exactly once – *decidedWave* is incremented after a direct commit – if the retrospective commit rule is fulfilled. This will be done when the process is able to directly commit a wave $w > w'$. The graph construction of DAG-Rider must ensure that whenever a correct process p_i is able to directly commit a wave w', any other correct process p_j must in any case be able to (1) to directly commit w as well, or (2) to observe the retrospective commit rule to be fulfilled when directly committing a "younger" wave w,

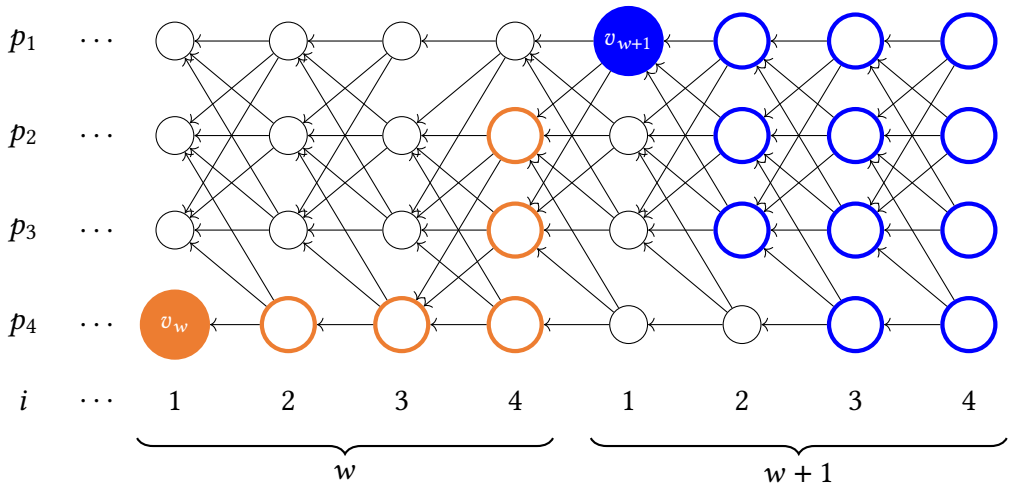


Figure 4.3: Example DAG as constructed by $n = 4$ DAG-Rider processes of which at maximum 1 may be faulty illustrating the interplay of direct and retrospective commit rules. Shown is the "global" state of the graph, i.e., after every process eventually received every vertex of 8 rounds. The x axis denotes $i := (r \mod 4) + 1$, i.e., the round within the wave. For simplicity, all vertices are valid. For readability, weak edges are left out. Vertices that have a path to the root of their wave are drawn slightly bigger and are colored according to the wave root. The vertex created by process p_4 for round$(w, 1)$ was selected as wave root v_w for wave w. From the strong edges process p_1 selected for its vertex for round$(w + 1, 1)$, we can conclude that p_1 did *not* observe the direct commit rule for wave w to be fulfilled when it completed the wave. Consequently, p_1 did not deliver any vertices when completing wave w. All other process, however, observe the vertices created by p_2, p_3, and p_4 for round$(w, 4)$ to have a strong path to v_w. Thus, they all observe the direct commit rule to be fulfilled and deliver all vertices that have a path to v_w when completing wave w. The direct commit rule ensures that *any* vertex of rounds $r \geq$ round$(w + 1, 1)$ have a strong path to v_w. Moreover, the direct commit rule is fulfilled for wave $w + 1$ for all processes as all vertices of round$(w + 1, 4)$ have a strong path to the selected wave root v_{w+1} allowing all processes to directly commit wave $w + 1$. Because the complete ancestry of a vertex v must be part of the DAG to be able to add v to the DAG, p_1 can neither propose its own vertex for round$(w + 1, 1)$ nor complete said round before receiving and adding v_w to its DAG as at least 2 of the vertices p_1 must select as strong edges have v_w in their ancestry. Hence, when p_1 completes round$(w + 1, 4)$ and, thus, wave $w + 1$, it will observe the retrospective commit rule for wave w to be fulfilled. Consequently, p_1 will first start a graph traversal from v_w and then from v_{w+1} which ensures the Total Order property.

i.e., $w > w'$. As explained above, asynchrony makes it impossible to ensure (1). Instead, we have to rely on the retrospective commit mechanism (2). To this end, DAG-Rider ensures a property we call the **wave root connectivity** property: The direct commit rule enforces that a wave root v_w of wave w which is selected from round$(w, 1)$ can only be directly committed if at least $2t + 1$ vertices of round$(w, 4)$ have a strong path to v_w. We define the set of these vertices of round$(w, 4)$ that have a strong path to v_w as the set $V, |V| \geq 2t + 1$. For a process p_i to create a valid vertex v_{p_i} for round$(w + 1, 1)$, p_i must select at least $2t + 1$ out of $3t + 1$ vertices from round$(w, 4)$ as strong edges. By quorum intersection, process p_i will at least select $t + 1$ vertices from V. Thus, it is guaranteed that in any case, v_{p_i} will have a strong path to v_w. Hence, *any* valid vertex of round$(w + 1, 1)$ will have a strong path to v_w. In particular, *any* valid vertex of any round $r \geq$ round$(w + 1, 1)$ will have v_w in its ancestry. Because any valid vertex can only be added to the DAG if its complete ancestry is already part of the DAG, any vertex of any round $r \geq$ round$(w + 1, 1)$ can only be added to the DAG if v_w was already received and added to the DAG. This includes any

possible wave root of any future wave. Thus, when a correct process p_j was not able to directly commit wave w' with wave root $v_{w'}$ while correct process p_i directly committed w', p_j will have $v_{w'}$ in its DAG when it completes any wave $w > w'$. In particular, this means that p_j will observe the retrospective commit rule to be fulfilled for wave w' when it directly commits a wave $w > w'$. Hence, all correct processes will use in any case the same wave roots as start points for their graph traversals. A correct process will order the waves for which the retrospective commit rule was fulfilled from the oldest to the newest (the stack of committed roots if traversed last-in first-out). This ensures that all correct processes will start their graph traversals from the same wave roots in the same order. Figure 4.3 exemplarily illustrates the importance of the direct and retrospective commit rules. In conclusion, all wave roots that are committed by any correct process are connected via strong paths ensuring that all correct processes will use the same wave roots as start points for the graph traversals which enforces the total order property.

4.4.2 System Model

We consider a system of n processes $P := \{p_1, \ldots, p_n\}$ that aim to implement an atomic broadcast primitive. The processes are connected via an asynchronous network; the processes communicate via secure, perfect point-to-point links (see Sections 2.2 and 2.4). Thus, messages can be reordered and arbitrarily delayed but not dropped. Each process is equipped with a trusted execution environment (TEE, see Definition 2.1). Of the n processes, at most $t < \frac{n}{2}$ processes can be Byzantine faulty. The TEE is assumed to only fail by crashing; its internal state is lost when the TEE crashes. If not explicitly stated, code is not executed inside the TEE.

4.4.3 Reducing the Quorum Size – Increasing the Fault Tolerance

In this subsection, we show that DAG-Rider can be adapted to withstand t Byzantine faults with a fault tolerance of $n > 2t$ processes if a reliable broadcast with a fault tolerance of at least $n > 2t$ (e.g., as described in Section 4.2), and a common coin with reconstruction threshold of t is used. DAG-Rider (Algorithm 4) uses a quorum condition in exactly three places: (1) in l. 11 as a condition to complete a round, (2) in l. 25 to verify that a vertex is valid, and (3) in l. 28 to verify that the direct commit rule is fulfilled. We change the required quorum size at all three occurrences to $\lfloor \frac{n}{2} \rfloor + 1$. A quorum intersection argument is required to prove Total Order (wave root connectivity) and AB-Agreement (common core argument). In the original DAG-Rider paper [Kei+21], the wave root connectivity is proven in Lemma 1 and the common core argument is proven in Lemma 2. In the following, we formally specify and prove both lemmas for a quorum size of $\lfloor \frac{n}{2} \rfloor + 1$. Both proofs were previously published by Leinweber and Hartenstein [LH23].

Lemma 4.2 (Wave root connectivity)**.** *If a correct process $p_i \in P$ commits the wave root v_w of a wave w when it completes wave w in* round$(w, 4)$*, then* any *valid vertex v' of* any *process $p_j \in P$ broadcast for a round $r \geq$* round$(w + 1, 1)$ *will have a strong path to v_w. In*

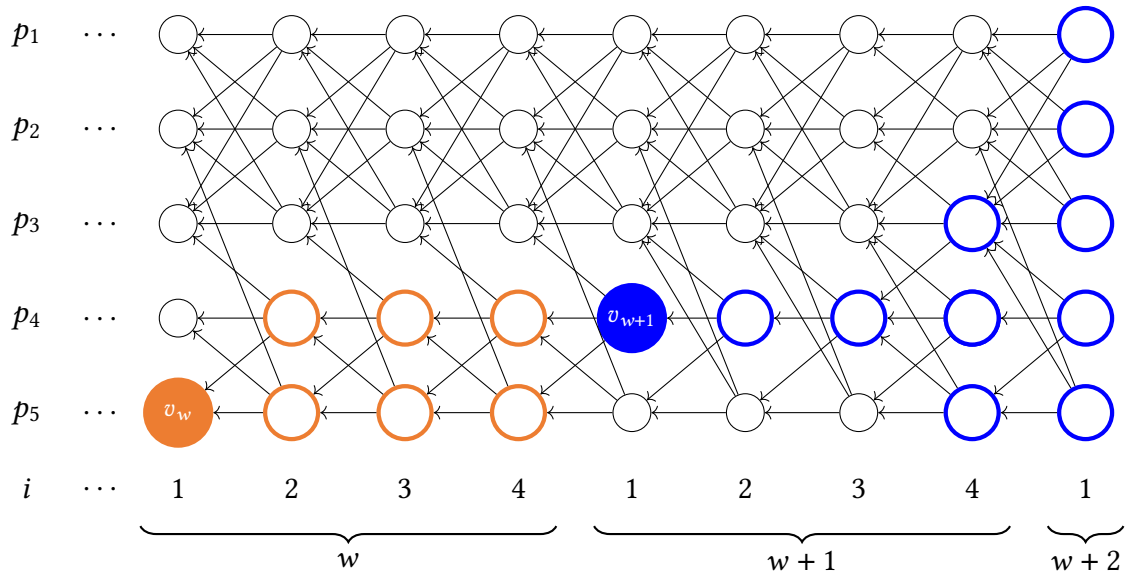


Figure 4.4: Example DAG as constructed by $n = 5$ TEE-Rider processes of which at maximum 2 may be faulty. Shown is the "global" state of the graph, i.e., after every process eventually received every vertex of 9 rounds. The x axis denotes $i := (r \mod 4) + 1$, i.e., the round within the wave. For simplicity, all vertices are valid. The direct commit rule is not fulfilled for any process for wave w as the wave root v_w is only reached by two vertices of round$(w, 4)$. The direct commit rule is fulfilled for processes p_3, p_4, and p_5 for wave $w+1$ (wave root v_{w+1}). Vertices that have a path to the root of their wave are drawn slightly bigger and are colored according to the wave root. The direct commit rule ensures that a correct process can commit a wave retrospectively if it was not able to commit when it finished the wave. Consequently, p_1 and p_2 will eventually commit wave $w + 1$. Since v_{w+1} has a path to v_w, wave w will be committed retrospectively although no process observed the direct commit rule for wave w to be fulfilled.

particular, the wave root $v_{w'}$ of any wave $w' > w$ that can be committed will have a strong path to v_w.

Proof. Since p_i commits v in round$(w, 4)$, the direct commit rule is fulfilled (l. 28): $\exists U \subseteq DAG[\text{round}(w, 4)] : |U| \geq \lfloor \frac{n}{2} \rfloor + 1 \wedge \forall u \in U : \text{strongPath}(u, v_w)$. A valid vertex must reference at least $\lfloor \frac{n}{2} \rfloor + 1$ distinct vertices of the previous round with a strong edge (l. 25). Thus, a process $p_j \in P$ broadcasting a valid vertex v_j for round$(w + 1, 1)$ selected at least $\lfloor \frac{n}{2} \rfloor + 1$ vertices of round$(w, 4)$ as strong edges for v_j. Any two subsets of size $\lfloor \frac{n}{2} \rfloor + 1$ of a superset of size n intersect in at least one element. Thus, every valid vertex of a process broadcast for round$(w + 1, 1)$ must have at least one edge to a vertex of U, and, via U to v_w. As every valid vertex of round$(w + 1, 1)$ has a strong path to v_w and every valid vertex of round$(w + 1, 2)$ connects to at least $\lfloor \frac{n}{2} \rfloor + 1$ vertices of round$(w + 1, 1)$, by induction, any valid vertex v'_j of any process $p_j \in P$ broadcast for a round $r \geq$ round$(w + 1, 1)$ has a strong path to v_w. Because wave roots that can be committed are valid vertices, the wave root $v_{w'}$ of any wave $w' > w$ will have a strong path to v_w. □

An example for a resulting graph with $n = 5$ processes, i.e. $t \leq 2$, is shown in Figure 4.4. The example shows the impact of the direct commit rule: If a process can directly commit, the wave root will have a path to any future valid vertex. This allows processes p_1 and p_2

to commit wave $w+1$ retrospectively when committing a wave $\geq w+2$, even though they were not able to commit wave $w+1$ directly, thus, maintaining the total order property.

Lemma 4.3 (Common core)**.** *When a correct process $p_i \in P$ completes* round$(w,4)$ *of wave w, then* $\exists V_1 \subseteq DAG[\text{round}(w,1)], V_4 \subseteq DAG[\text{round}(w,4)] : |V_1| \geq \lfloor \frac{n}{2} \rfloor + 1 \wedge |V_4| \geq \lfloor \frac{n}{2} \rfloor + 1 \wedge (\forall v_1 \in V_1, \forall v_4 \in V_4 : \text{strongPath}(v_4, v_1))$.

Proof. By use of reliable broadcast and validity checks in ll. 9 and 25, faulty processes are limited to omission faults. Thus, the *get-core* argument of Attiya and Welch [AW04, Sec. 14.3.1] still holds [AW04, Sec. 14.3.3]: Let $A \in \{0,1\}^{n \times n}$ be a matrix that contains a row for each possible vertex of round$(w,3)$ and a column for each possible vertex of round$(w,2)$. Let $A[j,k] = 1$ if the vertex of process p_j of round$(w,3)$ has a strong edge to the vertex of process p_k of round$(w,2)$ or p_j sends no vertex (or an invalid one) but p_k sends a valid vertex for round$(w,2)$. As there are at least $\lfloor \frac{n}{2} \rfloor + 1 \leq n - f$ correct processes, each row of A contains at least $\lfloor \frac{n}{2} \rfloor + 1$ ones and A contains at least $n(\lfloor \frac{n}{2} \rfloor + 1)$ ones. Since there are n columns, there must be a column l with at least $\lfloor \frac{n}{2} \rfloor + 1$ ones. This implies there is a vertex v_l by process p_l in round$(w,2)$ s.t. $\exists V_3 \subseteq DAG[\text{round}(w,3)] : |V_3| \geq \lfloor \frac{n}{2} \rfloor + 1 \wedge \forall v_3 \in V_3 : \text{strongPath}(v_3, v_l)$. As at most f vertices in V_3 belong to faulty processes that may commit send omission faults for round$(w,3)$ and $\lfloor \frac{n}{2} \rfloor + 1 \geq t + 1$, by quorum intersection at least one vertex of V_3 is received by any correct process $p_j \in P$ before it sends its vertex for round$(w,4)$. Thus, every valid vertex in $DAG[\text{round}(w,4)]$ has at least one strong edge to a vertex of V_3. Since v_l must be valid and thus has a strong edge to each vertex of a set $V_1 \subseteq DAG[\text{round}(w,1)], |V_1| \geq \lfloor \frac{n}{2} \rfloor + 1$, any valid vertex of rounds $r \geq$ round$(w,4)$ has a strong path to every vertex, including V_1, reached by v_l via strong paths. Please note that the construction of the set V_1 is valid for all correct processes that complete the wave and, thus, represents the 'common core'. □

Due to the changed quorum size, we have to adapt [Kei+21, Claim 6] to the following lemma:

Lemma 4.4 (Expected commit latency)**.** *For every correct process p_i and for every wave w, the expected number of waves until p_i observes the direct commit rule to be fulfilled is 2.*

Proof. By Lemma 4.3, the probability for the direct commit rule to be fulfilled for an arbitrary correct process p_i is at least

$$\lim_{n \to \infty} \left(\frac{\lfloor \frac{n}{2} \rfloor + 1}{n} \right) = \frac{1}{2}.$$

By applying the Bernoulli distribution, the expected number of waves until the direct commit rule is fulfilled is 2. □

Lemmas 4.2 and 4.3 are a direct stand-in for the lemmas 1 and 2 in [Kei+21]; Lemma 4.4 replaces Claim 6. Thus, all atomic broadcast properties follow directly from the original proofs.

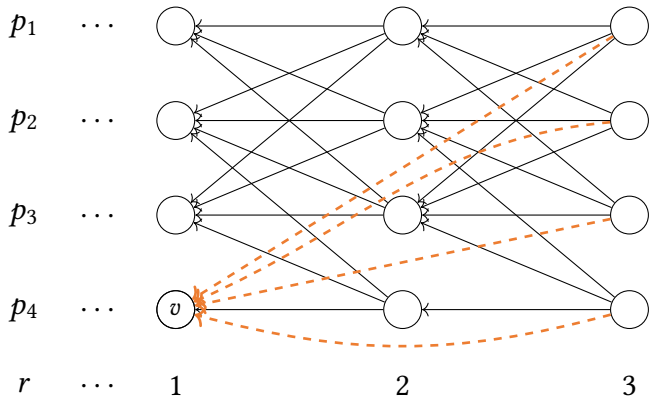


Figure 4.5: Example DAG as constructed by $n = 4$ DAG-Rider processes of which at maximum 1 may be faulty illustrating the importance of weak edges. Shown is the "global" state of the graph, i.e., after every process eventually received every vertex of 3 rounds. The x axis denotes the round index r. For simplicity, all vertices are valid. Strong edges are solid, black arrows. Weak edges are orange, dashed arrows. Vertices of p_4 are only referenced with strong edges of p_4 itself. As a result, the vertices of p_4 are not in the ancestry of vertices by other processes *and*, if the common coin selects a vertex of p_4 as a wave root, the commit rules will not be met. Thus, vertices of p_4 will never be committed. For the example, we assume that p_4 is only delayed by one round and, thus, all processes have added v to their DAG when starting round 3. Thus, starting from round 3, any correct process will observe that v is not reachable and connect its vertex for round 3 with a weak edge to v. This ensures that any v will eventually be in the ancestry of a wave root created by a correct process and, subsequently, will be delivered.

4.4.4 Removing Weak Edges – Reducing Computational Complexity

The weak edges in DAG-Rider have a single purpose: They are required to ensure that, when a correct process broadcasts a message, i.e., a vertex, eventually all correct processes will deliver the vertex (i.e., AB-Validity) [Kei+21, Proposition 4]. Figure 4.5 illustrates an example for the importance of weak edges. While being important for the correctness of DAG-Rider, the weak edges come with two significant trade-offs:

1. Weak edges increase the expected computational complexity of the protocol as every time a vertex is created the reachability of vertices has to be checked.
2. Weak edges make garbage collection impossible as every new vertex received may have a weak edge to a vertex that was already garbage collected [Dan+22, Sec. 8.2].

In the following, we will show that weak edges can be removed from DAG-Rider without losing AB-Validity if a fair network scheduler can be assumed, i.e., if an attacker cannot control the delivery delays of network messages and network delays are uniformly distributed. First, we show that when using the USIG-based causal order broadcast as the reliable broadcast abstraction, correct processes can immediately deliver their own vertices. Then, we show that we can safely remove weak edges from the protocol without sacrificing AB-Validity. Finally, we show that a correct process will be able to deliver its own vertex in expectation after a constant number of waves.

Lemma 4.5 (Immediate delivery of vertices)**.** *When a correct process p_i constructs and broadcasts a vertex v for round r using the USIG-based causal order broadcast, p_i can immediately, i.e., without additional communication, add v to its DAG.*

Proof. A correct process p_i is required to reliably broadcast a new vertex v (Algorithm 4, ll. 19 and 23ff.). By Lemma 4.1, the reliable broadcast abstraction will directly deliver v at p_i. Subsequently, p_i will add v to its vertex buffer (Algorithm 4, l. 25). As v is a valid vertex and p_i only adds vertices as edges which it already added to the DAG (Algorithm 4, l. 15), as soon as the r_deliver handler was invoked (ll. 23–25), v will be added to the DAG in the next iteration of the main loop (ll. 7–10). □

Theorem 4.4 (AB-Validity without weak edges)**.** *If a correct process p_i constructs and broadcasts a vertex v for an arbitrary but fixed round r using the USIG-based causal order broadcast, network message delays are uniformly distributed, and process p_i ensures that p_i's vertex of round $r-1$ is a strong edge of v, then every correct process will eventually deliver v with probability 1.*

Proof. When p_i constructs, broadcasts, and handles a vertex v for round r, by Lemma 4.5, p_i will add v in the next iteration of the main loop. Thus, a simple extension to the check in l. 12, i.e., checking if v is already in the DAG ($DAG[r][p_i] \neq \bot$), and the check in l. 25, i.e., checking that the strong edges contain the creator's vertex of the previous round, ensures that p_i will have v in its DAG before completing round r. Moreover, this check does not conflict with liveness or cause additional latency, as, in the worst case, p_i simply adds all vertices it received for round r to its DAG before adding its own vertex v and completing the round. It follows by induction that the vertices of p_i build a chain made of strong edges. Following the argument for Lemma 4.4, the probability for p_i to directly commit a wave is at least $\frac{1}{2}$. The probability that the wave root v_w is a vertex of p_i is $\frac{1}{n}$. Due to the uniformly distributed network delays, these probabilities are independent and, thus, the probability that p_i commits a wave with a wave root v_w of p_i is at least $\frac{1}{2n}$. Thus, in expectation, p_i will commit a wave with a wave root v_w of p_i after $2n$ waves, i.e., with probability 1, which then will deliver v as part of the ancestry of v_w. □

While Theorem 4.4 gives a sufficient condition for AB-Validity without weak edges, the result to wait $2n$ waves is not promising. However, the result is only an upper bound on the expected time until a correct process p_i delivers its own vertex v for round r:

Lemma 4.6 (Expected time of ancestry inclusion)**.** *Under the assumption that for an arbitrary but fixed v' of round $r+1$ every vertex v of round r has the same probability to be selected as a strong edge, a vertex $\tilde{v}$ of round r created by process p_i will, in expectation, be in the ancestry of every valid vertex v'' of round $r+2$, i.e., every valid v'' will have a strong path to $\tilde{v}$.*

Proof. If we simplify the quorum size, i.e., the required number of strong edges, to $\frac{n}{2}$, we get an estimate for the probability that v' has a strong edge to v using the hypergeometric distribution [Wik25b]:

$$\Pr[X = \tilde{v} \text{ is strong edge of } v'] = \frac{\binom{1}{1}\binom{n-1}{\frac{n}{2}-1}}{\binom{n}{\frac{n}{2}}} = \frac{1}{2}.$$

Now, by construction, every vertex v'_k, i.e., the vertex of p_k for round $r+1$, has a strong edge to v_k, i.e., p_k's vertex for round r. Thus, a vertex of every round $\geq r+2$ has *at least* the same probability of $\frac{1}{2}$ to "hit" a vertex that has a strong path to $\tilde{v}$. Hence, by applying the Bernoulli distribution, in expectation every valid vertex of round $r+2$ has a strong path to v. □

Remark. The probability of $\frac{1}{2}$ is a lower bound. If we use the correct quorum size of $\lfloor \frac{n}{2} \rfloor + 1$, the true probability of inclusion is, depending on n, at least $\frac{1}{2}$ or higher.

Lemma 4.7 (Expected time of delivery)**.** *Let w be the wave of round r and v a vertex created by a correct process p_i for round r. Under the assumption of a uniform distribution of message delays, the probability that p_i will deliver v in expectation not later than when committing wave $w+3$ is 1.*

Proof. By Lemma 4.6, if $r < \text{round}(w, 4)$, we can conclude that the probability of $\tilde{v}$ to be in the ancestry of the wave root v_{w+1} of wave $w+1$ is 1. If $r = \text{round}(w, 4)$, we can conclude that the probability of $\tilde{v}$ to be in the ancestry of the wave root v_{w+2} of wave $w+2$ is 1. By Lemma 4.4, p_i will, in expectation, be able to commit either wave $w+2$ or wave $w+3$ directly. Hence, in expectation, p_i will deliver v not later than when committing wave $w+3$. □

Remark. If uniformly distributed latencies, i.e., a fair scheduler, cannot be assumed, the results above do not hold as there exist executions in which a vertex of a process is never delivered[4]. As we use TEE-Rider in the remainder of the work at hand as the basis for SMR, we do not suffer from this limitation: Depending on the actual client implementation, an SMR client either broadcasts its request to all replicas or uses a timer to broadcast if needed. Both ensures that the client request will be proposed by a correct replica that can, by the design of DAG-Rider-like protocols, not suffer from a network adversary[5]. Moreover, an attacker that controls the links between a majority of correct replicas seems to be unlikely if replicas are sufficiently distributed.

[4] Please note that a wide-area network deployment where some replicas have a significantly higher latency to the remainder of the peer-to-peer network, e.g., when placing replicas around the globe, the fair scheduler assumption is not met.

[5] In DAG-Rider-like protocols, replicas can only make progress if there is sufficient progress in adding new DAG vertices. If a network attacker prevents this progress, the protocol will stall. The common coin and the design of the quorums ensure that there are always correct replicas part of the replica set that is able to make progress. These correct replicas will propose all valid client requests they received.

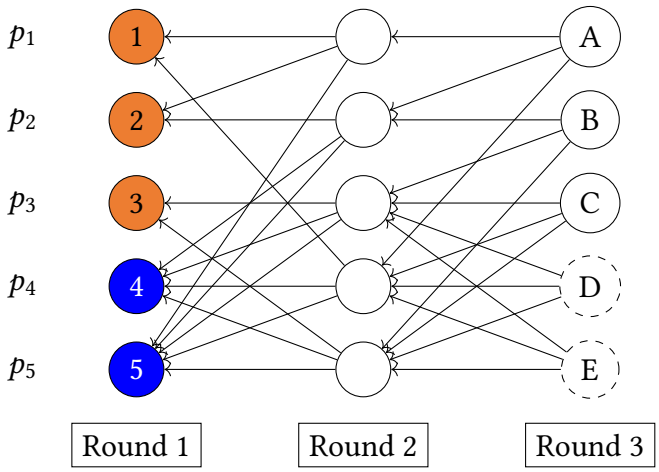


Figure 4.6: Counter example for the common core property with $n = 5$ and a wave length of three rounds. As all vertices are valid, all vertices of round 3 should have a path to a shared subset of round 1 which is of size 3. The maximum shared subset, however, is of size 2 (vertices 4 and 5 in blue). If now a process completes round 3 with vertices A, B, and C of round 3, there will be no common core of size 3 but of size 2. If, however, it completes round 3 with vertices C, D, and E, the common core will be of size 4. This is in conflict with the common core property that requires that for *any combination* of vertices in round 3 the common core is at least of size 3. Thus, for the common core property to be fulfilled, all vertices of round 3 must have an additional path to the same vertex of the vertices 1–3 in orange.

4.4.5 Reducing the Expected Commit Latency?

In DAG-Rider, waves are the construct defining the expected commit latency. A wave consists of four consecutive rounds. If we could reduce the number of rounds per wave, obviously we would reduce the expected commit latency. The wave length of four rounds is required to prove the common core argument (Lemma 4.3) that, in turn, is required to prove AB-Agreement. The underlying get-core construct by Attiya and Welch [AW04, Section 14.3] relies on three steps of all-to-all communication: In the first step, every process sends a message to every other process. In the second step, every process collects $n - t$ messages sent by other processes and broadcasts the collected set of messages. In the third step, every process collects $n - t$ of these sets and broadcasts a union of the sets. It is tempting to compare this scheme to the three phases of reliable broadcast: init, echo, ready. We know that when using a TEE, e.g., a USIG, the ready phase is not required (see Section 4.2). Naively, we could map this optimization and try to reduce the number of rounds per wave to three. However, as we show in the following counter example, this is not possible without breaking the common core property.

Let $n = 5$, the quorum size be $\lfloor \frac{n}{2} \rfloor + 1$, i.e., $t = 2$, and the wave length be three rounds. For the common core property to hold, any combination of three vertices in round 3 must share a set of at least three reachable vertices in round 1, i.e., there is a path between the vertex of round 3 and the vertex of round 1. Figure 4.6 shows a counter example for the common core property. If a correct process finishes the third round with vertices A, B, and C, the common core is of size 2 (vertices 4 and 5 in blue). Put differently, only vertices 4 and 5 are reachable from any valid vertex of round 3. If, however, the process finishes the

third round with vertices C, D, and E, the common core is of size 4: all three vertices have a path to the same vertices 1, 3, 4, and 5 of round 1. The authors of Fides claim a similar property for their protocol (with $n > 2t$ and three rounds per wave): "For every wave w there are at least $f + 1$ [i.e., $t + 1$] vertices in the first round of w that satisfy the commit rule" [Xie+25, Lemma 6]. However, if a process completes the wave as depicted above with vertices A, B, and C in the third round, the direct commit rule, i.e., the condition that the selected wave root is reached by at least $t + 1$ vertices of round 3, is only fulfilled for t vertices of round 1, i.e., the vertices 4 and 5.

It is important to note that the violation of the common core property is no safety violation. Safety solely relies on the direct commit rule. From a safety point of view, it would be possible to try to commit *every* round. However, the common core property is required to proof Lemma 4.4 (Claim 6 in [Kei+21] and Lemma 7 in [Xie+25]). Using a combinatorial argument, the findings in Section 4.4.4 show that when assuming a "fair" network scheduler the common core property is not required to show liveness and that a lower commit latency can be expected. Xie et al. follow a similar argument in their Lemma 8 [Xie+25]. Under adversarial conditions, however, we are not aware of a proof that shows liveness for a DAG-based approach with a fault tolerance of $n > 2t$ that does not rely on the common core property. In fact, Xie et al. show that, under adversarial conditions, the wave length must be greater than two rounds to prevent stalls [Xie+25, Claim 2]. However, this does not rule out the existence of a combinatorial proof for a wave length of 3 assuming an asynchronous adversary. For TEE-Rider, we use a wave length of four rounds and leave the investigation of combinatorial arguments to future work.

4.4.6 TEE-Rider Protocol

In this subsection, we present the complete TEE-Rider protocol, i.e., with changed quorum, removed weak edges and with an "inlined" USIG-based causal order broadcast. The goal of the enclave design is a small trusted computing base and the reduction of costly context switches for efficient execution. The enclave combines the well-known USIG concept to prevent equivocation (see Section 4.2) and a variant of the naive TEE-based common coin to provide randomness (see Section 4.3.1). We describe the protocol in three parts: the main loop of the protocol, the broadcast logic, and the enclave logic.

The main loop of the protocol, i.e., "vertex buffer walk" and consensus logic, is shown in Algorithm 6. Besides the changed quorum sizes and the removed weak edges, the logic is not changed to DAG-Rider. Whenever the process wants to atomically broadcast a message, it calls the A_BROADCAST(·) function (l. 21) which adds the message to the message buffer. In an endless loop, the process checks the vertex buffer for vertices that can be added to the DAG (ll. 9–20). Note that, in a practical implementation, one would not implement this as a busy waiting loop but as an event-driven logic that is triggered by the arrival of a new message. However, with the loop representation, we are not required to handle edge cases (e.g., how the very first vertex of a process is created). When the process was able to add $\lfloor \frac{n}{2} \rfloor + 1$vertices including its own vertex to the current round of the DAG (l. 13), it first checks if it completed a wave (l. 14) and transitions then to the next

round (l. 15ff.). Before it can create a vertex for the new round, it has to have received at least one broadcast request (l. 16). It then creates the vertex, lets it sign by the enclave and broadcasts the vertex in best effort fashion. If the process has completed a wave, it calls the waveReady(·) function (l. 23ff.) which implements the logic to deliver vertices and, thus, messages. First, the naive TEE-based common coin is tossed (l. 24). As the coin can only be tossed once for an instance, the result is stored in the coin store variable. Then, the process verifies if the direct commit rule is met (l. 26–27). If this is the case, it will check undelivered waves for the fulfillment of the retrospective commit rule (ll. 29–32). Finally, the process delivers the vertices wave by wave in deterministic order, updates the *decidedWave* variable, and clears the coin store to limit memory consumption (coins of decided waves are not required anymore) (ll. 33–39).

Algorithm 7 lists the broadcast logic that implements a USIG-based causal order broadcast on top of the DAG. As we track the system state in the DAG and the vertex buffer, in contrast to the USIG-based causal order broadcast, we do not require a separate data structure for the received messages and the expected counter values. Instead, the round is used as the counter value. The use of the round instead of a separate counter value does not break the FIFO property as a process must create exactly one vertex for each round to be correct. First, the process checks if the vertex is valid in terms of edges (l. 5), if a vertex by the creating process for the vertex round was already received, i.e., the USIG-enforced RB-Integrity property (l. 6), and if it is accompanied by a valid enclave signature (l. 7). If any of these checks fail, the vertex is ignored. If the vertex is valid, as we do in the USIG-based causal order broadcast, possible vertex requests are answered (ll. 9–10). Then, the process checks if it has to request vertices referenced as edges (ll. 11–16). Finally, the process can add the vertex to the vertex buffer (l. 17) which will be processed in the next iteration of the main loop. If a process receives a V_REQUEST message (l. 18), it checks if it has a vertex for the requested round and process in its DAG or vertex buffer and answers the request (ll. 19–22). If it does not know a matching vertex, it adds the request to the *receivedRequests* map (l. 23). The utility functions path(·) (ll. 24–26) and round(·) (ll. 27–29) are the same as in DAG-Rider.

In Algorithm 8, we show the enclave logic that implements a USIG-equivalent signature service and the naive TEE-based common coin (see Section 4.3.1) in which validly signed vertices are used as coin shares. The enclave is hosted by a TEE (Definition 2.1). The enclave state is small: it only contains the asymmetric key pair, a round counter for the signatures, an array of public keys, a variable to track the next toss round, and the cryptographically secure pseudorandom number generator (CPRNG). The sign(·) function (ll. 5–8) is equivalent to the USIG. When invoked, it signs the message m together with the current round counter value and subsequently increments the counter. The computeCoin(·) function (ll. 9–17) implements a form of the naive TEE-based common coin. Note that, in contrast to Definition 4.1, the function does not take the coin instance as an argument. The naive TEE-based common coin enforces a total order on the coin tosses by using the round counter of the enclave as a coin instance. In TEE-Rider, we use the vertices of the last round of the wave as the coin shares. To be able to toss a coin, a process has to convince the coin implementation that it made sufficient progress on the consensus layer by providing sufficient validly signed vertices from peering replicas. Thus, the function

Algorithm 6 TEE-Rider Atomic Broadcast for Process p_i: Main Loop

state DAG: array of array of vertices, $DAG[0]$ initialized with genesis vertices
state r: $\mathbb{N}$, initialized with 1
state $decidedWave$: $\mathbb{N}_0$, initialized with 0
state $messageBuffer$: queue of messages, initialized empty
state $vertexBuffer$: set of vertices, initialized empty
state $coinStore$: map of key $\mathbb{N}$ to value $\mathbb{N}$, initialized empty
state $enclave$: enclave instance
state P: set of processes
while true **do**
    for $v \in vertexBuffer$ **do**
        if $v.r \leq r \wedge \forall p_u \in v.edges$: $DAG[v.r-1][p_u] \neq \bot$ **then**
            $DAG[v.r][v.p] \leftarrow v$; $vertexBuffer$.remove(v)
    if $|DAG[r]| < \lfloor \frac{n}{2} \rfloor + 1 \vee DAG[r][p_i] = \bot$ **then continue**
    if $r \mod 4 = 0$ **then** waveReady($\frac{r}{4}$)
    $r \leftarrow r + 1$
    wait until $\neg messsageBuffer$.isEmpty()
    $v \leftarrow$ new vertex; $v.block \leftarrow messageBuffer$.pop(); $v.r \leftarrow r$; $v.p \leftarrow p_i$
    for all $u \in DAG[r-1]$ **do** $v.edges$.add($u.p$)
    $\sigma = enclave$.sign(v)
    for all $p_j \in P$ **do send** $\langle \text{VERTEX}, v, \sigma \rangle$ to p_j
function a_broadcast(m)
    $messageBuffer$.insert(m)
function waveReady(w)
    $coinStore[w] \leftarrow enclave$.computeCoin($DAG[r]$)
    $v_w \leftarrow DAG[\text{round}(w,1)][coinStore[w]]$
    if $v_w = \bot \vee |\{u \mid u \in DAG[\text{round}(w,4)] : \text{path}(u, v_w)\}| < \lfloor \frac{n}{2} \rfloor + 1$ **then**
        return
    $committedRoots \leftarrow$ new stack; $committedRoots$.push(v_w)
    for $w' \leftarrow w - 1$ down to $decidedWave + 1$ **do**
        $v_{w'} \leftarrow DAG[\text{round}(w',1)][coinStore[w']]$
        if $v_{w'} \neq \bot \wedge \text{path}(v_w, v_{w'})$ **then**
            $committedRoots$.push(u); $v_w \leftarrow v_{w'}$
    while $\neg committedRoots$.isEmpty() **do**
        $v_w \leftarrow committedRoots$.pop()
        $verticesToDeliver \leftarrow \{u \mid u \in \cup_{r'>0} DAG[r'] : \text{path}(v_w, u) \wedge \neg u.delivered\}$
        for $u \in verticesToDeliver$ in deterministic order **do**
            $u.delivered \leftarrow$ True
            a_deliver $(u.p, u.r, u.block)$
    $decidedWave \leftarrow w$; $coinStore$.clear()

Algorithm 7 TEE-Rider Atomic Broadcast for Process p_i: Broadcast Logic and Utility

```
1: state PK: array of public keys
2: state requested: set of tuples (ℕ, ℕ), initialized empty
3: state receivedRequests: map of key (ℕ, ℕ) to list of processes, initialized empty
4: upon VERTEX(p_k, v, σ)
5:     if |v.edges| < ⌊n/2⌋ + 1 ∨ v.p ∉ v.edges then abort
6:     if DAG[v.r][v.p] ≠ ⊥ ∨ vertexBuffer.knows((v.r, v.p)) then abort
7:     if ¬vfy(PK[v.p], (v.r, v), σ) then abort
8:     v.σ ← σ; v.delivered ← False
9:     for all p_j ∈ receivedRequests.remove((v.r, v.p)) do
10:        send ⟨VERTEX, v, v.σ⟩ to p_j
11:    for all p_u ∈ v.edges do
12:        if DAG[v.r − 1][p_u] = ⊥ ∧ ¬vertexBuffer.knows((v.r − 1, p_u)) then
13:            if (v.r, p_u) ∉ requested then
14:                for all p_j ∈ P do
15:                    send ⟨V_REQUEST, v.r, p_u⟩ to p_j
16:                requested.add((v.r, p_u))
17:    vertexBuffer.add(v)
18: upon V_REQUEST(p_k, r′, p_v)
19:    v ← ⊥
20:    if DAG[r′][p_v] ≠ ⊥ then v ← DAG[r′][p_v]
21:    if v = ⊥ ∧ vertexBuffer.knows((r′, p_v)) then v ← vertexBuffer.get((r′, p_v))
22:    if v ≠ ⊥ then send ⟨VERTEX, v, v.c, v.σ⟩ to p_k
23:    else receivedRequests[(r′, p_v)].add(p_k)
24: function PATH(v, u)
25:    return exists a sequence of vertices (v_1, v_2, ..., v_k) ∈ ∪_{r′≥0} DAG[r′] such that
26:    v_1 = v ∧ v_k = u ∧ ∀i ∈ [2, k] : v_i ∈ v_{i−1}.edges
27: function ROUND(w, i)
28:    if i < 1 ∨ i > 4 ∨ w < 1 then abort
29:    return 4(w − 1) + i
```

takes a set of vertices *vs* as input. It first checks if the vertices are valid, i.e., if they are for the next toss round and if the signatures are valid (l. 12). Moreover, we ensure that the potentially Byzantine host system does not propose the same vertex multiple times (enforced by the *seen* set.). If at least $\lfloor \frac{n}{2} \rfloor + 1$ many valid vertices are provided (l. 16), the next toss round is incremented by 4, i.e., to the end of the next wave (l. 17), and a random number in the range of $[0, n] \subset \mathbb{N}$ is returned by invoking the CPRNG (l. 18).

4.4.7 Concluding Remarks

The algorithm as presented in this section proposes a single message per vertex. To achieve a scalable system, practical deployments require that a protocol step orders a batch of requests [Sin+08; Dan+22; Gir+24]. In the case of TEE-Rider, this can be achieved by

Algorithm 8 TEE-Rider Atomic Broadcast for Process p_i: Enclave (in TEE)

```
1:  state (sk, pk): asymmetric key pair
2:  state r: ℕ, initialized with 1
3:  state PK: array of public keys
4:  state nextTossRound: ℕ, initialized with 4
5:  state rng: CPRNG, initialized with a common seed
6:  function sign(m)
7:      σ ← sgn(sk, (r, m))
8:      r ← r + 1
9:      return (r − 1, σ)
10: function computeCoin(vs)
11:     valid ← 0; seen ← empty set
12:     for all v ∈ vs do
13:         if v.p ∉ seen ∧ v.r = nextTossRound ∧ vfy(PK[v.p], (c, v), σ) then
14:             seen.add(v.p)
15:             valid ← valid + 1
16:     if valid < ⌊n/2⌋ + 1 then return ⊥
17:     nextTossRound ← nextTossRound + 4
18:     return rng.uniform(0, n)
```

adding more than one message as a vertex' payload. As batching adds complexity to the protocol description and only affects the performance but not the correctness, we did not include batching in the protocol description.

For TEE-Rider to work properly, we can identify the following requirements:

- **Setup:** The enclave and the host system must be able to verify the validity of the vertices. In fact, it has to be ensured that every correct process works with the same set of public keys. In particular, the corresponding public keys must be managed inside an enclave that is trusted by all parties which is verified using attestation. For the common coin to work properly, it has to be ensured that every enclave initializes the CPRNG with the same seed.
- **Garbage collection:** Since TEE-Rider has an endlessly growing state, practical deployments require garbage collection to manage memory consumption.
- **Crash recovery:** Every process will eventually crash which will also crash the enclave. Without a recovery procedure, the system will eventually come to an halt since the number of crashed processes f becomes greater than t. When the enclave crashes, however, it must be impossible to rollback the enclave state. Otherwise, the enclave could sign the same vertex multiple times, which would violate the USIG property. Thus, USIG-based agreement protocols require procedures to reinitialize the USIG of a crashed process.

In the next two sections, we first discuss fundamental issues that arise from the requirements above and the combination of asynchrony and TEEs. Then, we present NxBFT, a

resilient and practical state machine replication protocol that addresses these issues in a pragmatic way.

4.5 Fundamental Issues with Asynchrony and TEEs

As pointed out before, a practical protocol following the SMR approach must ensure seamless operation – not only in the presence of faults. For this, the protocol must have support functions that go beyond the core of achieving some form of consensus and answering client requests. These functions increase resilience and usability in a practical deployment and must include mechanisms to set up the protocol, i.e., to initialize the system and to add new processes, to constrain the memory consumption with garbage collection, and to recover from faults, e.g., by restarting crashed processes. When a TEE is used to prevent equivocation, the TEE is trivially safety-critical and wrong usage can lead to safety violations. We find that, while the TEE alone makes this support functions harder to implement, the combination with asynchrony makes it even harder. In the following, we show that

1. the setup of a TEE-based agreement primitive requires consensus on the enclave identities and, thus, is impossible to achieve in partial synchrony with a fault tolerance of $n < 3t$ when assuming Byzantine faults,
2. the backfilling of the USIG-based causal order broadcast forbids garbage collection when operating in asynchrony and having no additional supporting protocols, and
3. USIG reinitialization, i.e., the establishment of a new enclave identity for a sending process, in partially synchronous, TEE-based reliable broadcast requires active participation of *all* processes.

The impossibility of crash recovery was previously published by Leinweber and Hartenstein [LH25]. The requirement for consensus during setup was also discussed by Marius Haller [Hal25].

4.5.1 Setup of a USIG-Based Peer-to-Peer Network

In TEE-Rider, every process is equipped with a USIG-like TEE that ensures that processes cannot equivocate. As pointed out above, in such a peer-to-peer network, it must be ensured that every correct process communicates with the same set of enclave identities. If correct processes would not use the same enclave identity for the same peering process p_i, p_i could start multiple enclaves, each with its own identity and counter value. A malicious process p_i could then use these different identities to send conflicting messages to different peering processes. Hence, p_i would have the ability to equivocate.

Moreover, it is important that the signature keys used by the USIG are actually bound to the enclave identity to prevent so-called simulation attacks in which an attacker can pretend to execute code inside TEE when it is not the case. Assume we have a set of n

processes $P := \{p_1, \ldots, p_n\}$ and a PKI assumption that allows the processes to identify each other. This PKI does not allow to distinguish between correct and faulty processes and it does not allow to verify enclave identities. Assume further that during setup, each process p_i will start its enclave that generates an asymmetric key pair. If now p_i would simply broadcast its public key, others would have no means to ensure that this public key stems from an enclave. If p_i is malicious, it could use a key pair that is not managed by an enclave and, thus, sign messages with arbitrary counter values which, again, is the ability to equivocate. Hence, the attestation must be used to bind the public key to the currently executed enclave.

In summary, we need to solve the following problem: n processes start their enclave which generates an asymmetric key pair and produces an attestation certificate that binds the public key to the enclave identity. Then, every process must provide its attestation certificate to the other processes and all correct processes must agree on the same set of enclave identities. Put differently, n processes propose an input value (their attestation certificate) and all correct processes must agree on the same output value (the set of attested enclave identities). This is a consensus problem (see Section 2.6.1 and [LLR02, Section 5, first paragraph]). In Section 2.6 and Table 2.2, we show that when assuming either partial synchrony or asynchrony, a PKI assumption, i.e., the authenticated Byzantine fault model, is not sufficient to reach consensus with a fault tolerance of $n > 2t$. Thus, if the setup phase should be able to tolerate the same amount of faults as the operation phase, we require either synchrony, the assumption of benign faults during setup, or a trusted third party that distributes the enclave identities to the processes. Another option is to abort in the case of a fault instead of tolerating it (see Section 4.6.4). Note that the distributed key generation required to establish common (for the naive TEE-based common coin) or shared (for the TEE-based Cachin coin) secrets requires consensus as well.

4.5.2 Backfilling vs. Garbage Collection

Narwhal proposes the following simple garbage collection scheme for asynchronous, DAG-based atomic broadcast protocols [Dan+22, Section 3.3]:

1. There is a predefined garbage collection offset g used by correct processes.
2. When a correct process updates its *decidedWave* variable to w, it garbage collects all rounds $r < \text{round}(w - g, 1)$.
3. When a correct process p_i garbage collects a vertex v that was created by p_i itself, it checks if v was delivered. If v was not delivered, p_i proposes v's payload with the next vertex it creates again ("re-injection" in the original Narwhal paper).

This garbage collection scheme piggybacks the ongoing consensus logic and does not require any additional communication or coordination. The re-injection of undelivered payloads ensures that AB-Validity is not violated. As discussed before, weak edges are in conflict with such a garbage collection scheme: a process may receive a vertex with a weak edge to a very old round. If the process garbage collected this round already, it

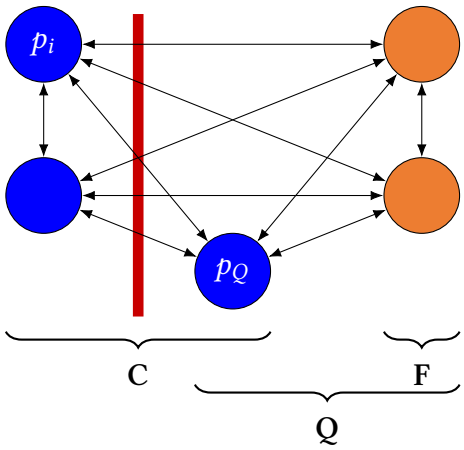


Figure 4.7: Network setup for the backfilling counter example with $n = 5$. Blue processes on the left form the set of correct processes $|C| = t + 1$, orange processes on the right form the set of faulty processes $|F| = t$. There is a temporary network partition between two correct processes, one of which is p_i, and the rest of the network. The network partition enforces that only p_Q and the two faulty processes together can for a quorum $|Q| \geq \lfloor \frac{n}{2} \rfloor + 1$ and make any progress.

cannot accept the vertex. Safety will not be at risk but the consensus logic at said process will eventually stall. While we safely removed weak edges, we introduced backfilling to significantly improve the communication complexity. Backfilling is necessary: without backfilling, the processes would "drown" in messages (see Figure 4.2) that, in most cases, are not required to make progress. In the following, however, we will show that backfilling and garbage collection are in conflict with each other when operating in asynchrony. In Section 4.6.5, we make us of the fact that we use the atomic broadcast for SMR and present a garbage collection scheme that is compatible with backfilling and does not require secondary storage to ensure liveness.

Let $n \geq 2t + 1$ and, thus, the quorum size be $\lfloor \frac{n}{2} \rfloor + 1 > t$. The actual number of faulty processes is $f = t$. Thus, we can divide the set of processes P into the set of correct processes $C, |C| = t + 1$ and the set of faulty processes $F, |F| = t$. Further, we assume that all but one correct process $p_q \in C$ observe a network partition denying any progress. Now assume that wave w is the last wave a correct process $p_i \in C$ committed. Further assume that the correct process $p_Q \in C$, which does not suffer from the network partition, commits a wave w' with $w' - (g + 1) > w$. As p_Q is correct, it observed vertices of at least $\lfloor \frac{n}{2} \rfloor + 1$ processes (in the following grouped in set $Q, |Q| = t + 1$) in round$(w', 4)$ and p_Q garbage collected all rounds $r' < r_g := \text{round}(w' - (g + 1), 1)$. By assumption, all processes in Q except p_Q are also in F: $Q_f = Q \cap F, |Q_f| = t$ and $Q_c = Q \cap C = \{p_i\}, |Q_c| = 1$. The network configuration is shown for $n = 5$ in Figure 4.7. Thus, p_i is only guaranteed to eventually receive the vertices of rounds $r' > \text{round}(w, 4)$ that p_Q created as all other processes are faulty. If now p_i receives a vertex v created by p_Q for round $r_v, r' \leq r_v < r_g$ that has an edge to a vertex it did not receive, it cannot accept v (yet). The reason is that v is for a round that lies in the range where p_Q already garbage collected ($< r_g$) and p_i could not transition to yet ($\geq r'$). As all correct processes except p_Q lag behind, v must have been created by a process in F. Process p_i will send a V_REQUEST message to all processes.

There is no guarantee that any server in F will answer and the only server in C that could answer is p_Q. As p_Q is correct, it applied the garbage collection scheme and, thus, cannot answer the request once the network partition is lifted. Thus, progress at p_i will stall.

It follows that garbage collection breaks liveness when combined with backfilling in asynchronous (and also partially synchronous) networks. Narwhal implicitly acknowledges this observation by stating that garbage collection moves vertices to cheap and slow storage [Dan+22, Section 3.3, last paragraph] even though they state to have made a significant improvement concerning garbage collection in comparison to DAG-Rider [Dan+22, Section 8.2]. Under the assumption that network partitions as described above do not happen frequently, this slow storage must not be accessed a lot. However, graph data must not be deleted from the slow storage to ensure liveness and, once a network partition occurs, the slow storage may become a bottleneck and hinder progress.

4.5.3 Crash Recovery: USIG Reinitialization

Faults like hardware failures, human configuration errors, and maintenance require that a process can be put back into the condition to validate received messages and produce valid messages. We are interested in an algorithmic solution that can recover processes while allowing the system to continuously operate. For TEE-Rider, such a recovery procedure must ensure that the recovery does not allow (1) equivocation and (2) to learn coin values beforehand. Hence, a recovery procedure must be aware of the enclave.

The enclave must not be able to be started in a rolled back state. This is a safety requirement as the enclave must not be able to sign different vertices with the same counter value. However, there is no practical solution out there that allows that every state transition is persisted in a way that it can be recovered from and previous states are inaccessible. Instead, TEE technology allows to export data from the enclave to the host system with confidentiality, integrity, and authenticity being preserved. Freshness, however, is not guaranteed. Hence, the enclave logic must not support to export or import safety critical state. In case of TEE-Rider, this means that the enclave must neither import nor export the private key. For the common coin, rollbacks are not an issue: A rollback does not allow to learn coin values beforehand; instead, it allows coin values to be tossed a second time.

If now an enclave crashes, it must be restarted in a fresh state with a new asymmetric key pair and, thus, a new identity as the corresponding public key is bound to the enclave identity during attestation (see Section 4.5.1 above). Consequently, a recovery procedure has to safely ensure that all correct processes agree on the new public key of the restarted enclave. In TEE-Rider and similar approaches, the only *interactive* coordination mechanism are the n asynchronous, TEE-based reliable broadcast instances. Following the argumentation in this chapter, all decisions are safely and passively derived from the reliable broadcast message history, i.e., the DAG. Thus, recovering a TEE-Rider process is recovering the reliable broadcast state in a way that the reliable broadcast properties RB-Agreement, RB-Validity, and RB-Integrity are preserved for *all* correct processes. We show that to

preserve the reliable broadcast properties, a recovery procedure must honor the input of all correct processes making it impossible to work with a quorum size smaller than n.

Assumptions Let P be a set of processes that implement a reliable broadcast primitive for a single sender $p_s \in P$ as in Definition 2.2. Of the $n := |P|$ processes, $t < n$ processes may be Byzantine faulty. The processes communicate over partially synchronous, perfect point-to-point links. The sender p_s is equipped with a USIG (see Algorithm 1) hosted in a TEE (Definition 2.1). The USIG may only fail by crashing and looses its private key when it crashes. **USIG reinitialization** is the process when a sender p_s aims to change its USIG identity. When p_s (re-)initializes the USIG, the USIG creates a new asymmetric key pair, i.e., a new identity.

Lemma 4.8 (Multiple valid USIG instances allow equivocation)**.** *When USIG reinitialization allows a faulty sender in USIG-based reliable broadcast to exchange its USIG identity only for a subset of correct receiving processes, this allows equivocation and RB-Agreement is violated.*

Proof. Assume that the reliable broadcast protocol was properly set up and that the sender p_s is the only faulty process. The set $C \subset P \setminus \{p_s\}, |C| = n - 1$ is the subset of correct processes. Now, p_s starts a new USIG instance with a new public key; p_s initializes attestation with the processes of a proper subset $C_A \subset C$ of correct processes. As the attestation is valid, the processes in C_A accept the new public key and start to use it. From this point on, the processes in C_A reject messages signed by the old USIG identity. Receiving processes in $C_B := C \setminus C_A$ still accept messages signed by the old USIG identity. RB-Agreement requires that all correct processes C deliver the same message for a given tag c. In USIG-based reliable broadcast, the tag is the USIG counter value. As p_s can now sign messages with both USIGs, p_s can commit an equivocation fault and can send two different messages for the same tag c to the processes in C_A and C_B. As USIG-based reliable broadcast omits measures to prevent equivocation (besides the USIG), the processes in C_A and C_B will deliver different messages for the same tag c. This violates RB-Agreement as the processes in C_A and C_B do not agree on the message delivered for tag c. □

Remark. Forcing the sender to reliably broadcast the new USIG identity or the attestation to all correct processes does not suffice to solve the problem as it does only ensure that eventually all correct processes will exchange the USIG identity of the same identity. In particular, it does not enforce this change to happen simultaneously.

Consequently, the recovery procedure must ensure that all correct processes exchange the public key of the sender's USIG at the same point in logical time, i.e., the change must become effective at the same round. It follows that it must be possible to logically assign tags to the time before and after the change which can be achieved by enforcing an ordering on the tags (e.g., as done in FIFO broadcast presented in Algorithms 2 and 3) or by changing the tag to be a tuple (e, c) where e is a monotonic epoch identifier that is incremented when the USIG identity is changed. Such agreement can be achieved starting a separate consensus protocol or by piggybacking the running atomic broadcast algorithm:

Processes do not only have their enclave to sign messages but also a second key pair that can be used to identify the process itself. This second key pair is managed outside the enclave and it can be saved in backups. After the crash, the now recovering process sends an administrative request, i.e., a special message, to all processes in which it requests to update its enclave identity. This request is signed with the second key pair and ordered using the running consensus algorithm. Once the consensus algorithm decided on the request, correct processes will exchange the public key of the crashed process' enclave. As long as not more than t processes crash, the new public key will be known to all correct processes. For TEE-Rider and similar approaches however, this procedure is not safe without additional measures.

Theorem 4.5 (USIG reinitialization in partial synchrony requires n-sized quorums)**.** *Assume that a reinitialization procedure achieves consensus on the counter value up to which a previous USIG public key is accepted and after which the new public key has to be used. If this consensus does not ensure that the inputs of all correct processes are honored in the decision, i.e., the quorum size required to achieve consensus is smaller than n, RB-Agreement is violated.*

Proof. Let the sender p_s be the only faulty process and let $C \subset P \setminus \{p_s\}, |C| = n - 1$ be the set of receiving processes. Moreover, let $p_i \in C$ be a correct process that is not part of a set $C_A \subset C \setminus \{p_i\}$ of correct processes. Assume that a partially synchronous, fault-tolerant consensus protocol $\mathcal{D}$ is given in which each process can vote on the new USIG identity and the highest counter value c' for which the old USIG public key is accepted. Output of the consensus protocol is an attestation with a public key and a counter value c' from which on the new identity is to be used. The reinitialization protocol proceeds as follows:

1. When receiving the administrative request, a correct process $p_j \in C$ checks if the attestation is valid. If the attestation is invalid, p_j aborts.
2. Process p_j stops accepting messages signed by the old USIG identity.
3. Process p_j inputs the received attestation and all messages it was able to deliver – together with their counter value and their signature – into the consensus protocol $\mathcal{D}$.
4. The consensus protocol $\mathcal{D}$ identifies the maximum counter c' value for which a valid signature was input by a process. Once the consensus protocol $\mathcal{D}$ decides on the attestation and counter value c', process p_j accepts the new USIG identity and starts to accept messages from the sender again.
5. Process p_j requests messages for counter values $c \leq c'$ which it was not able to deliver from the processes in P to ensure RB-Agreement for these counter values.

In Section 2.6.3 and Table 2.2, we discuss that atomic broadcast, which is equivalent to consensus (see Section 2.6.1), cannot be solved in partial synchrony with a fault tolerance of $t \geq \frac{n}{2}$. Consequently, the consensus protocol $\mathcal{D}$ must work with a quorum to be able to tolerate faults. Assume that, without loss of generality, the quorum size is $n - 1$. Hence, $\mathcal{D}$

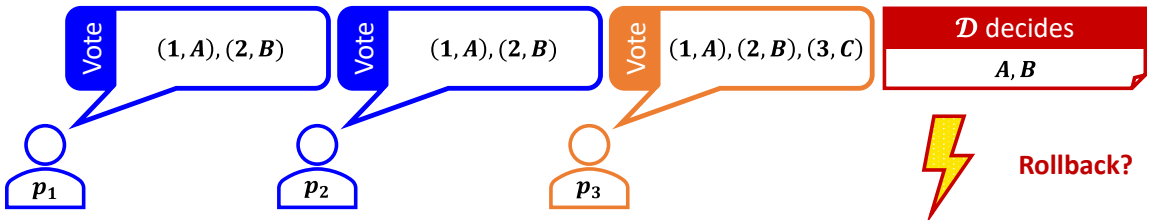


Figure 4.8: Illustration of the rollback problem in USIG reinitialization for $n = 3$ processes; all processes are correct. The sending process is not illustrated. A quorum of size $n - 1$ build with processes p_1 and p_2 may outvote process p_3. This can happen due to the partially synchronous communication links. As p_3 already delivered the message for tag $c = 3$, it is forced to rollback to prevent that RB-Agreement is violated.

must proceed as soon as it received $n-1$ votes. By the assumption of partially synchronous links, it is possible that the processes in C_A form the quorum being used to decide and p_i's vote is not considered. Assume now that the maximum counter value p_i delivered a message for is c. USIG-based reliable broadcast can deliver messages without coordination (it does not require a ready phase, see Section 4.2.1). Although p_i is correct and echos the message for c, the partially synchronous link can delay the delivery of the echo message such that p_i is the only process who delivered the message for counter value c before p_s crashed. Thus, $\mathcal{D}$ must output a counter value $c' < c$. It follows that processes in C_A will at least reject the message for counter value c which p_i delivered. Hence, either p_i performs a rollback undoing the delivery or the processes in C_A deliver different messages for counter value c. Both variants break RB-Agreement. □

Figure 4.8 illustrates the problem for $n = 3$ processes. With Theorem 4.5, the following corollary follows trivially:

Corollary 4.1. *USIG reinitialization in TEE-based reliable broadcast protocols requires a quorum size of n to maintain RB-Agreement when operating in asynchronous networks.*

Remark. The impossibility result stems from the fact that USIG-based reliable broadcast protocols work without quorums. If the broadcast protocol would establish a quorum of at least $\lfloor \frac{n}{2} \rfloor + 1$ processes and the reinitialization protocol would use a quorum size of $n > 3t$ to prevent that Byzantine processes can outvote correct processes in quorums, the reinitialization protocol could ensure that c' is set to the maximum counter value c that was delivered by any correct process. This, however, would nearly completely nullify the benefit of using the USIG as it would require to follow the well-known Bracha protocol scheme with three phases (see Section 4.2.1).

If the reliable broadcast protocol is used as a building block in higher-level protocols, e.g., causal order reliable broadcast, atomic broadcast or state machine replication, as it is the case for TEE-Rider, the violation of RB-Agreement obviously *may* break safety properties as well. On the one hand, if the RB-Agreement property can be changed such that rollbacks, i.e., the "undoing" of `r_deliver` decisions, are allowed, the presented impossibility results do not hold. However, if this would also cause a rollback in the higher-level protocol,

Lewis-Pye and Roughgarden [LR25] show that the number of rollbacks is unbounded if no additional synchrony assumptions are made. On the other hand, additional agreement primitives may be used to ensure that the properties of the higher-level protocols are preserved while, at the same time, the reliable broadcast properties are violated. We will propose such an approach to enable reconfiguration in NxBFT in Section 4.6.5.

4.6 NxBFT: Resilient and Practical State Machine Replication

Until now, we discussed reliable broadcast and atomic broadcast protocols as a crucial fundament for state machine replication (SMR) protocols. With TEE-Rider, we presented an optimized atomic broadcast protocol that is, from a theoretical point of view, an efficient and resilient solution for atomic broadcast. As this dissertation aims at the investigation of practical SMR protocols, the question arises how TEE-Rider can be used to implement a practical SMR protocol. The discussion of the leader bottleneck in Section 4.1.1 and the impossibility results above show that simply adding an SMR client logic to TEE-Rider is not sufficient to implement a practical and efficient SMR protocol. We propose NxBFT, a practical SMR framework that is based on TEE-Rider and addresses the leader bottleneck, setup, garbage collection, and recovery issues.

We first define the system model of NxBFT and then present the NxBFT SMR framework in four parts:

1. We discuss the requirement for voting in BFT SMR protocols and its scaling-limiting implication for leaderless protocols. To minimize redundancies in the TEE-Rider DAG and improve scalability, we propose the "Not eXactly Byzantine" (NxB) client model, an adaption of the hybrid fault model. Moreover, we present and briefly discuss alternatives to the fault model adaption.
2. We describe the NxBFT SMR framework logic for normal case operation (i.e., without setup, garbage collection, and reconfiguration) at the example of a full Request–Consensus–Reply cycle. NxBFT deploys measures to improve performance by request batching while at the same time ensuring progress in low-load scenarios.
3. We propose an algorithmic setup procedure that allows to set up a peer-to-peer network of enclaves for NxBFT. We argue that in practical deployment faults should not be simply tolerated, i.e., ignored. Instead, the proposed protocol detects faulty behavior and reliably aborts in the case of faults thereby circumventing the impossibility result from Section 4.5.1. In a fault-free setup, the protocol achieves consensus on the enclave identity of each of the replicas ensuring the safe operation of NxBFT.
4. We propose a protocol that regularly establishes agreement on the current application state (a checkpoint). In a state transfer protocol, checkpoints can be safely transmitted between replicas. We find that a checkpoint-based state transfer protocol can be used to circumvent the garbage collection impossibility result (see Section 4.5.2) and the recovery impossibility result (see Section 4.5.3).

The NxB model, the NxBFT framework, and the setup procedure were previously published by Leinweber and Hartenstein [LH25]. The same publication contains an alternative recovery procedure similar to the one described in the proof of Theorem 4.5. This recovery procedure requires the participation of all replicas to circumvent the impossibility result for quorum-based recovery protocols. Recovery based on reconfiguration, as presented in this dissertation, is a novel contribution improving the resilience of the recovery procedure.

4.6.1 System Model

We consider a federation of n predefined operators o_i offering a common service for an arbitrary number of clients. Operators can be authenticated using a public key infrastructure (PKI). Each operator operates a replica $p_i \in P := \{p_1, \ldots, p_n\}, |P| = n$. The replicas are connected via an asynchronous network; replicas and clients communicate via secure, perfect point-to-point links (see Sections 2.2 and 2.4). Thus, messages can be reordered and arbitrarily delayed but not dropped. For liveness of garbage collection, state transfer, and reconfiguration, we assume partial synchrony in the GST model (see Section 2.4). Each replica is equipped with a trusted execution environment (TEE, Definition 2.1). Of the n replicas, at most $t < \frac{n}{2}$ replicas can be Byzantine faulty (see Section 2.3). The TEE is assumed to only fail by crashing; its internal state is lost when the TEE crashes. The number of Byzantine clients is not limited.

4.6.2 Not eXactly Byzantine: Unlock TEE-Rider's Scaling Capabilities

Probably the most important difference between atomic broadcast and SMR is that the latter promises guarantees towards an possibly unbounded number of clients of which all may behave Byzantine. In SMR, a client must be aware of the fact that it interacts with a distributed system to maintain the safety and liveness guarantees towards the client. In this subsection, we discuss the implications of this requirement and we propose the "Not eXactly Byzantine" (NxB) operating model allowing NxBFT to scale to a large number of clients, requests, and replicas.

Problem Statement: On the Requirement for Voting in BFT SMR

The SMR system can be split into two groups: the group of clients and the group of n replicas. Tolerating a Byzantine client is not more complex than in the centralized setting with a single server: The server-side logic must ensure that the client cannot influence the server's state in a malicious way. This is typically achieved by ensuring the client's request's authenticity and integrity using signatures as well as by a stateful application logic that ensures that the client cannot influence the server's state in a way that violates the server's safety properties.

The client, however, cannot distinguish between correct and Byzantine faulty replicas (as correct replicas cannot as well). If the client talks to a single replica, this replica may be

faulty and, by a simple omission fault, the client may never receive a response. Moreover, the client must ensure that, before accepting a response, it received a correct response and that the state transition leading to the response will eventually be persisted at all correct replicas. Consequently, BFT SMR clients must perform a *voting* procedure ensuring that a consistent response is received from a quorum of replicas [Li+18; BRB21; HHM24]. This step cannot be omitted in the hybrid fault model; a client must receive a consistent reply from at least $t + 1$ replicas. Typically, the client communicates with all n replicas (see, e.g., [Ver+11, Figure 1]).

TEE-Rider, as the underlying atomic broadcast protocol of NxBFT, comes with a significant performance improvement over leader-based coordination protocols: Every process can independently propose messages. If we now naively add well-known BFT SMR client logic, at least $t + 1$ replicas will receive the same client request. Because of the leaderless nature, there is no designated replica that tasked with proposing the request. Consequently, all $t+1$ replicas will add the request to their next TEE-Rider vertex. Hence, the TEE-Rider DAG contains a significant amount of redundant request data. Moreover, the voting requires communication which itself causes additional overhead on client and replica side. This issue is not unique to NxBFT and a common issue for DAG-based protocols [WKM24, Section 3.4]. As a result, the performance results reported in the literature (e.g., [Aru+25; Ton+25]) do *not directly* translate to the BFT SMR use case of atomic broadcast. To the best of our knowledge, there is no clear statement in the literature that addresses the non-negligible impact of BFT SMR clients on the performance of DAG-based protocols. Similar challenges have been observed in non-DAG protocols as well [Gol+19; Sta+22]. In conclusion, it is disadvantageous to use a consensus-agnostic SMR framework. To maximize the benefits of using DAG-based atomic broadcast, it is required to pursue a co-design of the client logic and the underlying protocol.

Approaches to Maximizing the Uniform Distribution of Clients Across Replicas

In the following, we discuss three alternative approaches, namely client agents, the NxB model, and threshold cryptography, to reduce the impact of SMR client logic on the performance of TEE-Rider and similar protocols. As this makes the presentation and the implementation a lot easier, we assume that correct clients are synchronous: A client blocks until it receives the required amount of consistent responses. Only then may a new request be submitted. Consequently, faulty clients breaking this assumption will loose the SMR guarantees for their requests.

Client Agents The first approach is to assign clients to a fixed set of replicas with size $t+1$. Each of these replicas is then a client agent that is responsible for handling the requests of the clients assigned to it. The client only contacts a replica not part of the agent set if the client does not receive $t + 1$ consistent responses (i.e., to preserve safety) or a not sufficient number of responses within a reasonable time span (i.e., to preserve liveness). This fallback behavior requires that each replica stores for each client the last request and response [Dis21; Bes+23]: If a client observes a timeout and contacts a replica not part

Figure 4.9: Illustration of the NxB operating model. Clients are potentially Byzantine faulty. Replicas are potentially Byzantine faulty when interacting with other replicas but limited to omission faults when interacting with clients.

of its agent set, said replica must be able to answer the request. However, it is possible that the replica already forwarded the request to the application layer and dropped the response because there was no active connection to the client. Because the replica cannot forward the request to the application layer a second time – this would trigger another state transition –, the replica must retain the response to be able to answer a client request after the request was already ordered by the atomic broadcast protocol. The client agent approach preserves Byzantine fault tolerance and allows limited scaling. In expectation, replicas share 50% of client requests limiting the improvement to a constant of factor 2 in the common case.

Not eXactly Byzantine: The NxB Operating Model Omission fault-tolerant SMR clients do not require to contact more than one replica to maintain safety. In the omission fault model, a replica will, at most, omit to forward the request to the atomic broadcast or the response to the client. If the client receives a response, it is guaranteed to be correct. If we could assume that replicas do not bribe their clients or that such misbehavior is guaranteed to be detected and prosecuted, we could adapt the fault model to allow omission faults when replicas interact with clients. This is the **Not eXactly Byzantine** (NxB) model (see Figure 4.9): the assumption of potentially Byzantine behavior from clients and between replicas but only omission faults when replicas interact with clients. Adapting this approach to TEE-Rider allows to have one client agent per client. In expectation, this would eliminate any shared client connections between replicas and improve the performance by a factor of $O(n)$ in the common case. There are two different situations in which the NxB assumption has no negative impact on security:

- **Fully enclaved execution:** The complete server side of the SMR framework, i.e., SMR server, atomic broadcast, and application server, is executed inside a TEE. Based on attestation, the client can ensure that it interacts with a correct replica. The client then has the guarantee that every response received has been correctly sorted and processed. The Confidential Consortium Framework (CCF) [How+23] follows this approach. Fully enclaved execution eliminates any Byzantine behavior on the replica side. However, such an approach significantly increases the trusted computing base, and the significantly increased number of context switches into and out of the TEE has a negative impact on performance [Li+18; WAK18].
- **Auditability:** The client can ensure that misbehavior of a replica can always be detected. While this misbehavior breaks SMR guarantees, the client can prove its correct behavior to a quorum of replicas or a third party (e.g., a court) to ensure that there will be no negative consequences for the client. An example for such a

scenario is the ticketing system as discussed in Chapter 3. The client can provide evidence of its correct behavior (i.e., its signed history of check-ins and check-outs and signed answers of replicas), thus protecting itself from potential repercussions.

Server-Side Voting Proxy If a client can safely task a replica to perform the voting step on the client's behalf, it suffices for the client to contact a single replica. As with the NxB model, this approach allows a performance improvement of factor $O(n)$ but it achieves full Byzantine fault tolerance. A recent line of research [Gol+19; HHM24] proposes to use threshold cryptography to allow a client to derive from a single response whether its request was safely ordered and handled. The simplified CART approach [HHM24] works as follows[6]: Clients contact a single replica, i.e., their client agent, with their request. The client agent forwards the request to the atomic broadcast protocol. After the request was ordered and executed, replicas sign their responses using a threshold signature scheme. The client agents collect the threshold signatures and, when having received $t + 1$ valid partial signatures, the client agents can combine them to a full signature. The full signature is sent to the client together with the response. However, this approach adds additional overhead to the protocol: Replicas have to exchange partial signatures whenever they delivered a request on the atomic broadcast and subsequently computed a response. Heß et al. [HHM24] show that such a protocol has a non-negligible negative impact on performance: The author's report between 10% and 20% reduced throughput and between 10% and 20% increased latency. An alternative is to execute the response aggregation inside a TEE as proposed by Troxy [Li+18]. The authors implement the Troxy approach for the Hybster hybrid fault-tolerant, multi-leader protocol [BDK17] and report a performance loss of up to 43%. We conclude that threshold cryptography as proposed by Heß et al. [HHM24] should be used if the NxB model cannot be justified, i.e., in the case of non-enclaved execution or when auditability cannot be guaranteed. As in our intended use cases the NxB model can be justified, we do not consider threshold cryptography in the following and leave the investigation of a suitable NxBFT protocol extension and the analysis of the performance impact to future work.

4.6.3 The NxBFT SMR Framework

NxBFT is a state machine replication framework that is based on TEE-Rider and the NxB model. NxBFT is structured in three layers (see Figure 4.10): SMR framework, causal order broadcast (i.e., dissemination), and atomic broadcast (i.e., consensus). The latter two are unchanged from TEE-Rider. The SMR framework implements the client logic and the SMR server that ensure, by deploying an atomic broadcast, the SMR properties. In the following, we describe the normal operation, i.e., without setup, garbage collection, and reconfiguration, with a full Request–Consensus–Response cycle.

[6] The authors also propose optimizations for batched processing of requests which we do not consider in the explanation.

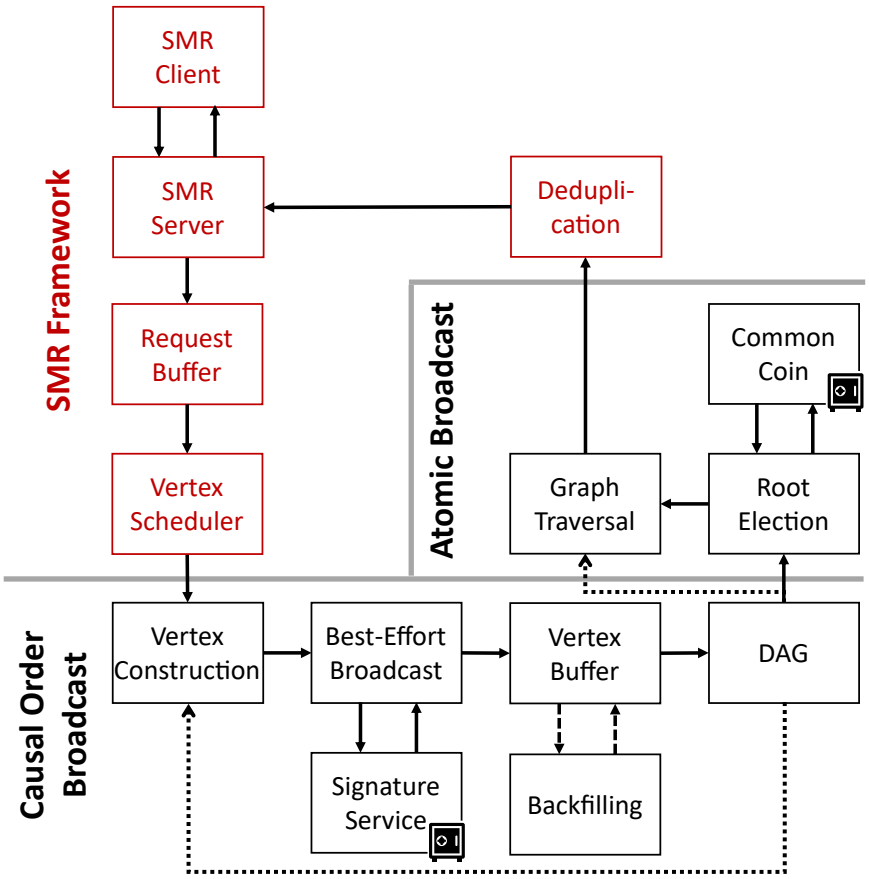


Figure 4.10: NxBFT SMR framework components for normal operation (setup, garbage collection, and reconfiguration left out). Red colored components are added in comparison to TEE-Rider. The safe icon marks components executed inside the TEE. Solid arrows show the flow for a client request to be successfully ordered and executed. After being received by the replica, a request is buffered and eventually broadcast as a vertex payload in TEE-Rider. A vertex is added to the DAG when its ancestry was already added (optional backfilling, dashed arrows). Each added vertex will eventually be ordered by a graph traversal; the corresponding root is selected using a common coin. After request deduplication, the request is output to the application layer and the response sent to the client. The dotted arrows highlight components using the DAG as input.

Request: NxB Client Model for Throughput Scaling When a client wants to use the federated service, it composes a request containing the command and a sequence number and sends the request to a single replica. Clients select the replica at random and unicast their request accompanied with a client id and a sequence number. When the client issues its request, it starts a timer. If the selected replica does not answer within the time interval, the client selects a different replica for its request (and so forth). A receiving replica buffers the request until it can be included in the payload of a broadcast round. If the request was already ordered and executed, the replica responds to the client immediately. As explained above, to be able to answer already ordered requests, the replica stores the last request-response pair for each client. A correct client will only have one unanswered request at a time. If a client issues more than one request simultaneously or equivocates

on sequence numbers, i.e., it is faulty, it is possible that some of its requests will not be answered.

Vertex Construction and Consensus Invocation: Pacing Pending requests are buffered at the replica until the replica is allowed to broadcast a new vertex. To increase the achievable throughput, a replica holds back its vertex until it can propose at least a configurable amount of client request as the vertex payload. To prevent denial-of-service attacks, additionally a maximum vertex payload size is configured. Without broadcast requests, TEE-Rider cannot make progress. A replica needs at least $\lfloor \frac{n}{2} \rfloor + 1$ valid vertices for its current round to transition to the next round and to be allowed to broadcast its next vertex. It is a common pattern in asynchronous atomic broadcast protocols that progress depends on a sufficient number of processes with not yet delivered A_BROADCAST(·) requests. If now only a few clients issue requests, the system would come to a halt as the quorum for a round to complete would need a lot of time to be established. The NxB client model worsens this problem as requests are equally distributed among the replicas. We address this issue by working with local timers on the side of the replica: When a replica is not able to broadcast a vertex containing enough client request within a pre-defined time interval, the replica will broadcast the vertex anyway. This allows to complete the round within a reasonable time span.

Response: At-Most-Once Semantics The requests are executed by the server-side application in the order they were added to the vertex by the proposing replica. After execution, the result is sent as a response to the initially requesting client. The client is guaranteed that its request will eventually be ordered, however, due to the timeout-based fallback logic to ensure liveness, a request can appear in the DAG up to n times. NxBFT employs a simple deduplication logic: The replicas store for each client the client's last executed sequence number. A request is only input to the state machine when its sequence number is greater than the stored sequence number.

Remark. Please note that NxBFT can also be operated with a classic, synchronous BFT SMR client or with the client agent model as described above that both contact in the common case at least $t + 1$ replicas. This allows the client to tolerate Byzantine behavior of replicas in any case but has a significant performance impact (see Section 5.5.1).

4.6.4 Algorithmic Enclave Network Setup

During setup, NxBFT enclaves mutually attest and verify their integrity, reach consensus on the deployed enclave code as well as the peers' public enclave keys to prevent simulation attacks (see Section 4.5.1), and establish the authenticated channels between replicas. The second objective of the setup protocol is to initialize the CPRNG underlying the common coin with a collaboratively generated random seed. We do not consider a fault-tolerant setup procedure to be reasonable: When a replica already shows a fault during setup, the actual fault tolerance of the operation phase would be reduced from the very beginning. We

argue that an implicit task of the setup phase is to ensure that all replicas are available and correctly configured (e.g., deployed code and parameters not defined during setup). Thus, the proposed setup protocol aborts in the case of faults. We require the participation of all n replicas and construct the setup protocol using n synchronous authenticated reliable broadcast instances with a fault tolerance of $n > f$ [PSL80] to preempt equivocation by faulty replicas. By requiring all replicas to participate and aborting in the case of a fault instead of tolerating the fault, we circumvent the impossibility result identified in Section 4.5.1.

We describe the setup protocol of NxBFT as a one-way handshake between replicas p_i and p_j. A replica p_i also performs this setup handshake with itself.

1. Upon initialization, the enclave of p_i initializes its state variables and generates both a new enclave key pair and a random seed share for the later initialization of the common coin PRNG.
2. Replica p_i now uses a synchronous authenticated single-echo reliable broadcast to disseminate its identifier and its signed attestation certificate carrying its public enclave key.
3. Replica p_j waits to receive n validly signed echo messages carrying consistent and valid attestation certificates.
4. Replica p_j then sends its encrypted seed share to p_i (encrypted using p_i's public enclave key), thereby completing its part of the handshake.
5. Replica p_i then invokes its enclave with the received encrypted seed share. The enclave decrypts the seed share and XOR's it with the current seed value.
6. Once a replica has completed all n handshakes, it has completed the setup protocol and broadcasts a ready message to all replicas.

The XOR construction is a variant of the straightforward $t = n$ secret sharing; the resulting seed is kept confidential by the secure communication and the TEE. Replicas start a timer for each handshake, the expiration of which raises an error and leads to an abortion of the entire setup. Similarly, conflicting or invalid messages and certificates raise errors and lead to an abort.

4.6.5 Checkpoint-Based Garbage Collection and Reconfiguration

Garbage collection typically relies on the checkpoint concept [CL02; Dis21]. When a replica processed a certain amount of requests, it creates a snapshot of its state and deletes all information that is older than the snapshot. Partially synchronous and asynchronous links, however, complicate the procedure: Assume $n > 2t$ and a replica p_c reached a checkpoint. It may now be the case that p_c is the only *correct* replica that reached the checkpoint. When now other correct replicas, due to network delays, fell behind and, due to Byzantine faulty replicas that contributed to the checkpoint but do not send all their messages to all replicas, they may not be able to catch up with the correct state. In the following, we call

such replicas **stale**. To allow stale replicas to catch up, other correct replicas that are "up to date", in our scenario only replica p_c, must be able to provide the required information that allows stale replicas to safely update their state. Such information can either be the full message history of the atomic broadcast protocol or a quorum of snapshots. If replica p_c would delete the state information without coordinating with other replicas, it cannot prove to stale replicas that the snapshot it created is the result of the consecutive execution of all request ordered by the atomic broadcast protocol. In such a case, the correct replica p_c would have applied requests that other correct replicas cannot apply thus breaking SMR safety.

As discussed in Section 4.1.3, the combination of a checkpoint protocol and a state transfer protocol is the basis for garbage collection that does neither break SMR safety nor SMR liveness. A **checkpoint protocol** establishes so-called **stable checkpoints**. A checkpoint is stable when a replica received a quorum of consistent votes that confirm the validity of the checkpoint. When a replica stabilizes a checkpoint, it can safely delete all information that is older than the stable checkpoint but it keeps the received signed votes. In a **state transfer protocol**, the state of a stable checkpoint is transferred to a stale replica [CL02; Dis21]. The state transfer allows the stale replica to safely update its local state to the checkpoint state such that it (1) can validate messages received from clients and peering replicas and (2) that it can send messages that other replicas will accept. Moreover, state transfer can be used to implement a reconfiguration protocol in which replicas can be added, removed, and replaced [Dis21, Section 8].

In this subsection, we describe how checkpoint and state transfer protocols can be implemented as the basis for garbage collection and reconfiguration in NxBFT such that the impossibility results of Sections 4.5.2 and 4.5.3 are circumvented. In the following, we will first derive how the state that is agreed upon in the checkpoint protocol and transferred in a state transfer must be structured to ensure a safe and live state transfer. Subsequently, we use the checkpoint protocol to propose garbage collection and reconfiguration protocols for NxBFT. Finally, we analyze the proposed protocols and discuss their relation to the impossibility results from Section 4.5.

State Transfer Requirements

According to Distler [Dis21, Section 7], the required checkpoint data can be split into three parts: application state, replies, and protocol state. When the checkpoint is created, a correct replica must create an atomic snapshot of all three parts that is then immutably stored until it may be garbage collected. Following Distler's classification, application state and replies are consensus-agnostic and, thus, do not differ for NxBFT in comparison to other BFT SMR protocols. The application state is the state of the replicated business logic that is executed by the SMR framework. The set of replies contains for *each* client the last request (including the request sequence number) and the corresponding response. The set of replies ensures that the receiving replica can maintain the at-most-once semantics of the SMR framework and that a receiving replica can answer client requests that were already answered by other replicas.

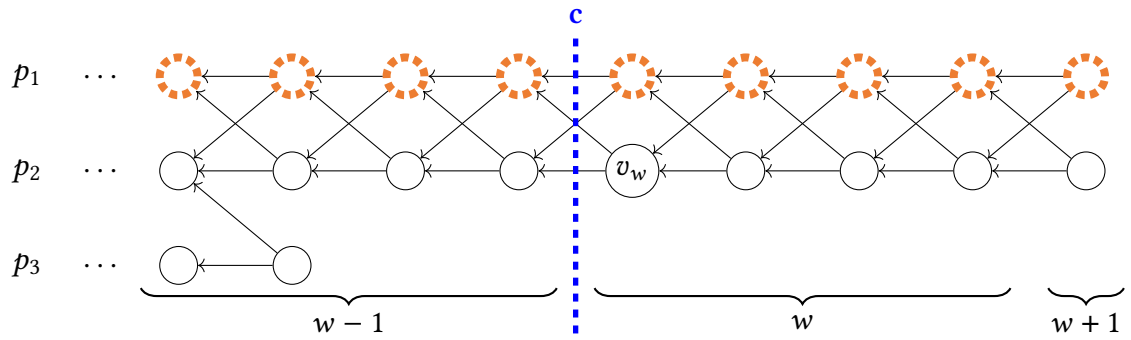


Figure 4.11: Exemplary illustration of the checkpoint protocol of NxBFT for $n = 3$ replicas. Checkpoints are aligned with waves and correct replicas try to establish a checkpoint when a wave can be committed. Replica p_1 is faulty and commits omission faults towards p_3 (illustrated by orange, dotted vertices). Replica p_3 is correct but stale. Replica p_2 makes progress together with replica p_1. The checkpoint protocol must ensure that p_3 will be able to eventually catch up, even if p_2 has already deleted old checkpoints and can no longer backfill old messages. checkpoint c will only become stable if p_2 receives a consistent and correctly signed vote from p_1. Otherwise, p_2 cannot stabilize the checkpoint and will not delete old state to be able to answer backfill request by p_3 for vertices created by p_1.

The protocol state, however, is specific and depends on NxBFT. As described above, the state transfer protocol must ensure that stale replicas can continue to participate in the protocol after the state transfer. This means that the receiving replica must be able to continue to add vertices to the TEE-Rider DAG and to order them. In particular, this means that a stale replica must be able to add all vertices it receives to its DAG if at least a single other correct replica was able to add those vertices. Hence, the state transfer must contain all information that is required to validate and add a vertex to the DAG which includes a vertex' ancestry, process identifiers, and public keys. To be able to perform coin tosses, the state transfer must also include the sealed CPRNG state (sealing is the TEE-specific way to confidentially export enclave state, see Section 2.5). To be able to produce valid vertices, the stale replica must update its enclave state after state transfer such that the enclave-managed round counter (see Algorithm 8) matches the applied checkpoint.

Checkpoint Protocol and Garbage Collection

The checkpoint protocol is a consensus protocol that allows to agree on the checkpoint data as described above. Since the output is known to correct replicas a priori, the checkpoint protocol can be implemented using a simple broadcast protocol: It only collects a quorum of signed votes.

Checkpoints are identified by an identifier c; a checkpoint belongs to wave w. Correct replicas use a predefined, configurable checkpoint interval g; a checkpoint will be created every g waves. The checkpoint is created when a corresponding wave w is committed, i.e., when the wave root v_w of w is added to the stack of committed roots (Algorithm 6, ll. 28 and 32). The last stable checkpoint must be stored by each correct replica until a new checkpoint is stabilized. Each replica maintains a set of checkpoint votes for future waves.

The checkpoint creation protocol works as follows:

1. When a replica p_i commits a wave w where $w \mod g = 0$, it creates a checkpoint $S_c, c = \lfloor \frac{w}{g} \rfloor$, consisting of:
   - a snapshot of the application data,
   - the sequence numbers and responses for the request executed last for each client,
   - the current network configuration consisting of the replica identifiers, their public keys, and their USIG public keys,
   - the sealed CPRNG state, and
   - the vertices of $\textsc{round}(w-1, 4)$ that have a path to w's wave root v_w
2. Replica p_i computes a hash $h_c = h((S_c, c))$ and a signature σ over h_c using the operator's private key[7].
3. Replica p_i broadcasts a message $\langle \text{CHECKPOINT\_VOTE}, c, h_c, \sigma \rangle$ to all replicas.
4. When a replica p_j receives a checkpoint vote message, it verifies the signature and stores the message in its checkpoint votes set.
5. When a replica p_j received $t+1$ consistent checkpoint votes for the same checkpoint c, it **stabilizes** checkpoint c.
6. Replica p_i stores the votes for checkpoint c together with the checkpoint data S_c and the hash h_c and deletes all vertices of rounds $r' \leq \text{round}(w-1, 4)$ and all checkpoints $c' < c$.

Figure 4.11 illustrates the case for three replicas where one replica is correct but stale and another replica is faulty. Note that there may be checkpoints that will never stabilize because the corresponding wave is never committed.

Remark. Re-injection to preserve AB-Validity, as proposed for Narwhal [Dan+22], is not required for NxBFT. The client logic ensures that a request is eventually handled by the federated service and, thus, ensures SMR liveness.

State Transfer Protocol

The state transfer protocol is used to transfer the checkpoint data to a stale replica. It works as follows:

[7] Please note that this is *not* a signature computed by the enclave/USIG.

1. When a replica p_i requests a vertex from a replica p_j and p_j already garbage collected the corresponding state, p_j will respond with a $\langle \text{CHECKPOINT}, c, h_c, \Sigma \rangle$ where Σ is the set of all checkpoint vote signatures collected for checkpoint c that stabilized last for p_j.
2. Replica p_i verifies that its round $r < \text{round}(cg, 1)$, that the signatures in Σ are valid signatures over h_c, and checks whether the checkpoint c is stable, i.e., $|\Sigma| > t$.
3. If a check fails, replica p_i will request the checkpoint from another replica.
4. If all checks pass, replica p_i will initiate a state transfer with replica p_j by sending a $\langle \text{STATE\_TRANSFER\_REQUEST}, c, D \rangle$ message. The set D may contain additional information for an efficient state transfer (see, e.g., [CL02, Section 6.2]).
5. Replica p_j responds with a $\langle \text{STATE\_TRANSFER\_RESPONSE}, c, S_c \rangle$ message containing the checkpoint data S_c.
6. Replica p_i verifies the correctness of the received state using h_c and Σ. If the verification fails, it will request the state from another replica.
7. If the verification passes, replica p_i will update its state with the received checkpoint data S_c, fast forward its TEE-Rider round to $r := \text{round}(cg, 1)$, set the vertices received with the checkpoint as its genesis vertices, and fast-forward its enclave by letting the enclave increment the counter to r and update the CPRNG state according to the sealed information.

Reconfiguration Protocol

As analyzed above, we must not allow a replica to rollback or reinitialize the USIG. To circumvent the impossibility result but enable recovery, we rely on a reconfiguration step. A reconfiguration protocol allows to change the set of replicas that form the distributed system, i.e., that operate an replicated state machine: Active replicas can be removed from or new replicas can be added to the system. A recovery is then the removal of a crashed replica and the addition of a new replica operated by the same operator.

To support reconfiguration, we have to introduce the concept of epochs inside the enclave. Each enclave maintains an epoch counter e. A stable reconfiguration checkpoint (see below) for epoch e allows a replica to invoke an **epoch change** to epoch $e + 1$ at its enclave. When the epoch is incremented, the enclave will reset its round counter to 1 and increment its epoch counter by 1.

The **removal** of a replica p_i works as follows; p_j is a correct replica operated by o_j, p_k is an arbitrary but fixed correct replica:

1. When an operator o_j aims to remove a replica p_i, it inputs the administrative request $\langle \text{REMOVE\_REPLICA}, p_i, \sigma \rangle$, where σ is a signature created with operator o_j's private key, in form of a special client command to the SMR framework.

2. As soon as p_k observes that $t + 1$ unique and consistent $\langle \text{REMOVE\_REPLICA}, p_i, \sigma \rangle$ messages are payload of vertices and committed by a wave root, p_k starts the removal protocol.
3. Let w' be the wave that commits the next checkpoint. As soon as p_k reaches the next checkpoint, i.e., $c = \lfloor \frac{w'}{g} \rfloor$, it will
   a) stop the ongoing consensus process until the checkpoint is stabilized, and
   b) add the removal of p_i to the checkpoint data and extend the checkpoint vote message with the reconfiguration information for transitioning from epoch e to epoch $e + 1$
4. As soon as the reconfiguration checkpoint stabilizes (see above), p_k performs an epoch change to epoch $e + 1$:
   a) Replica p_k lets its enclave perform an epoch change.
   b) Replica p_k wipes the complete vertex buffer and the DAG.
   c) Replica p_k re-injects deleted requests that were not yet ordered to its request buffer.
   d) Replica p_k restarts the consensus process in epoch $e + 1$ accepting no more vertices of p_i.

The **addition** of a replica p_i works equivalently, except that the message exchanged is $\langle \text{ADD\_REPLICA}, p_i, o_i, pk_i^{\text{O}}, pk_i^{\text{USIG}}, \sigma \rangle$, where o_i a operator identifier, pk_i^{O} is o_i's public key, pk_i^{USIG} is the USIG's public key of p_i, and σ a signature created with operator o_j's private key, and that, from epoch $e + 1$ on, p_i is allowed to propose vertices. The new replica p_i performs a state transfer to catch up with the current state of the system.

For **crash recovery**, there is a combined command $\langle \text{RECOVER\_REPLICA}, p_i, p'_i, pk_i^{\text{USIG}}, \sigma \rangle$ that can be used to exchange a crashed replica p_i operated by o_i. The command has to be signed-off by $t + 1$ unique operators and, once aided by a quorum, it will remove p_i and add p'_i as a new replica operated by o_i following the protocols above within a single epoch change.

Analysis and Discussion

We analyze the protocols for checkpoint creation, state transfer, and reconfiguration concerning their safety and liveness as well as their positive and negative impact on the performance of NxBFT.

Safety In the following, we argue that the checkpoint protocol, state transfer protocol, and reconfiguration protocol preserve safety under the assumption of an asynchronous network. The checkpoint protocol with garbage collection may only delete data, if it can guarantee that any other correct replica will be able to eventually reach the same state. To this end, it relies on the state transfer protocol to allow state replicas to catch up with the current state of the system. The state transfer protocol ensures that only stable checkpoints can be transferred and applied. Stable checkpoints are backed by $t + 1$ replicas, i.e., at least one correct replica. This quorum guarantees that the state of the system is the result of applying the correct client requests in the correct order. The stability check and the fact that a replica only fast-forwards its state in a state transfer – in particular, there is no possibility for rollbacks – the state transfer protocol preserves safety. The reconfiguration protocol could harm safety if a correct replica would execute a request that another correct replica has not executed. Hence, it must be ensured that as soon as a correct replica delivered a vertex and executed the corresponding requests, this cannot be undone and all other correct replicas will eventually do alike. The state transfer ensures that all replicas can reach the same, most current state. The alignment of the checkpoint protocol with waves ensures that the reconfiguration is executed before any "younger" wave is committed. The use of epochs allows to safely delete DAG state, i.e., future rounds, and to reset the round counter of the enclave. If the reconfiguration would not rely on epochs, replicas who already proposed vertices for future rounds have to rollback their enclave's round counter which is impossible. Moreover, if the reconfiguration protocol would not stop the consensus process until the reconfiguration checkpoint stabilizes, a correct replica would continue its operation and commit future waves that will eventually be deleted by the reconfiguration protocol. Replicas that invoke the epoch change when there is no actual epoch change cannot produce valid vertices anymore. They have either to wait until the epoch is reached or to be removed from the system. Thus, the reconfiguration protocol preserves safety.

Liveness As long as there is a quorum of replicas, the SMR protocol makes progress in ordering and executing client requests also in asynchrony. Checkpoint protocol and reconfiguration protocol preserve liveness also under the assumption of an asynchronous network: They make progress whenever they reach a quorum and there is the guarantee that eventually all correct replicas propose a vote to build said quorum. This is not the case for the state transfer protocol as a receiving replica must have the chance to receive the checkpoint data from a correct replica before all replicas already deleted the checkpoint data. If the detection of the staleness of a replica and the corresponding state transfer request take too much time on the communication layer, the requested state will be gone. The state transfer is only live when the receiving replica's communication is faster than the communication required for a quorum to establish the next checkpoint. Hence, the garbage collection interval g is equivalent to a timeout and must be chosen such that it allows, in phases of synchrony, for timely state transfer requests to be fulfilled. To this end, the state transfer protocol relies on partial synchrony in the GST model for liveness. Note that Byzantine replicas can answer state transfer requests with outdated checkpoints. If they are newer than what a receiving replica knows, it will apply the transfer. Hence,

it may "linger" behind. This can be circumvented through voting: a checkpoint is only accepted by the receiving replica if it collected a quorum for it. We do not propose this here as it would require additional communication overhead.

Effect on Memory Consumption and Communication Overhead Following the results for the expected commit latency in TEE-Rider (Lemma 4.4), the garbage collection lowers the memory consumption, in expectation, to $O(n)$. The checkpoint protocol requires the exchange of checkpoint votes resulting in $O(n)$ constant-sized messages per checkpoint interval. Reconfiguration messages are exchanged as regular client requests and do not add additional overhead. The state transfer protocol requires the exchange of state data; its size is primarily driven by the number of clients and the size of the application state.

Concluding Remarks In this subsection, we used the checkpoint concept and state transfer to circumvent the impossibility results of garbage collection in backfilling-based reliable broadcasts (see Section 4.5.2) and USIG reinitialization (see Section 4.5.3). We take advantage of the fact that NxBFT "only" has to ensure SMR safety and SMR liveness. As long as these properties are preserved, the properties of the reliable broadcast component may be broken. For garbage collection, it is sufficient that the state transfer protocol can replace the backfilling property of backfilling-based reliable broadcast (see Section 4.2). We enable recovery through reconfiguration which eliminates the need to renew a USIG identity while the replica identity remains the same. On closer inspection, reconfiguration is a setup protocol based on a functioning BFT consensus protocol in which agreement must be reached on only a single party. This circumvents the impossibility result from Section 4.5.1.

5 Performance and Resilience Evaluation of TEE-Based State Machine Replication

From a theoretical point of view, the results of the previous chapter appear to be superior in a number of categories: fault tolerance is maximized, replicas are maximally independent from the quality of the network, and the leader bottleneck is eliminated. In comparison to state-of-the-art partially synchronous approaches, however, the number of messages is increased and probabilism as well as the wave construct increase the expected commit latency. While this fact is known from the literature (e.g., [Dan+22; Aru+25; LNS25]), the gains of the *hybrid fault model in asynchrony* and the impact of a *BFT SMR client on DAG-based atomic broadcast* have not been evaluated in the literature so far. Moreover, related work has shown more than once that theoretical results do not necessarily translate into practical performance. Practical performance is, amongst other factors, influenced by implementation quality [Din+17; WKM24; NO25], the programming language [Lia+24; WKM24], the network stack [Sec+24], network quality [BTR24; LG24], selected cryptographic primitives [Hyl+24; SRC24], and, obviously, parametrization [Ami+24; KK25]. An evaluation of NxBFT and the NxB client model simply based on analytical results or reported performance results of related work falls short of providing a complete picture.

Consequently, a comprehensive and comparative evaluation of NxBFT and the NxB client model is necessary to understand their practical performance characteristics, benefits, and limitations in contrast to state-of-the-art approaches. The evaluation approach must ensure that the comparison is fair and conclusive. We must ensure that we only consider the effects of algorithm logic, and not those resulting from different uses of, e.g., network technology, cryptography, or programming languages. In particular, to ensure plausible and conclusive results, a reasonable, scientifically grounded software and experiment design is required.

As discussed before (see Section 4.1), the distributed systems research community investigates three major paradigms: leader-based protocols with a static leader in the tradition of PBFT [CL02], leader-based protocols with a rotating leader that follow the streamlined pattern in the tradition of Chained HotStuff [Yin+19, Section 5], and leaderless protocols. Leaderless protocols are not necessarily DAG-inspired but the most performant, state-of-the-art protocols (e.g., Autobahn [Gir+24], Shoal++ [Aru+25], and Raptr [Ton+25]) follow this paradigm. Our goal is to evaluate the performance of NxBFT, which itself belongs to the category of leaderless, DAG-based protocols, and to clearly identify its strengths and weaknesses. One could argue that this requires the comparison of NxBFT with other Byzantine fault-tolerant, DAG-based protocols that do not operate in the hybrid

fault model. We, however, see no need for such a comparison: The analytical investigation (see Sections 4.2 and 4.3) and related work (e.g., [Ver+11; DCK16; Liu+19; Dec+22a; Dec+24]) show that hybrid fault-tolerant approaches always outperform their non-hybrid counterparts. Thus, we do not provide a general comparison between TEE-based and non-TEE-based approaches. Instead, we want to investigate

1. in which deployment scenario which TEE-based SMR approach is superior,
2. if the benefits from using a DAG-based approach sustain when operating in the hybrid fault model, and
3. which paradigm benefits to which extent from the NxB client model.

For a complete picture, which we want to achieve, we have to select a reasonable, TEE-based algorithm from each category. To this end, we choose MinBFT (static leader) [Ver+11], Chained-Damysus (streamlined) [Dec+22a, Section 7], and NxBFT (leaderless; this work). While MinBFT is already over 10 years old, it still serves as a reasonable and, in our opinion important, *baseline*. Chained-Damysus was the first proposal that transformed Chained HotStuff to the hybrid fault model using a small TEE.

The contributions of this chapter are as follows:

- We introduce the ABCperf framework that aids the fair comparison of state machine replication and atomic broadcast algorithms (Section 5.2).
- We describe the implementation of the analyzed algorithms which are MinBFT (static leader), Chained-Damysus (streamlined), and NxBFT (leaderless) (Section 5.3).
- We investigate the influence of the different NxBFT algorithm phases on NxBFT's performance and outline potential optimizations (Section 5.4).
- We conduct extensive experiments to compare the performance of MinBFT, Chained-Damysus, NxBFT in terms of throughput, latency, and resilience under the BFT client model, the NxB client model, different payload sizes, different network sizes and latencies, as well as under crash faults (Section 5.5).

We discuss related work in Section 5.1 and conclude the chapter with a discussion of results and limitations in Section 5.6.

The ABCperf framework was originally developed in the bachelor's thesis by Tilo Spannagel [Spa22] and later published by Spannagel, Leinweber, Castro, and Hartenstein [Spa+23]. The MinBFT implementation was created by Tilo Spannagel, Adriano Castro, Bela Stoyan, Conrad Teichmann, and Marc Leinweber. Tilo Spannagel assisted in the implementation of Chained-Damysus and NxBFT. The comparison of TEE-based common coins stems from the master's thesis by Marius Haller [Hal25]. NxBFT and the comparison study were previously published by Leinweber and Hartenstein [LH25]. All software used in this chapter, except the common coin comparison, is available open source as part of the ABCperf project: `https://github.com/abcperf/abcperf`.

5.1 Background and Related Work

We discuss related work in the context of the evaluation of state machine replication algorithms and the impact of implementation choices.

5.1.1 Comparison and Analysis of BFT SMR Algorithms

The discussion of related work on state machine replication evaluation is further divided into two parts: the evaluation methodology and the evaluation results.

Evaluation Methodology

Empirical evaluations in distributed systems research typically perform measurements on either simulated, emulated, or real-world systems [Whi+02; Bou10; Law15]. Informally, emulation software imitates the behavior of a real system whereas simulation applies a (statistical) model of the system to predict its behavior under certain conditions [Whi+02; Col17]. Even when the evaluation is based on real-world systems, it is common to use simulation or emulation to test the system under different conditions, such as varying workload, network conditions, or fault scenarios [Gra+23; BTR24; LG24]. When looking at fault injection and randomized network behavior, the gap to fuzz testing, which is the randomized testing of software to find implementation flaws and security vulnerabilities [Sch+24], becomes very small (e.g., [Ban+21; NO25]). The same techniques can be used to judge on the resilience of a system [Vie+12]. Our experiment framework ABCperf supports the emulation of omission and crash faults as well as network latency emulation. Moreover, we use randomized integration tests to ensure the implementation quality of the algorithms.

In the context of BFT SMR, the systematic comparison of algorithms gained momentum in recent years. For an overview, we have to distinguish between solutions focusing on BFT SMR (or atomic broadcast) and those that focus on full blockchain ecosystems. Table 5.1 summarizes the comparison of evaluation frameworks.

To the best of our knowledge, the first systematic comparison of BFT SMR algorithms was conducted by Singh et al. [Sin+08] using their framework BFTSim. Tool [Wan+22] and the work by Berger et al. [BTR24] are two recent simulation-based frameworks that focus on the evaluation of BFT SMR algorithms. While BFTSim [Sin+08] and Tool [Wan+22] only require a developer to define the algorithm logic to conduct experiments, Berger et al. [BTR24] require to define a full-fledged replica software. In particular, it is not sufficient to only supply the state machine replication logic or the atomic broadcast algorithm respectively as an experiment input. All three use network simulation. Berger et al. [BTR24] find that for small networks ($n < 32$), network simulation does not give a good approximation of the performance of BFT SMR algorithms while for larger network sizes, network simulation is sufficient as network latency becomes the dominating factor. We additionally argue that the assignment of compute resources to simulation processes and the throughput estimation is

Framework	Prototype	Network	Faults	Workloads
BFTSim [Sin+08]	Logic	Simulated	Crash	Random bytes
Blockbench [Din+17]	Full	Emulated	Crash	Random bytes, custom
BCTMark [SLM20]	Full	Emulated	—	Custom
Gromit [Nas+22]	Full	Emulated	—	Asset transfer
Tool [Wan+22]	Logic	Simulated	Byzantine	—
Diablo [Gra+23]	Full	Native	—	Predefined applications, custom
METHODA [Rez+23]	Full	Emulated	Custom	Custom
Berger et al. [BTR24]	Full	Simulated	Crash	Random bytes, DoS
ABCperf (this work)	Logic	Emulated	Omission	Random bytes, custom

Table 5.1: Comparison of evaluation frameworks for BFT SMR algorithms. For each framework, we list whether it requires a full-fledged replica software or only the algorithm logic, whether it uses network simulation, emulation or native execution, what type of fault emulation is supported, and what type of workloads are supported.

less straightforward than for native execution or emulation. Typically, throughput is given as the number of requests processed per second. If throughput is to be estimated, the actual computation work of the protocol must be done somewhere. Simulation is used to save resources, but the simulation processes must be assigned to physical machines such that the required computation does not suffer from resource contention; otherwise, the throughput estimation is not representative. We argue that in such cases resource allocation will be similar to emulation or native execution. The simulation-based frameworks do not support the definition of custom workloads or replicated applications; their focus is primarily to investigate the effect of network conditions and faults on the performance of BFT SMR algorithms. Bedrock [Ami+24] is a generic SMR implementation platform that aids the implementation of SMR algorithms; while the paper provides a comparison of state-of-the-art algorithms, Bedrock itself is not an evaluation framework (no network manipulation, no fault emulation, limited workloads). However, the authors of Bedrock emphasize the importance of a fair comparison by using identical programming languages, network stacks, and libraries; we fully agree with this requirement.

Blockbench [Din+17], BCTMark [SLM20], Gromit [Nas+22], and Diablo [Gra+23] focus on the evaluation of blockchain systems. All four require a full-fledged replica software making them unsuitable for the evaluation of BFT SMR algorithms in isolation and hindering fair comparison and fast prototyping. Besides Blockbench, none of the frameworks

supports the investigation of faults. We observe that the blockchain-focused frameworks do not use network simulation; they either use emulation or native execution to produce realistic results. Lebedev and Gramoli [LG24] find that network emulation is capable of producing realistic results for BFT SMR algorithms and that the use of emulation improves the reproducibility of the results. METHODA [Rez+23] is a generic framework for the evaluation of distributed algorithms; its high flexibility, however, comes at the cost of requiring a full-fledged replica software and the custom specification of workloads and faults.

Aside from the evaluation frameworks, nearly every recent proposal with a system-oriented approach to BFT SMR includes a thorough evaluation. In most cases, e.g., [Dec+22a; Dec+24; Gir+24; Aru+25], the evaluation is conducted on cloud deployments and uses a randomly generated byte sequences of a fixed size following a targeted request rate as workload. Especially DAG-based approaches also investigate the impact of faults (e.g., [Dan+22; Aru+25; Ton+25]).

ABCperf, our contribution, is the first framework combining the advantages of simulation frameworks and emulation/native execution frameworks: it only requires the developer to implement the algorithm logic, it supports network emulation to produce realistic performance results, and it supports the emulation of omission faults. To maximize the fairness of the comparison, coordination algorithms and application logic are implemented in the same programming language; developers are not required to implement more than the actual algorithm or business logic. Moreover, it ships with the ability to define custom workloads allowing to investigate the impact of application logic, such as the size of the payload or the complexity of the business logic, and Byzantine client behavior on the performance of BFT SMR algorithms. The flexibility for workload and application definition also allows for prototyping the replicated application itself without the need to implement a full-fledged replica software. Finally, we offer a web-based live exploration mode that allows live manipulation of configuration parameters and that supplies real-time performance metrics.

Evaluation Results

All recent evaluations find that the performance of BFT SMR algorithms is highly dependent on the network size and the network latency: with increasing network size, throughput decreases and latency increases (see, e.g., [Ami+24; BTR24; Kan+25b]). The leader bottleneck is also well-documented [Cra+18; Gol+19; Bie+12; SPV22]. The impact of the leader bottleneck is especially pronounced in the presence of faults, as the leader must handle all requests and responses [Dan+22; Spi+22; BTR24]. Lawniczak and Distler [LD24] observe that a fault at the leader can lead to a metastable state. In such cases, the protocol is not able to recover performance after the fault is resolved (e.g., by a view change). Leaderless approaches show way better performance in the presence of faults [Dan+22; BTR24; Gra+24; Bab+25]. Moreover, if not used with a BFT client, DAG-based algorithms can increase peak throughput when increasing n [Dan+22; Spi+22; Wan+24]. Suitable concurrent handling of messages, i.e., parallelized software, can amplify this effect

[BSA14; BDK17; NMR21; Dan+22; Ein24]. The hybrid fault model is known to improve performance significantly [Ver+11; DCK16; Dec+22a]. Our experiments (1) confirm these findings and (2) show that the combination of DAG-based atomic broadcast and the hybrid fault model is beneficial for even more increased performance and resilience.

5.1.2 Impact of Implementation Level Choices

There are at least four categories of implementation level choices that can have a significant impact on the performance of BFT SMR: programming language, cryptographic primitives, network stack, and parallelization strategies. In general, said choices have a positive or negative impact independently from the deployed SMR algorithm. There are, however, some choices that are more relevant for certain algorithms than others.

Programming Language The choice of the programming language has a significant impact on the performance of an implementation. It defines the available libraries, computation time, memory consumption, type safety, and the ease of implementation. So-called system programming languages like C, C++, and Rust are typically used for performance-critical applications, as they allow for low-level memory management and fine-grained control over the execution flow[1]. Rust, in particular, is known for its memory safety guarantees and zero-cost abstractions which can lead to high performance without sacrificing safety [Bal+17; Jun+18; Zha+22]. In contrast, higher-level languages like Python, JavaScript, or Go are typically slower due to their interpreted nature and garbage collection mechanisms, but they offer higher productivity and ease of use. Liang et al. [Lia+24] investigate the impact of programming languages on SMR performance and correctness. They find that system programming languages are in terms of performance superior to languages with garbage collection and conclude that, while having a steeper learning curve, Rust is a perfect choice for "less sloppy and less error-prone" [Lia+24, Section 6] interfaces when implementing SMR algorithms. Matsakis [Mat25] and Endler [End25b] argue that Rust is the perfect choice for "foundational" software, emphasizing that Rust is not limited to classic system-level programming. Endler [End25a] points out that Rust is a good choice for sustainable prototyping. In this work, we use Rust as the programming language for the implementation of all algorithms under investigation and our evaluation framework.

Cryptographic Primitives Cryptographic primitives are a crucial part of modern BFT SMR algorithms, as they ensure the integrity, authenticity, and confidentiality of messages. Additionally, they are at the core of a Trusted Execution Environment (TEE). In general, a cryptographic function, regardless of whether it is used for encryption, signing, or hashing, implies a non-negligible computational cost. Kuznetsov et al. [Kuz+23] conduct a

[1] See, e.g., `https://github.com/kostya/benchmarks/blob/master/README.md` (accessed 2025-08-04) for a comparison of programming languages in terms of performance.

benchmark of cryptographic hash functions and find that SHA2 [Nat15a] and BLAKE are the best choices in terms of performance. After observing performance issues with the hash function in the NxBFT implementation, we conducted a benchmark comparing SHA2, SHA3, BLAKE2, and BLAKE3 with different input sizes. We find that the optimal choice of the hash function depends on the expected input size and, crucially, on the availability of hardware acceleration. The choice of the signature scheme, especially the elliptic curve being used, also has a non-negligible performance impact [BL07]. Heß et al. [HHM24] conduct a microbenchmark of the performance of a threshold signature scheme based on the BN254 curve pair [BN05] and show that a *single* signature verification takes 4.5 ms highlighting the cost of threshold cryptography. Seck et al. [SRC24] show that the choice of the signature implementation can influence the latency of HotStuff significantly. Finally, the TEE itself has, due to context switching and memory management, a non-negligible overhead [Mof+18; Mur+25] which can be addressed through enclave-aware programming [WAK18] and is typically accepted due to the savings caused by the hybrid fault model.

Network Stack As pointed out before, in SMR research we typically assume secure, reliable point-to-point links. In practice, this is achieved by a combination of routing and transport layer protocols [CGR11, Section 2.4.7]. Castro discusses the match of TCP [Edd22] and asynchronous systems and concludes that they do not match well [Cas00, Sec. 5.2]. Modern implementations, however, typically rely on the TCP implementations of the operating system[2]. Seck et al. [Sec+24] confirm that TCP is the best choice for BFT SMR algorithms in terms of performance and reliability; TLS [Res18] should be used to implement a secure overlay over TCP. Moreover, they find that packet loss above 0.5% already degrades the overall performance and loss of 2% yields a slowdown of approximately an order of magnitude. Consequently, ABCperf uses TCP with TLS as the network stack.

Parallelization Strategies We already mentioned the positive impact of parallelization: in leaderless and leader-based approaches, it is highly beneficial to handle incoming messages from peers and clients in parallel. Until those messages reach the "core" of the SMR framework, i.e., the atomic broadcast and the application logic, this parallelization is rather straightforward as there are typically no dependencies between the messages (e.g., serialization, deserialization, and TLS management). However, once the messages reach the core, parallelization becomes more complex. For USIG-based approaches, e.g., each replica maintains a data structure that manages the expected counter values. This can be solved by a worker thread per peering replica such that each worker can maintain its own independent data structure. But when it comes to deriving total order, all messages have to be honored. For DAG-based algorithms, the DAG has to be maintained and interpreted to derive consensus decisions. Parallelization in a way such that the overhead does not diminish the performance gain is not trivial. Danezis et al. propose a parallelization strategy

[2] See, e.g., the Aptos ledger: `https://github.com/aptos-labs/aptos-core/blob/main/network/README.md` (accessed 2025-08-05).

for Narwhal that allows to process multiple messages in parallel. For the management of the DAG, they use a single "primary" thread [Dan+22, Section 4.2]. As cryptographic operations like hashing, signature creation, and signature verification are a known bottleneck [BSA14; Hyl+24], such a parallelization strategy has a significant positive impact on the overall performance (see, e.g., [BSA14, Figure 7], [BDK17, Figure 5], and [Dan+22, Figure 7]). Chop Chop [Cam+24] is a wrapper for atomic broadcasts that takes validation tasks and batch creation off the critical path. This tasks are moved to a so-called distillation layer that can process independently of the main protocol execution. In its core, Chop Chop is a clever parallelization strategy. Our measurements confirm the observation of the Chop Chop authors that cryptographic tasks are the main computational bottleneck of modern atomic broadcast algorithms. ABCperf supports parallelization on the network layer for serialization, deserialization, and the management of the secure and reliable links. We deliberately do *not* parallelize the atomic broadcast logic to enable thorough testing of the algorithms. During our experiments, we identify where we reach the limits of this single-thread policy and where parallelization would be beneficial. Suri-Payer et al. [Sur+21] and Gelashvili et al. [Gel+23] identify sequential execution of the application logic as a bottleneck in BFT SMR algorithms and propose ways to allow deterministic, conflict-free parallel execution. This is out of scope for this work as we focus on the differences between the algorithms and not on the application logic.

5.2 ABCperf: Ensuring Plausible, Fair, and Reproducible Results

The discussion of related work shows that when designing a distributed system, it is crucial to consider the performance implications of various architectural and algorithmic choices. In particular, precise knowledge of the algorithm behavior under various conditions is necessary and the ability to evaluate differing system designs for a use case is needed. To this end, we propose and use the ABCperf framework. ABCperf provides a systematic approach to performance evaluation of prototypical distributed systems. In the following, we first describe ABCperf's objectives and design considerations. We then introduce ABCperf in more detail and also present our laboratory setup.

5.2.1 Objectives and Design Considerations

While ABCperf is not limited to consensus-based SMR, it is designed with a focus on the specific challenges and requirements of such systems with the following objectives and design considerations.

Fair and Straightforward Comparison The goal is to compare SMR frameworks fairly by focusing only on design decisions that affect coordination and consensus between replicas. In other words, we examine how a given SMR framework implements this core

logic. To ensure fairness, we exclude the routine tasks that all SMR frameworks and atomic broadcasts must handle regardless of their paradigm. Examples are serialization, link management, scheduling, or client handling. To this end, we provide a complete abstraction of the network stack in form of a peer-to-peer communication middleware allowing the SMR framework implementation at an abstraction level similar to pseudocode. The interaction between replicas and clients is handled by ABCperf and, thus, identical for all algorithms.

Powerful Emulation for Estimation of Real-World Performance Network conditions, such as latency and packet loss, as well as actual faults have a significant impact on the achievable performance of an SMR framework. To get an estimate for the real-world performance under differing conditions, ABCperf uses network emulation to enable configurable network latency and packet loss between replicas and clients. The use of network emulation and the deployment on cloud or cluster infrastructure allows for a more realistic evaluation environment yielding performance numbers that can be achieved in an actual deployment as well. Moreover, the peer-to-peer middleware can emulate omission faults. We deliberately do not support Byzantine fault injection (e.g., invalid messages or message reordering): Correct replicas use authentication mechanisms and maintain state information to detect and drop invalid messages. Selective omission faults will trigger similar behavior for the correct replicas (except for validation overhead). The consensus logic ensures that equivocation faults will be tolerated. Consensus logic is different for every algorithm. In fact, in TEE-based algorithms, equivocation is prevented in the first place. As a result, it is hard to implement a generic equivocation fault emulation layer. Moreover, the correct implementation of the mechanisms allowing to handle Byzantine faults should be ensured before evaluating the performance of an SMR framework, e.g., by using blackbox and randomized testing.

Exchangeable Application Layer ABCperf is designed to be agnostic to the specific application logic running on top of the SMR framework. Thus, it is possible to evaluate (1) the effects of different replicated applications on the performance of an SMR framework and (2) the performance and the impact of different design decisions of the replicated application itself. As with the SMR/consensus logic, the application can be implemented against a well-defined interface with ABCperf taking care of the necessary communication.

Flexible and Complex Analysis It often happens that one only knows which measurements to collect after having carried out an experiment. For example, one may not want to include all clients or one may want to exclude a warm-up and cool-down phase. That is why ABCperf does not just generate statistical key figures. Where possible, the raw data is stored in a database so that complex evaluations can then be carried out using SQL queries.

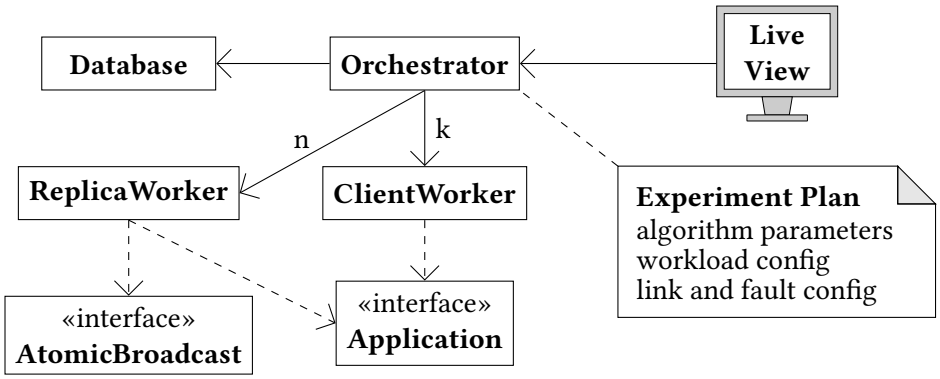


Figure 5.1: ABCperf architecture. An ABCperf cluster consists of an orchestrator, n replica workers, and k client workers. The replica workers execute the server side of the SMR framework and the application logic. The client workers execute the client side of the SMR framework and the application logic and generate client requests. SMR logic and application have to provide bindings to the `AtomicBroadcast` and `Application` interfaces, respectively. The orchestrator sets up the experiment, defines the experiment parameters, provides the live view, and optionally stores the experiment results in a database. The live view provides real-time manipulation of parameters and real-time rendering of metrics.

Live Exploration To allow quick feedback loops and manual sensitivity analysis, ABCperf provides a web-based live exploration mode. In this mode, load, network, and fault parameters can be adjusted in real-time. The live view renders real-time performance metrics of the overall system and each replica.

No SMR Implementation Platform ABCperf is no SMR implementation platform like, e.g., Bedrock [Ami+24], and cannot be used to run a replicated application with production quality. We simplified the implementation of ABCperf where possible without manipulating the performance of the SMR framework significantly. For example, clients and replicas are identified with simple integer IDs, there is a fixed number of clients, there is no client registration process, and all state is permanently stored in main memory.

5.2.2 Architecture Overview

ABCperf is written in Rust, chosen for its performance, type safety, and sustainable prototyping capabilities. The architecture of ABCperf is shown in Figure 5.1. An ABCperf cluster consists of an orchestrator, n replica workers, and k client workers. The orchestrator is a dedicated ABCperf service that reads the configuration, assists in establishing the peer-to-peer network and schedules experiments. Moreover, it provides a live view, allows manipulation of an ongoing experiment, and collects metrics from clients and replicas to store them in the database. The replica workers execute the server side of the SMR framework and application logic. The client workers execute the client side of the SMR framework and application logic generating the actual load in form of client requests the replicas have to handle. ABCperf should be deployed to a cluster (multiple replicas per physical server are possible), letting it scale with the hardware.

During an experiment, each replica worker uses ABCperf's peer-to-peer middleware which offers two features: First, it offers a message passing interface with eventual delivery, abstracting the complete communication stack and allowing to send arbitrary messages to replicas by addressing them with simple integer IDs. Second, it is capable of emulating both fault and network behavior transparently. The messaging middleware is implemented using the highly efficient and widely adopted (e.g., by [Dan+22; Spi+22; Gir+24; Aru+25]) `tokio` framework[3]. Omission faults are emulated by dropping messages from or to replicas marked as faulty following an adjustable probability distribution. Latency and packet loss are emulated using the capabilities of netem[4]. The configured latency value is added to the native round trip latency of the experiment cluster. To this end, ABCperf emulates 50% of the configured latency in each direction.

The users of ABCperf are required to implement two interface types that are used by replica and client workers: `AtomicBroadcast` and `Application`. `AtomicBroadcast`[5] is an interface for the server-side logic of an SMR framework (or, for simpler cases an atomic broadcast solely) and makes simple replacement of coordination and consensus logic possible whilst encouraging a clean implementation at an abstraction level similar to pseudocode. Users of ABCperf are only required to implement algorithm state, message handling and induced state transition, as well as message generation. In particular, the transport protocol, serialization, peer discovery, or parallelization do not have to be considered. The code implementing the `Application` interface implements the decentralized end-user application. It implements the server-side of the replicated application as well as the client-side logic. Moreover, it needs to provide a way to generate client requests. ABCperf is capable of running full-fledged decentralized applications. Nonetheless, they are typically simplified to facilitate the emulation of client requests (e.g., allowing simpler patterns of consecutive requests).

ABCperf has a default application we call the **no-op application**. In the no-op application, the client requests have a configurable, fixed payload size. To prevent overhead from generating random byte patterns, every payload bit is set to 0. ABCperf is configured to not use compression between replicas. Thus, the configured payload length has a significant impact on the data that has to be exchanged between replicas. The no-op client is a synchronous client: as long as it has not received a response for its current request, it will not send any new requests. After a request is forwarded by the SMR framework to the server-side application logic, the server-side application logic does not perform any operation but simply forwards the request to the client as a response.

5.2.3 Experiment Organization and Metrics

After the orchestrator initialized each worker, helped the replicas to form the peer-to-peer-network, and the setup phase of the SMR framework is completed, the experiment is

[3] `https://tokio.rs` (accessed 2025-08-11)

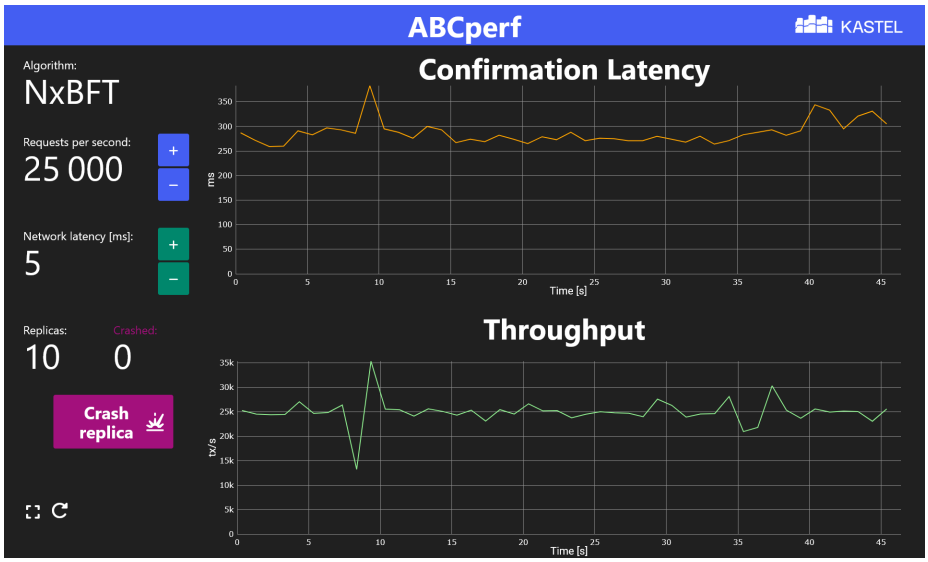
[4] `https://man7.org/linux/man-pages/man8/tc-netem.8.html` (accessed 2025-08-11)

[5] `SMRServer` would have been a better name but we stuck to `AtomicBroadcast` for historic reasons.

abcperf.yml

```yaml
experiment_label: fault_and_network_latency
experiment_duration: 70 # seconds
n: 10
t: 4

abc:
    algorithm: nxbft-tee
    config:
        vertex_timeout: 0.1 # seconds
        min_vertex_size: 100 # requests

client_workers: 5

orchestrator_request_generation:
    load:
        - ticks_per_second: 1000 # Hertz
          requests_per_tick: 25 # 1000 * 25 = 25 kOp/s
          invocation_time: 0
    maximum_client_instances: 10000
    distribution: constant
    smr_client: fallback
    fallback_timeout: 5 # seconds

application:
    name: noop
    config:
        payload_size: 256 # bytes

fault_emulation:
    - replica_id: 0
      omission_chance: 1 # = 100%
      invocation_time: 60 # seconds

network:
    replica_to_replica: []
        - latency: 5 # milliseconds
          jitter: 0 # milliseconds
          packet_loss: 0 # = 0%
          invocation_time: 0 # seconds
    client_to_replica: []
```

Figure 5.2: Exemplary ABCperf configuration file in YAML format. The experiment is configured to run the NxBFT algorithm for 70 s with 10 replicas. NxBFT is configured to create a vertex at least every 0.1 s; for faster creation the vertex must contain at least 100 requests. The load is generated by 5 client workers of which each spawns 10000 clients at maximum. The client workers together generate requests with a constant rate of 25 kOp/s. The "fallback" SMR client indicates that the NxB client model is being used; after 5 s of no answer a client contacts a different replica. The no-op application is used with a payload size of 256 Bytes. After 60 s, the replica with id 0 shows 100% omission faults. The network between replicas is configured with a constant round trip latency of 5 ms. Clients communicate with the default latency of the experiment cluster.



(a) Overview and performance metrics

(b) Replica resource utilization

Figure 5.3: Screenshots of the ABCperf live view for the experiment as described by the config file in Figure 5.2. Figure 5.3a shows the overall performance metrics, i.e., end-to-end-latency and throughput, while Figure 5.3b shows the replicas' resource utilization, i.e., CPU usage, RAM usage, and bandwidth usage. It is possible to to crash randomly selected replicas (omission rate of 100%). Metrics are rendered in real time.

started. The client workers generate requests following the configured load patterns. In most cases, this will be a fixed request rate (e.g., 50 kOp/s). Each client worker measures each client request issued (i.e., invocation and processing time). Replica workers record their resource utilization. An experiment has a predefined experiment duration; once, the experiment duration is reached, the orchestrator stops all workers and collects the measurement data. The measurement data is then written to the database. An exemplary ABCperf configuration file is listed in Figure 5.2.

The main performance metrics of an SMR algorithm are end-to-end latency and throughput. The **end-to-end-latency** (or confirmation latency) is the time it takes for a client request to

be processed and sufficient responses to be received. The **throughput** is the number of successfully answered client requests per time unit (typically seconds). The end-to-end latency heavily depends on the network conditions and the load on the system. If stressed with a certain request rate defined in kOp/s, an average end-to-end latency of more than 1 s is a clear sign for system overload: the system handles per second less requests than generated per second which leads to growing queues and, thus, increasing end-to-end latency with increasing experiment duration. However, an end-to-end latency of less than 1 s and throughput matching the request rate does not necessarily imply that the system is in a stable state. Instead, the queues may just grow slowly. Consequently, sustained throughput can only be determined by observing the system for an extended period of time and comparing the metrics for different time intervals of the experiment. Figure 5.3 shows screenshots of ABCperf's live view for the experiment as described by the config file in Figure 5.2.

To allow for a more (over)load-independent performance analysis, ABCperf also measures the **intermediary decision time (IDT)**. The IDT is the time between two decisions of the server-side SMR framework. A decision is made when the server-side SMR framework forwards one or more requests to the server-side application logic that then produces responses for the clients. While the IDT obviously is not independent of network conditions and load, it is more robust against short-term fluctuations as it does not implicitly measure queue lengths. When generating requests with a constant rate per second, an IDT over 1 s is not a sign of overload. This is in contrast to the end-to-end-latency and caused by batching effects that allow the system to decide on all requests that have accumulated since the last decision at once. Such batching effects make it possible that the average end-to-end latency is below 1 s while the IDT is above 1 s.

5.2.4 Laboratory Setup

The laboratory setup consists of an experiment cluster with multiple nodes assigned to different roles (orchestrator, replica workers, client workers). Each cluster node runs a Docker container[6] with the ABCperf software stack, allowing for easy deployment and management of the experiment environment. Our experiment cluster consists of 25 servers (or cluster nodes) with the following hardware configuration and role assignment:

- Orchestrator: Intel Xeon E-2288G CPU (8 cores), 64 GB main memory, 1 TB NVMe SSD, 1 GBit/s uplink
- Replica workers 00–09: Intel Xeon E-2288G CPU (8 cores), 64 GB main memory, 512GB NVMe SSD, 10 GBit/s uplink
- Replica workers 10–19: Intel Xeon E-2388G CPU (8 cores), 64 GB main memory, 512GB SATA SSD, 10 GBit/s uplink

[6] `https://docs.docker.com/get-started/docker-overview/` (accessed 2025-0812)

- Client workers 0–3: AMD EPYC 9274F CPU (24 cores), 128 GB main memory, 1 TB NVMe SSD, 10 GBit/s uplink

The experiment cluster is distributed over two switches which are interconnected with an aggregated 60 GBit link. Thus, the experiment cluster can communicate with at least 6 GBit/s between every cluster node. The base network roundtrip latency of our cluster is $\sim$ 0.15 ms. The actual number of replicas is equally distributed among the 20 replica nodes. To speed-up storing and analyzing of experiment data (the experiments conducted for this dissertation produced about 12 TB of raw data), we store the data in a ClickHouse database which is optimized for storing and analyzing large amounts of data[7].

5.3 Implementation of Algorithms

In this section, we describe the implementation of the three algorithms at investigation: MinBFT, Chained-Damysus, and NxBFT. Please note that our implementations do not provide a fault-tolerant setup (we use a trusted setup within ABCperf) and, except for MinBFT, no garbage collection or state transfer capabilities. We deem this reasonable as such mechanisms only have a significant impact during recovery procedures and not during normal operation. Additionally, we argue that the performance of checkpoint, state transfer, and recovery procedures is only secondarily influenced by the SMR algorithm's paradigm which is why we see them as out of scope for the empirical evaluation of this dissertation. In the following, we briefly describe the TEE being used (Intel SGX), relevant implementation details for all three algorithms, the test strategy, and the algorithm parametrization.

5.3.1 TEE: Intel SGX

Intel SGX, which is short for Software Guard Extensions, is a trusted execution environment developed by Intel [McK+13; CD16]. Intel SGX is an extension to the CPU firmware. The Skylake microarchitecture, released in August 2015, was the first generation to support SGX. SGX allows to define enclaves which are isolated from the rest of the system. The main memory area of an enclave is encrypted and managed by the CPU firmware; data enters and leaves the CPU only in encrypted form. An enclave depends on the operating system for I/O operations. Every SGX-capable CPU has a burnt-in hardware secret that can be used to generate attestations. We refer the reader to Costan and Devadas [CD16] for a detailed and thorough explanation of Intel SGX.

Although we are aware of the limitations and weaknesses of SGX (e.g., [Bul+18]), in our experiments SGX serves as a viable way to account for the overhead of enclaved execution. Moreover, we want to highlight that typically such vulnerabilities are hard to exploit in

[7] `https://clickhouse.com/docs/intro` (accessed 2025-08-12)

practice and mitigations have been implemented in newer CPU generations [Wik25a]. Please see Section 6.1 for a more detailed discussion.

We use the Rust SGX SDK by the Apache Teaclave project[8] to implement the enclaved functionality. As the remote attestation based on Intel's attestation infrastructure was deprecated by Intel and now a self-hosted attestation infrastructure is required, we do not support attestation features in our prototypes. Because attestation is only required during setup, the missing attestation has no impact on the evaluation results.

5.3.2 MinBFT

To the best of our knowledge, there exists no feature-complete MinBFT implementation satisfying our requirements. The implementation governed by the Hyperledger project[9] does not support view changes and is not longer maintained. The implementation by Kim Hammar[10] is a Python prototype that was published after we started our research. We, therefore, implemented MinBFT from scratch in Rust, using the descriptions by Veronese et al. [Ver+11] as a basis.

Our implementation follows the original message pattern: the incumbent leader broadcasts a PREPARE message whenever it has a batch of requests ready to order. The leader follows a simple batching strategy: It waits for a defined number of requests it can batch before sending a PREPARE message. When the minimum batch size is not reached within a configurable time interval, the PREPARE message is sent anyways. The timeout cannot trigger a PREPARE message with an empty batch. Receiving replicas echo the leader's proposal in a COMMIT message; the COMMIT message contains the complete message of the leader as a payload thus ensuring reliable broadcast of PREPARE messages. The USIG module signs message hashes; signatures are used to implement a hybrid fault-tolerant FIFO reliable broadcast (see Section 4.2). The hash function used is SHA2-256 [Nat15a] provided by the `sha2` crate[11]. The signature is an ECDSA signature with a key size of 256 bits [Nat23a] provided by Intel's SGX SDK. Replicas start a timer whenever they receive a client request; if the timer expires before the request was ordered, the replica requests a view change. The current timeout interval will be doubled whenever a client timeout is triggered (exponential backoff). MinBFT is the only of the three algorithms for which a checkpoint mechanism is implemented; we follow the original description in the paper.

The following behavior differs from the original description:

- A replica forwards a valid NEW − VIEW message to all other replicas to ensure that the message is reliably broadcast. Otherwise, some correct replicas may enter the view while others will initiate a second view change.

[8] `https://github.com/apache/incubator-teaclave-sgx-sdk` (accessed 2025-08-12)
[9] `https://github.com/hyperledger-labs/minbft` (accessed 2025-08-12)
[10] `https://github.com/Limmen/minbft` (accessed 2025-08-12)
[11] `https://crates.io/crates/sha2` (accessed 2025-08-13)

- The incumbent leader does not send a COMMIT message for its own PREPARE. Instead, the leader's PREPARE message is treated as a COMMIT message as well. This does neither harm safety or liveness (see Section 4.2) but saves messages.
- When the leader has a new batch ready, it sends a PREPARE message without waiting for the previous batch to be committed. This is in line with the sliding-window mechanism of PBFT [CL02, Section 6.1]; we, however, use an unbounded window size.

5.3.3 Chained-Damysus

Damysus was published in 2022 by Decouchant et al. [Dec+22a]; the authors made their C++ prototype available open source[12]. Since the original prototype implementation is very difficult to understand, some identifiers are named differently in the paper and in the implementation, and integration into ABCperf would have required several foreign function interface definitions, we decided to create our own implementation in Rust. We based our implementation on the descriptions in [Dec+22a, Section 7].

Chained-Damysus proceeds in rounds that are called views. To preserve liveness, a replica is the view leader for two consecutive views [Dec+22b, Appendix B]. The main message types of our implementation are NEWBLOCK and COMMITMENT. The incumbent leader proposes a batch of requests by broadcasting a NEWBLOCK message. Receiving replicas will send a COMMITMENT message back to the *next* leader. Once the next leader collected sufficient COMMITMENT messages, it is allowed to propose a new block. The original description requires a correct replica to send two messages per view to the next leader (a so-called prepare message and a so-called new view message). We omit the new view message and combine the information carried by the prepare and new view messages into a single COMMITMENT message. Replicas *always* send a COMMITMENT; if no NEWBLOCK message was received within a timeout, the replica will commit the last valid block received again. Our implementation uses the exponential increase and linear decrease as described in the original publication [Dec+22a, Section 3].

COMMITMENT messages are signed with a USIG-like module called checker which is used to implement a hybrid fault-tolerant FIFO reliable broadcast (see Section 4.2). The checker module does not only assign a unique counter value to a message; this would not suffice to ensure safety [Dec+22a, Section 4.1]. To ensure safety, the checker module also ensures that a replica cannot lie about blocks it committed in the past [Dec+22a, Section 4.2.1]. Additionally, the Chained-Damysus enclave provides an accumulator function that aggregates the received commitments for a view into a single commitment (a quorum certificate). The accumulator implements a TEE-based threshold signature scheme and prevents that a NEWBLOCK message has to contain all commitments. Thus, the NEWBLOCK message has a constant size instead of a size linear in n. Hashes that are computed outside the enclave are computed using SHA2-256 (see MinBFT description

[12] `https://github.com/vrahli/damysus` (accessed 2025-08-12)

above). The hashes of commitments have to be computed inside the enclave. As the `sha2` crate does not work inside SGX[13], we use the BLAKE2b hash function [Aum+13] provided by the `blake2` crate[14] inside the enclave. The signature is an ECDSA signature with a key size of 256 bits [Nat23a] provided by Intel's SGX SDK.

A replica in Chained-Damysus is allowed to decide a block when it was followed by two valid and consecutive blocks. To increase throughput, we add a batching mechanism similar to the one in MinBFT: After having a quorum of commitments, the leader waits for a configurable time interval. e.g., 0.1 s, to collect a configurable amount of requests, e.g., 100, before sending the NEWBLOCK message.

During the implementation, we found that the original description of Chained-Damysus lacks relevant details concerning the ordered and guaranteed delivery of messages. A faulty leader may fail in broadcasting its NEWBLOCK message proposing a block b. But when a quorum of replicas, i.e., $t + 1$ replicas, receive the message and the block is valid, they will commit it. Consequently, consecutive blocks will be built on top of block b. Blocks, however, are not reliably broadcast. Hence, a correct replica p_i may observe a timeout for a view where a quorum of replicas did not. In summary, there is no guarantee that all correct replicas will receive block b making it impossible for them to deliver any block that followed b[15]. To mitigate this problem, we introduce a backfilling mechanism that allows replicas to request missing blocks from other replicas. The HotStuff paper describes the necessity for such a procedure as well [Yin+19, Section 4.2]. When a replica receives a block for which it does not know the linked parent block or the block linked by the quorum certificate, it broadcasts a block request. To avoid that faulty replicas can inject arbitrary blocks in their responses, the requesting replica compares the hash value of a received block with the hash value received with the link information. A replica only accepts a block response if the hash values match. If a replica p_i receives a block for view v after v timed out for p_i, p_i *must not* create a COMMIT message for this block as it already created one after the timeout. If p_i would create a second COMMIT message, its USIG counter would become out of sync with the other replicas which, in turn, would drop any future message of p_i. Moreover, the implementation has to be able to handle messages for views that the replica did not reach yet. Such messages have to be held back until the replica reaches the corresponding view. We use a heap-based priority queue to manage such messages; after each view transition we check if we have stored messages for the new view.

[13] The `sha2` crate uses automated CPU feature detection to use hardware acceleration if possible. Although all CPU features are also available within the enclave, the detection mechanism does not work within SGX and renders the entire library unusable. The only known workarounds are to change the entire compilation process or to manually set CPU feature flags (see `https://github.com/RustCrypto/hashes/issues/321`, accessed 2025-08-14). We see both as too cumbersome for our research project.

[14] `https://crates.io/crates/blake2` (accessed 2025-08-13)

[15] Whenever a replica observes a gap in the chain of blocks defined by following the quorum certificates' hash links, it cannot deliver any block from a view higher than the gap.

5.3.4 NxBFT

The core of NxBFT is the TEE-Rider atomic broadcast. TEE-Rider defines the communication protocol for the replicas in the system, ensuring that messages are delivered reliably and in the correct order. NxBFT adds the NxB client model, request deduplication, and vertex scheduling logic as well as methods for setup, garbage collection, and reconfiguration. As we are interested in the common case performance, we restrict the implementation on the server side to the TEE-Rider atomic broadcast (Algorithms 6 to 8), the request deduplication, and the vertex scheduling logic (essentially as described in Section 4.6.3 and Figure 4.10)[16].

The USIG-like enclaved signing service is identical to MinBFT. At the time of writing, the TEE-based Cachin common coin exists only as a prototype. Thus, we stick with the naive TEE-based common coin and cannot investigate if the performance gain of the TEE-based Cachin coin has an impact on NxBFT's performance. We will, however, empirically compare the naive TEE-based common coin with the prototype of the TEE-based Cachin coin in Section 5.4.3. The naive common coin is implemented using a cryptographically secure pseudorandom number generator based on the ChaCha stream cipher [NL18] provided by the `rand_chacha` crate[17]. The naive common coin uses validly signed vertices as coin shares. This requires that the enclave can interpret vertex data and verify their signatures. For the latter, the enclave must compute hashes on its own for which we use BLAKE2b (see the Damysus description above).

NxBFT's vertex buffer is organized as a heap-based priority queue. The DAG is stored in a `vector` data type. Graph traversals for vertex ordering are implemented using a depth-first search. Graph traversals for validating the retrospective commit rule are implemented using a best-first search. The best-first search first explores the breadth of the DAG and changes to a depth-first search when the branch is found that is maintained by the replica that created the searched vertex.

5.3.5 Test Strategy and Implementation Quality

To ensure a correct implementation, we employ the following test strategy for the Damysus and the NxBFT implementation. Specialized or optimized data structures, e.g., vertex buffer or DAG, are unit tested. The algorithm logic is then tested in two types of integration tests as a black box. First, we use deterministic tests that run the protocol for all $n \in [3, 35] \subset \mathbb{N}$ and up to 49 rounds[18]. After each run, it is ensured that *all* replicas ended in the same

[16] Strictly speaking, this is not correct. At the time of writing, the codebase contains the setup procedure described in Section 4.6.4 and a prototype for the recovery procedure described in [LH25, Section IV-C2]. However, these will not be considered further in the following.

[17] `https://crates.io/crates/rand_chacha` (accessed 2025-08-13)

[18] In the remainder of this chapter, we use this notation to define an interval of natural numbers. We conduct experiments or tests for all numbers that lie in the interval. For example, $n \in [3, 6] \subset \mathbb{N}$ means that we consider the cases $n = 3$, $n = 4$, $n = 5$, and $n = 6$.

state. Second, we use randomized tests that run the protocol for all $n \in [3, 25] \subset \mathbb{N}$ and 25 rounds. In each randomized run, $n - (\lfloor \frac{n}{2} \rfloor + 1)$ random replicas are selected to be faulty. Faulty replicas drop received and sent messages with chances $\{0, \frac{1}{3}, \frac{2}{3}, 1\}$. Every time a message is to be added to a receive buffer, no matter if by or to a faulty replica, there is a chance $\in \{0, \frac{1}{3}, \frac{2}{3}\}$ that it is delayed. However, all messages sent by a correct replica will eventually be delivered at all correct replicas. In the case of Damysus, there is a chance of 0.25 or 0.5 that a timeout is triggered. After each run, it is ensured that *all correct* replicas delivered the same requests in the same order and that liveness for client requests is preserved. Each combination of n, drop chance, and delay chance (and timeout chance) is repeated 15 times. The randomized test strategy revealed a significant number of bugs that, in most cases, stem from message reordering, i.e., asynchrony. We are confident that the implementations are robust and do neither break safety nor liveness.

At the time of writing, the MinBFT implementation has only limited support for randomized testing. Moreover, we are aware of an unresolved bug that can occur when multiple consecutive view changes are performed. Due to the significant complexity of the view change logic and the fact that available descriptions lack significant implementation information, we were not able yet to track down the root cause of the bug. Please see Section 5.6 for a discussion.

5.3.6 Algorithm Parametrization

All three algorithm implementations have parameters that can be adapted to the needs of the deployment scenario. These are minimum (and maximum) batch sizes, timeout values for sending batches anyways, timeouts for view changes, and, for MinBFT, a checkpoint interval. Moreover, the NxB client model has a fallback timeout to contact a different replica if no response is received in time.

The algorithm parameters – that are fixed for all experiments – are as follows and brought the best performance in a manual sensitivity analysis. MinBFT requires a batch to contain at least 10k requests, Chained-Damysus and NxBFT require 100 requests. All algorithms propose at least every 0.1 s a new block. We do not limit the maximum batch size. MinBFT and Chained-Damysus have an initial view timeout value of 3 s. MinBFT has a checkpoint interval of 1000 committed batches. The fallback timeout of an NxB client is 5 s.

5.4 Impact of Implementation Design Choices on NxBFT

Before we dive into the comparison of the protocols, we investigate the performance profile of NxBFT. This analysis gives an understanding on the resource consumption and helps to explain the performance results in Section 5.5. As cryptographic operations dominate computation time, we analyze possible optimizations regarding their choice.

5.4.1 NxBFT Computation Time Breakdown

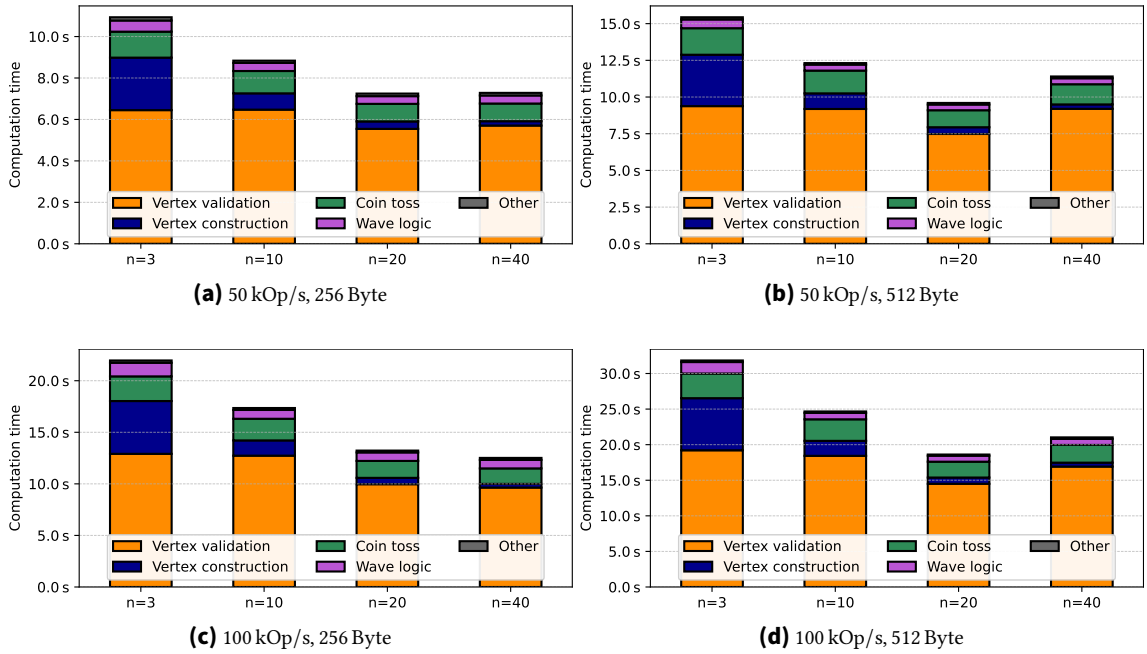
First, we analyze which protocol steps of NxBFT account for what proportion of the total computation time. To this end, we divide the NxBFT code in the following steps:

- **Client request handling:** Time spent for queuing new client requests in the request buffer (*messageBuffer* in Algorithm 6).
- **Vertex request handling:** Time spent for answering incoming vertex requests as part of the backfilling logic.
- **Vertex validation:** Time spent for verifying the validity of received vertices. This includes checking the number of edges, whether a vertex for the round by the creating replica was already received, and if it carries a valid enclave signature. For signature verification, a hash of the vertex has to be computed.
- **Graph construction:** Time spent for constructing the graph. This includes the search for and request of missing predecessors, the vertex buffer walk, and the check if a round can be completed.
- **Vertex construction:** Time spent for creating a new vertex whenever a round could be completed. This includes the creation of the vertex data structure and the signing of the vertex using the enclave.
- **Coin toss:** Time spent for performing the computeCoin(vs) function (Algorithm 8) whenever a wave is ready. This includes the context switch to the enclave, the validation of coin shares (i.e., vertices) in the enclave, and the PRNG logic.
- **Wave logic:** Time spent for executing the wave logic and reaching consensus, i.e., the waveReady(w) function in Algorithm 6 without the coin computation for w.

Please note that we only measure the time spent "inside" of the NxBFT module. Computations outside of the module which are performed by ABCperf, e.g., serialization and link management, are not measured.

We then conduct the following experiment: We set up a network of n replicas, where $n \in \{3, 10, 20, 40\}$, and 2M clients. The clients generate requests using ABCperf's no-op application with a rate of either 50 kOp/s or 100 kOp/s and requests have a size of either 256 Byte or 512 Byte. The experiment runs for 60 s and is repeated 5 times. For each run, configuration, and algorithm step, we determine the median time of all n replicas. The median times of all runs are then averaged to get the average computation time per algorithm step and configuration. Figure 5.4 shows the average computation times for all configurations. Client request handling, vertex request handling, and graph construction have a added up time of less than 0.5 s and are summarized as "Other".

We can see that vertex creation and vertex validation – which include the most cryptographic operations of NxBFT – take the most time in all configurations. With increasing n, the vertex creation time decreases, while the vertex validation time remains relatively stable. Independent from the request rate and the payload size, the vertex validation time



(a) 50 kOp/s, 256 Byte

(b) 50 kOp/s, 512 Byte

(c) 100 kOp/s, 256 Byte

(d) 100 kOp/s, 512 Byte

Figure 5.4: NxBFT computation time breakdown for $n \in \{3, 10, 20, 40\}$, request rates of 50 kOp/s and 100 kOp/s, payload sizes of 256 Byte and 512 Byte, and an experiment duration of 60 s. Average of 5 repetitions per configuration. In all cases, the validation of received vertices is the most time-consuming operation; the required time is primarily determined by number and size of requests. The vertex construction time decreases when n is increasing. The time spent on coin computation and wave logic/consensus is more or less independent from the system load. In particular, the impact of the graph operations required for the wave logic is negligible.

is roughly n times the vertex creation time. When doubling the payload size, the vertex validation time increases by a factor of ~ 1.5. The observed slowdown allows to conclude that the signature verification accounts for approximately 50% and hash calculation accounts for the other 50% of vertex validation time when the payload size is 256 Byte. When the system load increases, either by increasing the number of requests or the payload size, the total computation time of NxBFT increases from ~ 11 s ($n = 3$, request rate 50 kOp/s, payload size 256 Byte, Figure 5.4a) to ~ 32 s ($n = 3$, request rate 100 kOp/s, payload size 512 Byte, Figure 5.4d). The time spent on coin computation and wave logic/consensus is more or less independent from the system load. In particular, the impact of the graph operations required for the wave logic is negligible.

The results show that the overall computation time is primarily determined by the vertex creation and validation times. Because request rate and payload size independently increase the time of said protocol steps significantly, we can safely deduce that the time is primarily determined by the number of vertices and the number of bytes to process. From this observation, it can be concluded that hash and elliptic curve operations determine the operating costs of NxBFT. The decreasing vertex creation time for increasing n shows the positive load-balancing impact of the NxB client model: a single replica receives fewer client requests making the vertices smaller. The stable vertex validation time shows that

the overall "hash work" is not influenced by the load balancing effect of the NxB client model.

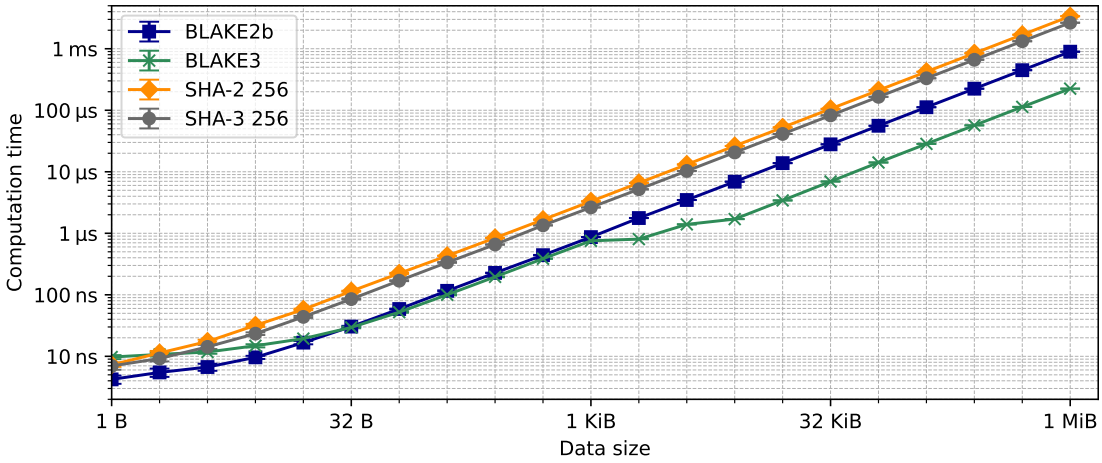
5.4.2 Hash Function

Since hashing lies on the performance-critical path, we investigate how common hash functions perform on the CPUs of the replica nodes of our experiment cluster (Intel Xeon E-2288G, released 2019; Intel Xeon E-2388G, released 2021) and on a recent AMD workstation and server CPU (AMD EPYC 4564P, released 2024). Those hash functions are SHA2-256 [Nat15a] (crate `sha2`[19]), SHA3-256 [Nat15b] (crate `sha3`[20]), BLAKE2b [Aum+13] (crate `blake2`[21]), and BLAKE3 [OCo+21] (crate `blake3`[22]). Except for BLAKE2b, all functions produce a hash value of 256 bit length. BLAKE2b is chosen as it is optimized for 64 bit infrastructures and outperforms BLAKE2s on such infrastructures. We conduct the following experiment: We create an array of 1000 random byte strings with sizes of 2^i Byte, $i \in [0, 20] \subset \mathbb{N}$. For each size, we measure the time it takes to hash all 1000 values. The experiment is repeated 10 times. Figure 5.5 shows the average hash time for each hash function and CPU.

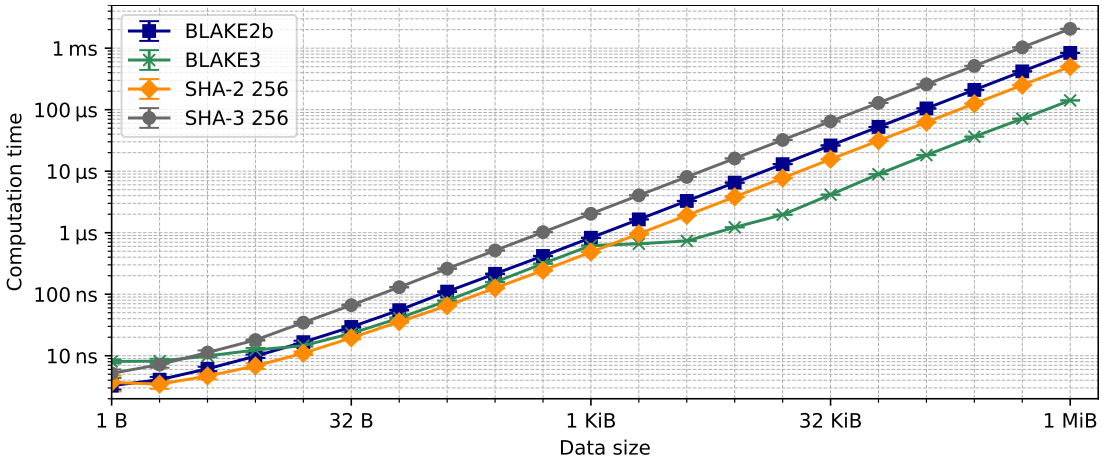
The CPUs Intel Xeon E-2388G and AMD EPYC 4565P support hardware acceleration for SHA2-256 making SHA2 the fastest hash function on these CPUs for input sizes up to 1 KiB. The experiment for the AMD CPU shows that recent hardware does not change the relative speed-up of the SHA2-256 hardware acceleration. Without hardware acceleration, however, SHA2-256 is surprisingly the slowest hash function (Figure 5.5a). BLAKE3 is for small input sizes significantly slower than BLAKE2b (slowdown $\sim 3\times$). For data sizes starting from 2 KiB, BLAKE3 outperforms all other hash functions (speedup $\sim 3\times$).

We implemented hashing in NxBFT rather naively. Whenever a hash of a data structure, e.g., of a vertex has to be computed, the hash function is invoked for every data structure field. If fields are heap containers, e.g., a vector, the hash function is invoked for every element, e.g., a client request, of the container. Memory operations are typically very fast, but hashing is a CPU-intensive operation. Thus, hashing approaches that minimize the number of hash function invocations and hash larger byte slices at once may be beneficial. For the following example, we look at the CPU Intel Xeon E-2388G. For a vertex payload size of 8 requests with a size of 256 Byte each, BLAKE3 would require 660 ns to compute the hash. Eight invocations of the hardware-accelerated SHA2-256 would require $8 \cdot 126\,\text{ns} = 1008\,\text{ns}$. This is a slowdown of $\sim 1.53\times$ which increases for bigger vertices. Moreover, as described in Section 5.3.4, NxBFT uses two different hash functions: SHA2-256 and BLAKE2b. When the enclave has to be able to compute a hash value, BLAKE2b is used because the `sha2` crate does not work inside SGX. We see that, except for the CPU Intel E-2288G, BLAKE2b is always slower than SHA2-256 and, for all

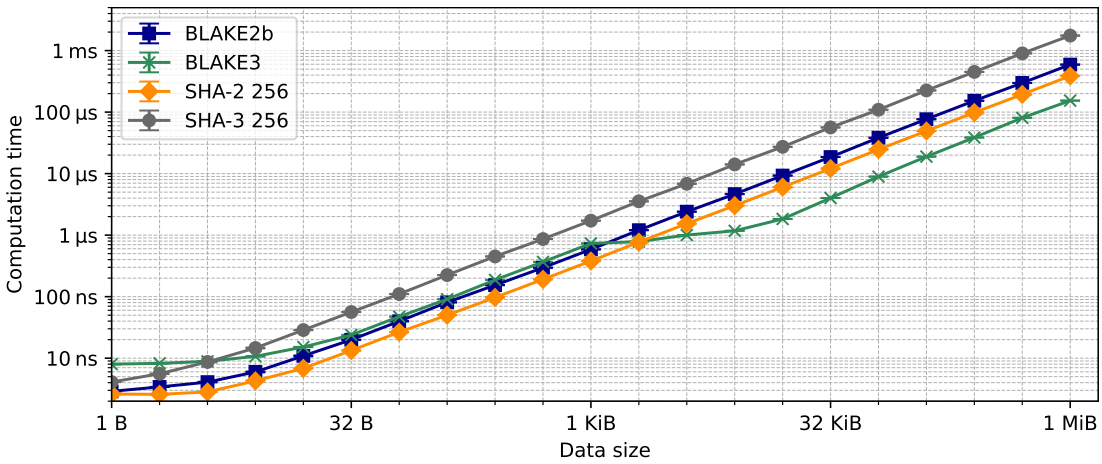
[19] `https://crates.io/crates/sha2`, accessed 2025-08-14
[20] `https://crates.io/crates/sha3`, accessed 2025-08-14
[21] `https://crates.io/crates/blake2`, accessed 2025-08-14
[22] `https://crates.io/crates/blake3`, accessed 2025-08-14



(a) Intel Xeon E-2288G



(b) Intel Xeon E-2388G



(c) AMD EPYC 4564P

Figure 5.5: Hash function benchmark for BLAKE2b, BLAKE3, SHA-2 256, and SHA-3 256 on CPUs Intel Xeon E-2288G, Intel Xeon E-2388G, and AMD EPYC 4564P. Average of 10 runs, each run consists of 1000 hash operations. Error bars indicate the 95% confidence interval. When hardware acceleration for SHA2 is available (Intel Xeon E-2388G, AMD EPYC 4564P), SHA2 outperforms all functions up to a data size of 1 KiB. Starting from 2 KiB, BLAKE3's parallelization is in clear advantage.

CPUs, significantly slower than BLAKE3. In conclusion, if the overhead of creating larger byte slices is acceptable, the switch to BLAKE3 must be considered.

5.4.3 Common Coin

In its current implementation, NxBFT uses the naive TEE-based coin. The NxBFT computation time analysis above showed that the common coin's COMPUTECOIN(·) phase takes between 10% and 12% of the total computation time. The analysis in Section 4.3.3 (see Table 4.1 in particular) showed that the TEE-based Cachin coin requires $\sim 2\times$ less elliptic curve operations for computing the coin value than the naive TEE-based coin. Thus, we can expect a speedup of $2\times$ for this phase when using the TEE-based Cachin coin instead of the naive TEE-based coin. When ignoring context switches to the enclave and considering coin share generation and validation as well, the speedup of the coin computation phase enables an overall expected speedup of $1.33\times$ for the common case and $1.2\times$ for the worst case: In the common case, the first $t + 1$ received coin shares are valid, leading to a total of $1 + 2(t + 1) + 2t + 2 = 5 + 4t$ multiplications for the naive TEE-based coin and $2 + 2(t + 1) + t + 1 = 5 + 3t$ multiplications for the TEE-based Cachin coin (see Table 4.1). For $t \to \infty$, this results in speedup of $1.33\times$. In the worst case, all n coin shares have to be verified, leading to a total of $1 + 2(2t + 1) + 2t + 2 = 5 + 6t$ multiplications for the naive TEE-based coin and $2 + 2(2t + 1) + t + 1 = 5 + 5t$ multiplications for the TEE-based Cachin coin. For $t \to \infty$, this results in speedup of $1.2\times$.

We conduct an empirical comparison of the native Cachin coin, the naive TEE-based coin and the TEE-based Cachin coin to determine if the expected performance gain in the common case is worth the way more complex implementation. The experiment was conducted as part of the master's thesis by Marius Haller [Hal25]. All coins are implemented in C++; the Intel SGX SDK[23] including its variant of the Intel Cryptography Primitives Library is used to implement the enclave logic and the cryptographic operations (this includes those operations outside the enclave as well). The elliptic curve used is NIST P-256 [Che+23]. We set up a peer-to-peer network of $n = 2t + 1, t = 2^i, i \in [0, 6] \subset \mathbb{N}$, processes. The processes are equally distributed among the 20 replica nodes of our experiment cluster; the network round trip latency is 0.15 ms. Each network performs 600 coin tosses for each coin implementation. The experiment ensures that all processes simultaneously start the coin share generation and distribution. We measure the time it takes at each process to generate the coin share, distribute the shares over the network, validate the first $t + 1$ coin shares, and compute the coin value.

Figure 5.6 shows the average time of all 600 runs for each configuration. First, we see a clear order: In *all* configurations, the naive TEE-based coin is faster than the native Cachin coin and the TEE-based Cachin coin is faster than the naive TEE-based coin. The naive TEE-based coin shows a median speedup of $1.42\times$ over the native Cachin coin. The TEE-based Cachin coin shows a median speedup of $1.22\times$ over the naive TEE-based coin

[23] `https://github.com/intel/linux-sgx` (accessed 2025-08-14)

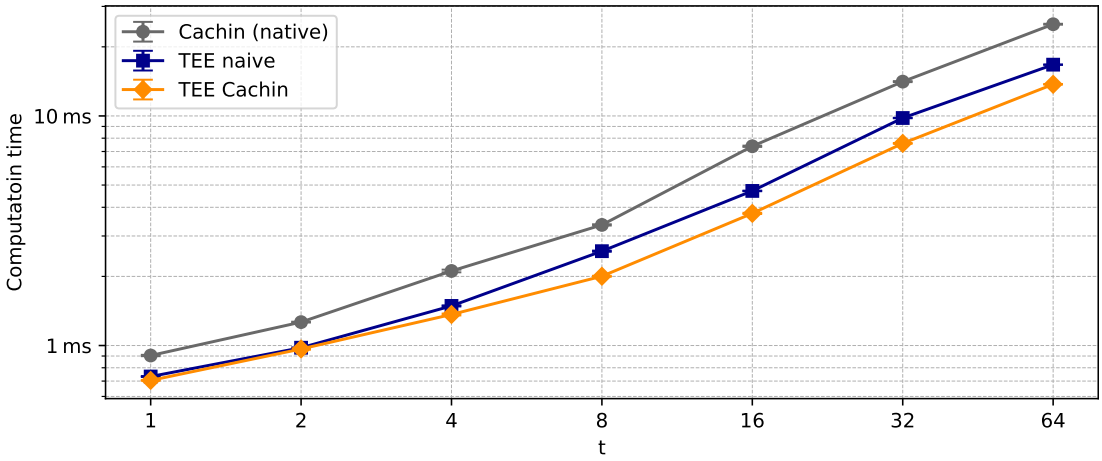


Figure 5.6: Comparison of the *total time* of the native Cachin coin, the naive TEE-based coin, and the TEE-based Cachin coin for $t = 2^i, i \in [0, 6] \subset \mathbb{N}, n = 2t + 1$ [Hal25]. Average of 600 repetitions per configuration. Error bars indicate the 95% confidence interval. The native Cachin coin is the slowest in all cases. The TEE-based Cachin coin outperforms the two other implementations in all cases. The median speedup of the TEE-based Cachin coin in comparison to the naive TEE-based coin is 1.22×.

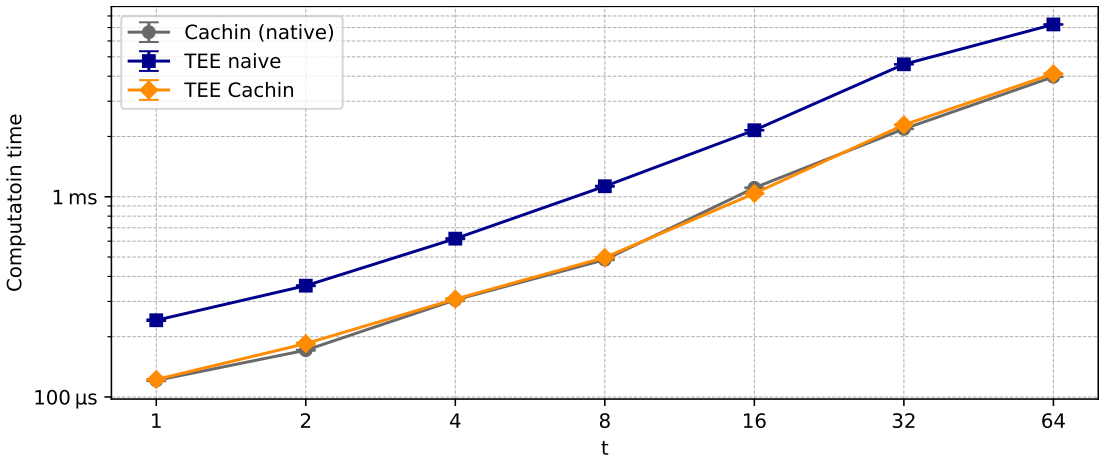


Figure 5.7: Comparison of the *coin computation phase time* of the native Cachin coin, the naive TEE-based coin, and the TEE-based Cachin coin for $t = 2^i, i \in [0, 6] \subset \mathbb{N}, n = 2t + 1$ [Hal25]. Average of 600 repetitions per configuration. Error bars indicate the 95% confidence interval. The native Cachin coin and the TEE-based Cachin have the identical algorithm in their coin computation phases. The naive TEE-based coin computes the coin value inside the TEE which requires to validate every coin share inside the TEE again making it roughly 2× slower than the other two implementations.

and a median speedup of 1.67× over the native Cachin coin. Moreover, we can see that the network communication overhead of our setup is negligible. The fastest implementation, the TEE-based Cachin coin, takes ~ 0.733 ms for $t = 1$. Even if it would take significantly longer than half a network round trip time (~ 0.075 ms) to distribute the coin shares (which is not the case), the overall time would still be dominated by coin share generation, validation, and coin computation.

Figure 5.7 shows the average time of the coin computation phase for each configuration and implementation. We can observe that the speedup of the TEE-based Cachin coin over

the naive TEE-based coin described above primarily stems from the way more efficient coin computation phase. The median speedup for this phase is 2×. Since the TEE-based Cachin coin and the native Cachin coin have the exact same coin computation algorithm, they take the same time. In this analysis, it appears that the cost of the context switch required by the naive TEE-based coin is negligible in comparison to the elliptic curve multiplications. This observation, however, does not generalize. We observe, in line with related work [WAK18], that the cost of the context switch depends on the amount of computation done before and after the context switch to and from the enclave and on the amount of data managed by the enclave. For the coin share generation phase, for example, the cost of the context switch is roughly 1.5× the cost of an elliptic curve multiplication.

The overall speedup of 1.22× lies within the expectation based on the analytical assessment above. In the computation time breakdown above, the coin computation phase took between 10.9% and 12.6% of the total computation time. We showed in Section 4.3.3 how to combine the USIG and the TEE-based Cachin coin without additional overhead. If we now simplify the share of the coin computation to 10% and follow Amdahl's law, a speedup of computeCoin(·) of 2× would lead to an overall speedup of 1.05× for NxBFT.

5.4.4 Concluding Remarks

The analysis of the NxBFT implementation shows that the overall computation time is primarily determined by the different cryptographic operations: hashing, elliptic curve computations, and the common coin which is, again, primarily determined by elliptic curve computations. While we will see below that the current NxBFT implementation performs very well, the analysis in this section shows that there is still room for improvement. For real-world deployments, (1) the hash computation logic has to be optimized to reduce the number of hash function invocations and (2) the common coin should be based on the TEE-based Cachin coin. Moreover, switching to an Edwards-based curve may improve the performance of the elliptic curve operations [BL07]. While these changes may have no impact on observed end-to-end latencies for low and mid-range throughput scenarios, every saved millisecond of computation certainly increases the resilience of the system as the impact of load fluctuations and short overload scenarios is lowered.

Most importantly, the analysis shows (3) that parallelization is indispensable when aiming for high throughput and low latency: Between 60% and 80% percent of computation time is spent for vertex validation. A parallelization using n worker threads that perform the vertex validation, which primarily is hash computation and elliptic curve operations, may show a speedup for the vertex validation phase of up to n×. According to Amdahl's law, such a speedup already leads to an overall speedup between 2.17× and 3.57× at $n = 10$. Combined with the TEE-based Cachin coin, we could achieve a theoretical speedup between 2.43× and 4.35× for $n = 10$.

	MinBFT		Chained-Damysus		NxBFT	
	BFT	NxB	BFT	NxB	BFT	NxB
$n = 3$	**125** 104 ± 3	**100** 169 ± 0	**200** 24 ± 0	**600** 77 ± 2	**200** 14 ± 0	**800** 39 ± 1
$n = 10$	**50** 97 ± 0	**75** 144 ± 0	**50** 12 ± 0	**800** 469 ± 2	**50** 185 ± 0	**1000** 42 ± 5
$n = 20$	—	**75** 148 ± 0	—	**600** 587 ± 38	—	**1000** 55 ± 2
$n = 40$	—	**50** 161 ± 46	—	**200** 348 ± 3	—	**1000** 146 ± 8

Table 5.2: Maximum sustained throughput (in kOp/s, bold font) and corresponding end-to-end latency (in ms, normal font, average of 5 runs with 95% confidence interval) in dependence on the client model (BFT or NxB) for request rates ∈ {25, 50, 75, 100, 125, 150, 175, 200, 400, 600, 800, 1000} kOp/s, 0 B payload, and 0.15 ms network round trip latency. For $n \geq 20$ and the BFT client model, none of the algorithms was capable to show a stable system state for the request rates tested which is caused by the single thread handling client requests instead of generating protocol messages. All algorithms benefit from the load reduction of the NxB client model. NxBFT shows the strength of the DAG-based approach that processes n proposals in parallel. Figure 5.8 shows the detailed end-to-end latencies of the NxB model; Figure 5.9 compares the end-to-end latencies for the two client models for $n = 10$.

5.5 Comparison of MinBFT, Damysus, and NxBFT

In this section, we analyze and compare the performance of MinBFT, Chained-Damysus, and NxBFT in a set of experiments with different foci: impact of the client model (BFT vs. NxB), impact of the payload size, impact of network size and network latency, and impact of faults. In all experiments, the algorithms are parametrized as described above (see Section 5.3.6). We use the no-op application of ABCperf. The maximum number of clients is set to 2M.

5.5.1 Impact of the Client Model: BFT vs. NxB

To evaluate the impact of the client model and the algorithms' scaling behavior, we run NxBFT, Chained-Damysus, and MinBFT for request rates ∈ {25, 50, 75, 100, 125, 150, 175, 200, 400, 600, 800, 1000} kOp/s and for $n \in \{3, 10, 20, 40\}$ replicas with the BFT and the NxB client model. We run each configuration five times for 60 s. To prevent measuring the effects of network speed and cryptographic operations and, instead, investigate the protocols' overhead, the requests have a zero byte payload and no additional network latency is emulated.

Table 5.2 shows the maximum request rates for which a stable system state was observed. Using the NxB client model, MinBFT achieves its peak performance for $n = 3$. Chained-Damysus' and NxBFT's throughput peaks at $n = 10$. When increasing n to 20, the throughput of Chained-Damysus decreases by 25%. NxBFT can sustain a throughput of 1000 kOp/s for $n \in \{10, 20, 40\}$. The table shows the maximum request rates for the BFT client as well: None of the algorithms can sustain a throughput of 25 kOp/s for $n \geq 20$. In comparison to the NxB client model, MinBFT achieves a slightly higher throughput for

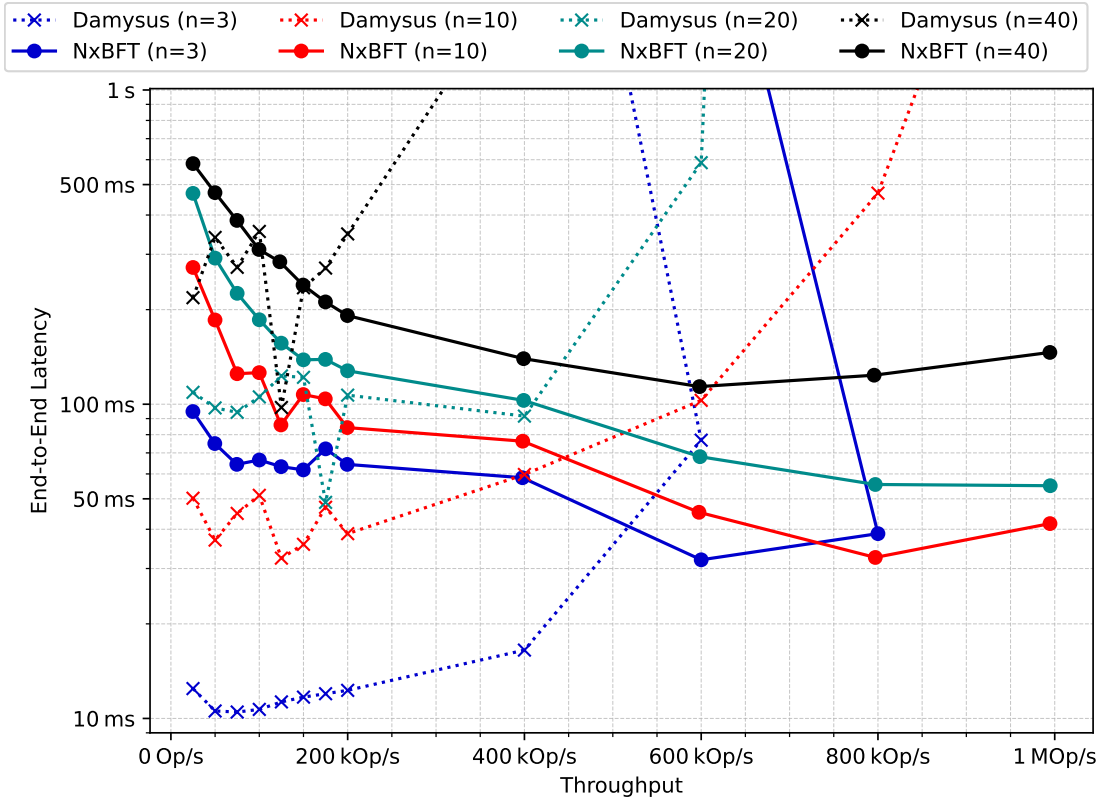


Figure 5.8: End-to-end-latency of Chained-Damysus and NxBFT for $n \in \{3, 10, 20, 40\}$, request rates $\in \{25, 50, 75, 100, 125, 150, 175, 200, 400, 600, 800, 1000\}$ kOp/s, 0 Byte payload, and 0.15 ms network round trip latency. Both algorithms use the NxB client model. To increase readability, confidence intervals are not plotted and the y-axis is limited. For $n < 40$, Damysus shows the benefit of the linear communication complexity. NxBFT's latency benefits from increased request rates. Maximum sustained throughput is summarized in Table 5.2.

$n = 3$ and a slightly lower throughput for $n = 10$. Chained-Damysus achieves a third of the NxB client model throughput for $n = 3$ and only 6.25% for $n = 10$. NxBFT achieves the same throughput as Damysus when using a BFT client model. These are, however, only 25% and 5% respectively of the NxB client model throughput.

Table 5.2 also shows the corresponding end-to-end latencies if one of the tested request rates stabilized. It is noticeable that in four of the six columns, lower end-to-end latencies can be observed in some cases as the number of replicas increases. Moreover, we can see that Chained-Damysus achieves significant higher end-to-end latencies when using the NxB client model. To get more insight, we also plot the end-to-end latencies for the NxB client model runs of Chained-Damysus and NxBFT in Figure 5.8 and we compare the latencies of the NxB client model and the BFT client model in Figure 5.9.

Figure 5.8 clearly shows for NxBFT and Chained-Damysus, that the latency increases when increasing the number of replicas. NxBFT achieves the best latency for $n = 3$ and a request rate of 600 kOp/s with $\sim$ 0.03 s and stays for all configurations, before saturating, below 0.6 s. Chained-Damysus achieves the best latency for $n = 3$ and a request rate of 50 kOp/s with $\sim$ 0.01 s. The zig-zag pattern of Chained-Damysus for request rates $<$ 200 kOp/s is

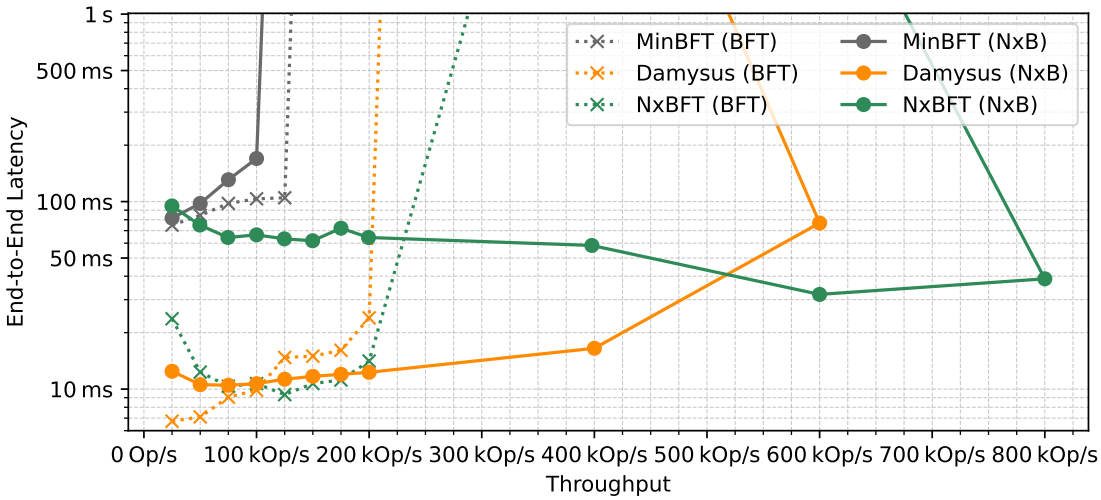


Figure 5.9: End-to-end-latency of MinBFT, Chained-Damysus, and NxBFT for the BFT and the NxB client model, $n = 3$, request rates $\in \{25, 50, 75, 100, 125, 150, 175, 200, 400, 600, 800, 1000\}$ kOp/s, 0 Byte payload size, and 0.15 ms network round trip latency. To increase readability, confidence intervals are not plotted and the y-axis is limited. All algorithms achieve lower latencies with the BFT client model. For request rates ≤ 400 kOp/s, Damysus outperforms MinBFT and NxBFT. While the NxB client model allows the algorithms to scale throughput, it has a clear latency penalty.

not caused by the number of five runs being not sufficient; the median confidence interval in this area has a relative size of 2%. Instead, we assume synchronization effects between batching logic and request rate to be the cause.

All algorithms achieve a significant speedup when using the BFT client model. MinBFT achieves the best latency for $n = 3$ and 25 kOp/s with ~ 0.07 s. Chained-Damysus achieves the best latency for $n = 3$ and 25 kOp/s with ~ 0.007 s. NxBFT achieves the best latency for $n = 3$ and 125 kOp/s with ~ 0.009 s. Figure 5.9 exemplarily compares the achieved end-to-end latencies of the two client models under investigation for $n = 3$.

In summary, we can observe that the NxB client model allows for load balancing when having a rotating leader or no leader at all but has a latency penalty. While in MinBFT the current leader has to handle *every* client connection, Chained-Damysus and NxBFT replicas handle, in expectation, $\frac{1}{n}$th of all client connections. Nonetheless, the NxB client model allows MinBFT to achieve higher throughput for $n \geq 20$ and, for the BFT client and $n \geq 20$, no algorithm was able to achieve a stable system state. This is caused by the fact that the BFT client model requires all replicas to process client requests, leading to increased contention and reduced throughput at the single thread that executes the SMR logic. In all algorithms, this leads the replicas to produce protocol messages, i.e., commit messages in MinBFT and Chained-Damysus, and vertices in NxBFT, at significantly reduced rates. A different scheduling strategy, e.g., prioritizing incoming protocol messages over incoming client requests, may mitigate this effect. We assume that the latency for the combination of NxB client model and MinBFT is increased because in the NxB client model the response must always come from the leader. In the BFT client model, other replicas – that have significantly less load than the leader – can also respond. Furthermore, we suspect that the BFT client model provides synchronization effects that enable higher throughput,

especially for $n = 3$. The latter statement in particular is a hypothesis that was not the focus of further investigation in this dissertation.

Commit messages in MinBFT and Chained-Damysus are sent by the replica to the leader and only contain the sending replica's approval as information of value. In NxBFT, however, *every* message, i.e., vertex, exchanged carries client requests *and* votes on previous vertices. Thus, while the workload in terms of messages exchanged and, thus, cryptographic operations performed increases when increasing n, NxBFT outperforms Chained-Damysus in terms of throughput significantly. The reduced connection load allows Chained-Damysus to scale as well, but for $n > 10$ the increasing work to be done by the atomic broadcast eliminates those benefits without producing as much value as it is the case for NxBFT. However, due to the random selection of a replica by a NxB client, requests have to wait longer for block inclusion. Especially Chained-Damysus suffers significantly, as, in the worst case, a request has to wait $2n$ blocks for inclusion since Chained-Damysus requires a replica to be in charge for two blocks [Dec+22b, App. B]. This maximum waiting time is reached when the client selected the replica which just proposed its second block.

For NxBFT, we observe that the replica workers have on average $\sim 25\%$ total CPU usage but the process executing the SMR logic utilizes one CPU core permanently to the maximum. This is another indication that for high load the single-threaded SMR logic is a bottleneck. As discussed above, suitable parallelization at each replica may improve scaling behavior even further.

The positive correlation between the number of replicas and end-to-end latency clearly shows that the decreasing end-to-end latencies in Table 5.2 give a misleading impression. Rather, it is that the request rates we chose are not fine-grained enough to determine the true maximum sustained throughput. While not being suited as an estimate for real-world performance, the identified maximum throughput and corresponding latencies are still a valuable metric for understanding the algorithms' performance characteristics with regard to the choice of client model.

5.5.2 Impact of Payload Size

The analysis of the NxBFT computation time revealed that cryptographic operations have a significant impact on the performance. Thus, the payload size of a request will increase the computation time. Moreover, increased payload will increase the network transmission time as well. As a result, the maximum sustained throughput will decrease with increasing payload sizes. To estimate the impact of payload size, we conducted experiments with varying payload sizes and measured the corresponding throughput and end-to-end-latencies. We run MinBFT, Chained-Damysus, and NxBFT for request rates $\in \{25, 50, 75, 100, 125, 150, 175, 200, 400, 600, 800, 1000\}$ kOp/s and $n = 10$ replicas with the NxB client model. We use payload sizes of 0 Byte, 256 Byte as, e.g., in [Dec+22a; Dec+24], and 512 Byte as, e.g., in [Dan+22; Spi+22; BTR24; Gir+24]. We estimate that the size of a check-in request in the MaaS application (see Chapter 3) is no larger than 150 Byte, which is why we consider the payload sizes used in the experiments to be reasonable for

	MinBFT	Chained-Damysus	NxBFT
0 Byte	**75** 144 ± 0	**800** 469 ± 2	**1000** 42 ± 5
256 Byte	**25** 299 ± 0	**150** 643 ± 15	**600** 214 ± 4
512 Byte	—	**75** 285 ± 3	**200** 92 ± 3

Table 5.3: Maximum sustained throughput (in kOp/s, bold font) and corresponding end-to-end latency (in ms, normal font, average of 5 runs with 95% confidence interval) for $n = 10$ replicas in dependence on payload size (0 Byte, 256 Byte, 512 Byte) for request rates $\in \{25, 50, 75, 100, 125, 150, 175, 200, 400, 600, 800, 1000\}$ kOp/s, 0.15 ms network round trip latency, and the NxB client model. For 512 Byte, the throughput of all algorithms drops significantly; MinBFT does not show a stable system state. NxBFT outperforms Damysus in terms of throughput and end-to-end latency. Figure 5.10 shows the detailed end-to-end latencies of the experiment.

the application categories we are interested in. We run each configuration five times for 60 s.

Table 5.3 shows the maximum sustained throughput rates with the corresponding end-to-end-latencies for the different payload sizes. In the experiments with 256 Byte, MinBFT's throughput drops to a third, i.e., 25 kOp/s. Chained-Damysus' throughput drops to 150 kOp/s (18.75% of the 0 Byte throughput) and NxBFT's throughput drops to 600 kOp/s (60% of the 0 Byte throughput). In the experiments with 512 Byte, MinBFT does not show a stable system state for the request rates tested. Chained-Damysus' throughput drops to 75 kOp/s (50% of the 256 Byte throughput) and NxBFT's throughput drops to 200 kOp/s (a third of the 256 Byte throughput).

Figure 5.10 shows the detailed results. MinBFT's and Chained-Damysus' end-to-end latencies increase significantly with increasing payload size. NxBFT's end-to-end latency, however, remains relatively stable. In Section 5.4.1, we already saw that the latency of NxBFT is dominated by signature creation and verification. For the payload sizes investigated, hashing is relevant but not dominating. Moreover, NxBFT's wave structure increases the latency even more as only every four rounds a decision can be made. This decision, however, can only address proposals from vertices that are at least four rounds old. Hence, NxBFT has a significantly increased base latency. We can see that the increased base latency of NxBFT hides the latency impact of the additional hashing required because of the increased payload size.

5.5.3 Impact of Network Size and Network Latency

Graph-based and leader-rotating protocols are known for being highly dependent on the network latency between replicas. We investigate the impact of deployment properties by using random byte payloads of size 256 B and adding emulated network round trip latencies of 0 ms (same datacenter), 5 ms (same country), 35 ms (Europe), and 150 ms (World) to the physical datacenter round trip latency (~ 0.15 ms). We use the NxB client model. For each configuration and in a total of ~ 2.5k experiment runs, we identify the maximum

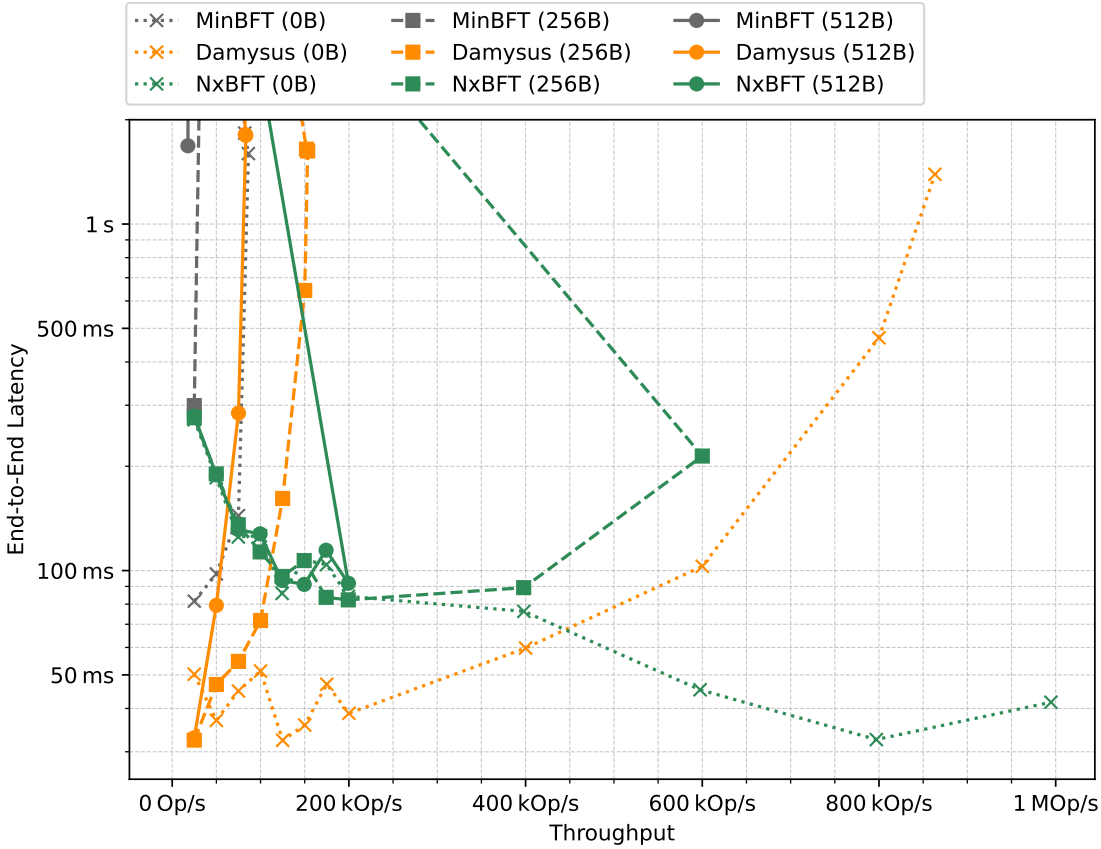


Figure 5.10: End-to-end-latency of MinBFT, Chained-Damysus, and NxBFT for 0 Byte, 256 Byte, and 512 Byte payload size, $n = 10$ replicas, request rates $\in \{25, 50, 75, 100, 125, 150, 175, 200, 400, 600, 800, 1000\}$ kOp/s, 0.15 ms network round trip latency, and the NxB client model. To increase readability, confidence intervals are not plotted and the y-axis is limited. MinBFT's and Damysus' latencies increase and the maximum throughput decreases when the payload size increases. NxBFT's latency is roughly the same but the throughput drops to a fifth.

request rate that saturates the system but does not cause overload: After experiment start, the latency emulation is started and the system is stressed with a fixed request rate. We measure the achieved throughput for 240 s. We follow a rather conservative rejection strategy: If request rate and throughput do not match or the system shows any burst or backlog patterns, the request rate is not accepted. Please note that except for Chained-Damysus, a latency of 150 ms, and $n \geq 20$, we identify the maximum request rate with a granularity of at most 1 kOp/s. Additionally, we measure the intermediate decision time (IDT, see Section 5.2.3). We measure the IDT instead of the end-to-end latency because the IDT is more independent from the client model being used; it does not reflect the time a client has spent waiting. In MinBFT, a replica is able to decide whenever it collects $t + 1$ commits for a block (w/o faults one network round trip in expectation). In Chained-Damysus, a replica is able to decide a block when it is followed by two valid and consecutive blocks (w/o faults one network round trip in expectation). In NxBFT, a replica is able to decide when it finished a wave, i.e., it finished four consecutive broadcast rounds and the wave root selected by the common coin is part of the local DAG (w/o faults two network round trips in expectation). Table 5.4 lists the maximum request rates in

Round Trip Latency	MinBFT				Chained-Damysus				NxBFT			
	$n=3$	$n=10$	$n=20$	$n=40$	$n=3$	$n=10$	$n=20$	$n=40$	$n=3$	$n=10$	$n=20$	$n=40$
0.15 ms	**69**	**28**	**18**	**11**	**223**	**132**	**64**	**36**	**335**	**503**	**276**	**153**
	124	253	129	137	7	41	65	46	21	69	81	279
5 ms	**63**	**25**	**17**	**11**	**120**	**64**	**56**	**31**	**304**	**234**	**191**	**99**
	121	184	143	149	28	69	177	123	69	247	269	486
35 ms	**49**	**20**	**13**	**11**	**29**	**20**	**14**	**3**	**91**	**55**	**45**	**28**
	120	121	126	153	150	791	539	323	352	680	1180	1404
150 ms	**33**	**15**	**12**	**9**	**6**	**4**	**1.2**	**0.2**	**45**	**26**	**24**	**20**
	111	117	123	137	1641	1580	1218	766	1299	1924	1625	1680

Table 5.4: Maximum sustained throughput (in kOp/s, bold font) and corresponding intermediate decision times (in ms, normal font, average of 30 runs, confidence intervals left out to increase readability) depending on network size and network latency with 256 B payload size and the NxB client model. MinBFT shows the best latencies and reasonable throughput for big networks with high latencies. Damysus shows the best latencies for small networks that are operated with datacenter network latencies. NxBFT shows the most throughput in all configurations investigated.

bold font and kOp/s and the average IDT of 30 runs in normal font and milliseconds. IDT confidence intervals are all below 9% and left out.

MinBFT's throughput peaks for $n = 3$ and no emulated latency with 69 kOp/s and an IDT of 124 ms. MinBFT achieves the lowest throughput for $n = 40$ and a network latency of 150 ms. Chained-Damysus peaks for $n = 3$ and no latency with 223 kOp/s and an impressive IDT as small as 7 ms. For $n = 40$ and a network latency of 150 ms, the throughput drops to 0.2 kOp/s. The IDT increases with network latency and n. NxBFT shows peak performance for $n = 10$ and no latency with 503 kOp/s and 69 ms IDT. For network round trip latencies ≥ 5 ms, the throughput peaks for $n = 3$. For $n = 40$ and 150 ms network round trip latency, the throughput drops to 20 kOp/s. The IDT increases with network latency and n.

Although MinBFT requires at least one network round trip time between two decisions (two all-to-all broadcasts), MinBFT can stay below 150 ms IDT for 150 ms network round trip latency. This is due to MinBFT's sliding-window mechanism: the leader proposes a new block whenever it has sufficient requests. Thus, as long as the system is not overloaded, MinBFT can benefit from ABCperf's parallelized network stack. Chained-Damysus and NxBFT, however, need quorums – which act as synchronization barriers – before being able to continue. The throughput of MinBFT, however, cannot benefit: The network latency determines the minimum time a request has to wait for decision as the leader is required to collect a commit quorum. Put differently, average IDTs can be below the network latency because due to the sliding window blocks are decided in batches. The end-to-end latency, however, cannot be below the network latency.

For network round trip latencies ≤ 5 ms, Chained-Damysus can take full advantage of the streamlined design, achieving IDTs clearly faster than MinBFT and NxBFT. As observed before (see Section 5.5.1 and Figure 5.9), Chained-Damysus severely suffers from the NxB client model when having big n. Increasing the block creation rate, e.g., by lowering the minimum block size, may improve throughput in such cases. NxBFT shows the benefit of

the NxB client model in terms of load balancing, significantly outperforming throughput in all configurations. While, for experiments with no payload, the load balancing allowed for higher throughput with increased n, both Chained-Damysus and NxBFT loose this ability. NxBFT, however, keeps the ability for no added latency. In fact, the load balancing increases the throughput but the increased cryptographic work (see Section 5.5.2 above) masks the observed effect.

It is noticeable that, compared to Table 5.3, MinBFT achieves a slightly higher throughput, while both Chained-Damysus and NxBFT achieve a significantly lower throughput for $n = 10$ replicas and 0.15 ms network round trip latency. It can be assumed that the 60 seconds of observation time in the experiment in Section 5.5.2 are not sufficient to stabilize the system near the load limit, and that five repetitions of the experiment are not sufficient to account for the variance in the system, even though the confidence intervals are convincing. However, since only the effect of different payload sizes on the algorithm behavior was to be investigated, and no estimate of throughput under real conditions was sought (which is the focus of this section), and since the algorithms differ significantly in the values determined, the choice of parameters in Section 5.5.2 is justified. Moreover, for Chained-Damysus and NxBFT, we observe that the IDT may decrease when increasing n, which indicates that the true maximum throughput may be higher (see also Section 5.5.1 above). First, our step size may be too big. Second, larger networks may be more sensitive to scheduling and transmission variance when operating close to maximum performance. Third, our rejection strategy may be too conservative or the experiment duration may not be sufficient: While sometimes an overload was only detected within the fourth minute of the experiment, it could be that the system would have stabilized again after a few more minutes. Nonetheless, the maximum throughput rates listed in Table 5.4 are a good estimate of the maximum sustained throughput for the configurations tested.

5.5.4 Impact of Crash Faults

We investigate the algorithms' performance under faults for $n = 10$, a request rate of 50 kOp/s 0 Byte payloads using the NxB client model, and no emulated latency. After 20 s, replica 0 crashes, after 80 s, replica 3 crashes, and after 140 s, replica 9 crashes. We selected those replicas to crash as this pattern allows Chained-Damysus to make the most progress. To prevent unlimited increase of Damysus' timeouts caused by the exponential increase and linear decrease mechanism, we do not crash four replicas. Figure 5.11 shows the average end-to-end latency of the experiment for ten repetitions.

In case of MinBFT, replica 0 is the current leader and MinBFT is forced to perform a view-change. The throughput of MinBFT stalls until the inactivity of the leader is detected and the view-change was successful. The new leader is not capable to cut the backlog down, yielding increased latencies for the remainder of the experiment (a metastable failure state [Hua+22; LD24]). The crash of non-leader replicas (3 and 9) has no effect. For lower request rates, MinBFT would be capable of recovering fast response times within the experiment time window.

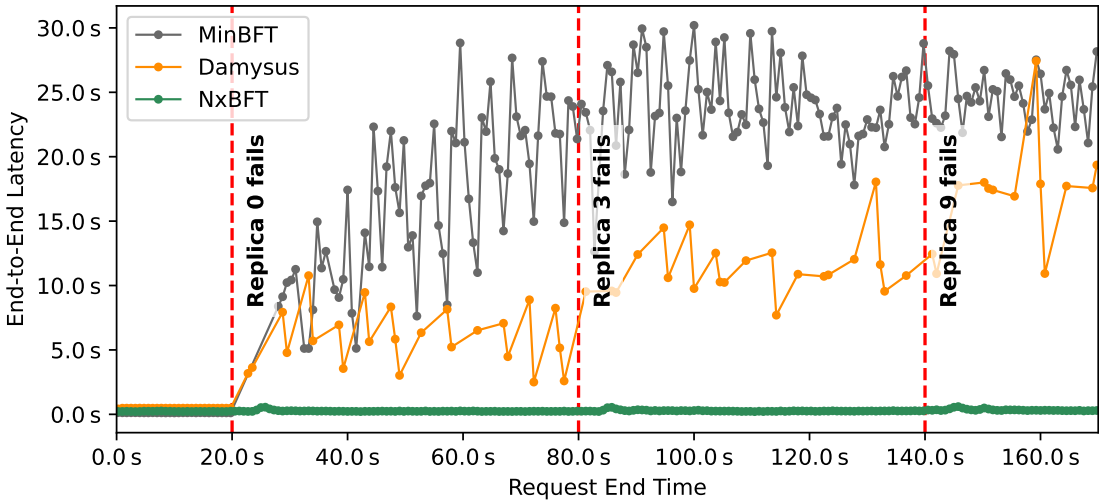


Figure 5.11: End-to-end-latency of MinBFT, Chained-Damysus, and NxBFT with crash faults for $n = 10$, a request rate of 50 kOp/s, 0 Byte payload size, 0.15 ms network round trip latency and the NxB client model. After 20 s, 80 s, and 140 s replicas 0, 3, and 9 respectively crash. After a single crash, MinBFT and Damysus are not capable to handle the incoming request rate. In case of MinBFT, this is caused by the view change after the incumbent leader (replica 0) crashed. In case of Damysus, the rotating leader paradigm periodically selects a crashed replica. NxBFT's small latency spikes stem from the time it takes for $\sim \frac{1}{n}$th of the clients to select a different replica.

Without a recovery procedure, Chained-Damysus cannot recover the response times and each crashed replica increases the end-to-end latency clearly. Consequently, throughput is significantly reduced. Since Chained-Damysus requires a replica to be in charge for two views [Dec+22b, Appendix B], the streamlined version of Damysus worsens this pattern.

As all client requests are uniformly distributed across all replicas, NxBFT's maximum achievable throughput degrades at most by $\frac{c}{n}$ for c crashed replicas. For the request rate of 50 kOp/s, the experiments show that the 7 remaining replicas can handle the additional load: As soon as all clients identified a failed replica, the end-to-end latency recovers.

5.6 Discussion of Results and Limitations

First, and foremost, the empirical analysis confirms (1) the leader bottleneck, (2) the scaling capabilities of DAG-based algorithms, and (3) the significant positive impact of the NxB client model. Based on the comparison, we can conclude that hybrid fault-tolerant leaderless approaches outperform classic BFT leaderless approaches. Moreover, we found clear evidence that the scaling capabilities of DAG-based algorithms are hindered when used as the coordination mechanism in classic BFT SMR. In fact, the results indicate that streamlined protocols achieve faster latencies while maintaining the same throughput when operating with a BFT client model (see Table 5.2). The experiments with network latency (Table 5.4) show that the positive impact of the NxB client model sustains for higher network latencies as well. However, the NxB model comes with a latency penalty as less replicas know of a request and can propose it for decision or will answer after a

decision was made. NxBFT achieves the highest throughput in all configurations, while Chained-Damysus achieves the lowest latencies for small networks with low network latencies. MinBFT achieves the most stable latencies and outperforms Chained-Damysus for latencies $\geq$ 35 ms. The asynchronous and leaderless core of NxBFT allows it to be more resilient to network partitions and replica failures.

We identified the cryptographic work as the main computational bottleneck for all algorithms; the choice of the payload size has a significant impact on the achievable throughput. We can say with a high degree of certainty that parallelization will have a significant positive effect on throughput and latency, especially for NxBFT. However, we were also able to show that NxBFT can hide latencies to a certain extent due to the increased base latency. A bug in an earlier version of the NxBFT implementation caused the hash computation to be taken over by the deserialization logic, which is executed in parallel by ABCperf, reducing the total calculation time of NxBFT. Measurements taken after this bug was discovered and corrected showed that only the throughput values for 0.15 ms network round trip latency in Table 5.4 were incorrectly increased. From this, it can be concluded that the speedup that can be achieved by parallelizing the hashing is not sufficient to make a noticeable difference at a latency of $\geq$ 5 ms. In summary, since a higher network latency only amplifies the latency hiding effect, it remains unclear whether parallelization can improve throughput for large network latencies $\geq$ 35 ms. Nonetheless, suitable parallelization strategies are required to increase performance and resilience even further.

We decided to use constant, homogeneous network conditions without loss in the experiments in Section 5.5.3. We omit loss emulation as packet loss only increases the latency observed above the reliable transport layer protocol. In real-world deployments, however, the network latency is not the same for every pairwise communication link. Moreover, it is impossible to distribute 20 replicas around the world in a way that they all have a pairwise latency of $\geq$ 35 ms. For small latencies, we are confident that our results are a valuable estimate of real-world performance. For higher latencies, we argue that the insights gained are a lower bound for the achievable throughput. Future work should emulate geo distribution latencies as proposed by Berger et al. [BTR24].

The MinBFT implementation has an unresolved bug in its view change logic that may lead view changes to fail when more than two view changes are triggered in a row. To date, we were not able to identify the root cause of the bug. While this illustrates that research software is not production ready, we argue that this bug has no impact on the results of the experiments conducted as we did not trigger multiple view changes. The implementations of Chained-Damysus and NxBFT do not support garbage collection. We argue that the required checkpoint logic only has negligible impact on the performance measurements we conducted. Finally, we decided to implement Chained-Damysus with an exponential backoff mechanism as described in the original paper. The experiments showed that this has a significant negative impact in the case of faults. A pacemaker protocol, e.g., as proposed by Lewis-Pye [Lew22], mitigates the impact by ensuring that correct replicas stay in sync without relying on exponential backoff.

6 Conclusions and Outlook

The aim of this dissertation is to research resilient systems. Since the early days of research into distributed systems in the 1970s, replication has been seen as the way to increase the fault tolerance of an application. Even though there has been increasing research into consensus-free replication systems in recent years (e.g., local-first software [Kle+19]), consensus-based replication can still be considered the standard approach. However, replication-based systems used in production are typically such that can only tolerate benign faults. Although research and industry are working on abstracting replication and making it usable "out of the box", e.g., the Hyperledger project of the linux foundation[1] or the Microsoft Azure Confidential Consortium Framework[2], Byzantine fault-tolerant replication is not widely used, except in cryptocurrencies, because it significantly increases the complexity of a system. This increase in complexity (1) significantly reduces performance metrics and (2), contrary to expectations, can even reduce the resilience of the system. Nevertheless Byzantine fault-tolerant, confidential replication is a promising approach to address the requirements of consortium-operated applications. With the Mobility-as-a-Service ticketing platform, we provided an exemplary analysis for which we are convinced that our observations generalize. Current research supports this conviction (see, e.g., [How+23; Fau+25; Vit+25]).

Consequently, we investigated ways to reduce overhead and increase resilience. To this end, we rely on asynchronous, leaderless atomic broadcast based on a directed acyclic graph (DAG) data structure, which has demonstrated impressive performance and resilience, and Trusted Execution Environments (TEEs), whose use in the hybrid fault model can significantly simplify coordination processes. With the TEE-Rider atomic broadcast based on DAG-Rider [Kei+21], we were able to prove that DAG-based atomic broadcast can tolerate Byzantine faults with a fault tolerance of $n > 2t$ replicas using a small TEE-based signature service. With NxBFT, we have extended TEE-Rider to a full-fledged state machine replication (SMR) algorithm that also supports garbage collection and reconfiguration. We proposed the NxB client model, which allows the inherent parallelism of TEE-Rider (and thus also of NxBFT) to be fully exploited. The extensive measurement studies have shown that NxBFT achieves the highest throughput among all examined scenarios. In contrast to leader-based approaches, NxBFT's performance is almost not impacted when actual crash faults occur. However, the asynchronous approach results in increased latency. If minimal latency is required, classic leader-based approaches should be used. For the Mobility-as-a-Service use case, NxBFT is an important step to a Europe-wide

[1] `https://www.lfdecentralizedtrust.org/` (accessed 2025-08-19)
[2] `https://azure.microsoft.com/de-de/products/managed-ccf` (accessed 2025-08-19)

deployment (e.g., with the nation states as replica operators). In the following, we discuss the implications of limitations of our approach and future work grouped by the three building blocks TEEs, SMR, and atomic broadcast.

6.1 Trusted Execution Environments

TEEs are a central pillar of this dissertation. We used them to simplify coordination processes and to guarantee confidentiality. In doing so, we rely on the integrity of the TEE itself. For virtually every known TEE, there are attacks that break the integrity of the TEE (e.g., [Bul+18; Buh+21; Sri+24]). As a result, there are voices arguing against the use of TEEs and, instead, promoting the use of secure multiparty computation (MPC). As discussed in Chapter 3, MPC can be considered the better choice in terms of achieved security guarantees. However, it must be countered that (1) MPC has a significant performance overhead [Has+19; Kel20; Kil+24] and (2) there is probably no such thing as bug-free software, including cryptographic libraries. In fact, the more complex the cryptographic concept, the higher the likelihood of vulnerabilities [Wei+20; BSW24]. Just as one must trust that the manufacturer of a TEE has developed it with the necessary care and, if necessary, provides updates and bug fixes, one must also trust that the developers of cryptographic libraries have exercised the necessary care.

TEEs offer the opportunity to reduce system complexity. Consequently, if used correctly, they can not only increase the performance of a system but also reduce the size of the trusted computing base and, thus, the attack surface. This leads to a significant demand for resilient and trustworthy TEEs [Kil+24; Rez+25]. There are approaches to harden the security of TEEs [Van+25] and such that define a TEE on openly specified hardware [Lee+20; Cer+25; KVS25] or on FPGAs [Wan+24]. The latter two directions decrease the reliance on proprietary hardware and can potentially offer more transparency and security. In the context of Byzantine fault tolerant SMR, compartmentalization and selective hybridization may further harden the architecture by executing different phases of the protocol on different machines allowing to isolate and protect critical components (e.g., by denying internet access) [Mes+22; LD25]. Another way for hardening TEE-based distributed systems is through hybridization with MPC. By combining the strengths of both approaches, it may be possible to achieve better security guarantees without incurring the full performance overhead of MPC. Recent research [Vit+25] has shown that such hybrid approaches can effectively mitigate the weaknesses of each individual method. For the Mobility-as-a-Service use case, for example, a hybrid approach could theoretically be designed such that agreement properties and metadata confidentiality are lost when the TEE is compromised, but the actual movement data remains confidential and integrity-protected, i.e., a form of graceful degradation. We conclude that, as long as MPC is not competitive in terms of performance, TEEs will remain an important building block for secure distributed systems. Future work should explore MPC-TEE hybrid approaches to further improve the security guarantees achieved.

6.2 State Machine Replication and Distributed Ledgers

Recent consensus research was primarily driven by the massive increase in popularity of distributed ledgers. Distributed ledgers typically deploy mechanisms to define a total order of requests. This total order is derived by using an atomic broadcast abstraction. However, atomic broadcast alone does not define a usable system with which clients can interact. To this end, public distributed ledgers like cryptocurrencies use a mempool: Clients tell some replica about their request which will then, if it is no malicious replica, distribute the request to all replicas such that the request is eventually added to the ledger. Formally, this is no SMR as there are no liveness guarantees for the clients. Most (academic) proposals, however, solely focus on the coordination layer between replicas and, to some extent, ignore the implications that would be caused when aiming to provide guarantees to the clients.

By providing true SMR, we were able to address fundamental issues associated with the use of TEEs in partial synchrony: For TEE-based reliable broadcast, we showed that the reinitializing a TEE, e.g., after a crash, is impossible without sacrificing fault tolerance or reliable broadcast properties. While SMR is agnostic of the replicated application, we can use SMR to define a state transfer logic. During state transfer, the current application state is transferred to a replica without requiring the receiving replica to verify the full request history that led to that state. In summary, the NxBFT SMR framework allows to implement (1) efficient garbage collection and (2) reconfiguration including recovery for TEE-Rider. The proposed garbage collection and reconfiguration protocols should also be applicable for classic leader-based approaches like MinBFT [Ver+11] and Damysus [Dec+22a]. As the demand for highly efficient permissionless ledgers grows as well, an interesting future research direction is to investigate if permissioned-to-permissionless compilers (e.g., by Komatovic et al. [Kom+25]) work with TEE-based protocols and if they can preserve the properties we achieved.

Based on related work and our experience with MinBFT, which has a checkpoint-based garbage collection, we assumed but did not verify that the performance of the checkpoint protocol itself is independent of the actual consensus protocol and that the checkpoint protocol has a negligible impact on the performance of the actual consensus protocol. This means that although NxBFT is a completely specified SMR framework, the current implementation does not contain a checkpoint protocol and therefore neither garbage collection nor reconfiguration are implemented. Moreover, the checkpoint, state transfer, garbage collection, and reconfiguration protocols are only informally specified and proven. Future work should investigate the impact of these missing protocols on the overall performance and resilience. In any case – and we can say this with complete confidence based on our experience during implementation – the complexity of SMR systems requires a formal treatment of algorithms [Gao+24; Ber+25] and implementations [Lat+24; Bas+25; LeB+25] to rule out bugs and misconceptions.

As discussed in Chapter 4, providing guarantees to the clients comes at a cost. We were able to show that the choice of guarantees one wants to provide to clients has a significant impact on the performance of the system. We propose the NxB client model

that minimizes the performance overhead of an SMR client while still providing strong guarantees. The NxB client model relies on the application to provide auditability: If the state of a client changes, the client must have triggered the state change itself or, at least, possess the necessary information to prove that it behaved correctly. One may argue that the combination of TEEs and the NxB client model eliminates any Byzantine behavior from the system. However, replicas can still exhibit Byzantine behavior, e.g., by manipulating the application layer, by not executing a client request (censorship), or by denial-of-service attacks. If auditability guarantees cannot be made, we either require replica-side enclaved proxy execution or threshold cryptography to maintain safety guarantees. The performance impact of both approaches should be evaluated in future work. It remains an open question if forensic mechanisms, e.g., as proposed in [She+21; Sha+22; Ber+24], can be integrated into NxBFT to improve the auditability property of a replicated application.

Like most replicated systems, NxBFT establishes a total order on received requests. Formally, SMR can be implemented with a partial order and a total order is only required for requests that depend on each other [ALO00; Gel+23; HH24]. Deriving a total order could thus be considered unnecessary overhead. Without knowledge of the application, however, it is impossible to identify potentially conflicting requests. Additionally, working on top of a total order reduces the implementation complexity of the application layer; application developers can assume a linear, non-concurrent stream of requests. Based on the analysis of the NxBFT computation time, we know that non-cryptographic operations have a negligible performance impact. Thus, when either parallelizing the cryptographic work on the application layer or offloading authentication and verification tasks to the atomic broadcast layer, it remains unclear what performance gains application-layer parallelism offers and what application characteristics are required for the application layer to become a bottleneck, i.e., slower than the atomic broadcast layer. The authors of Chop Chop [Cam+24] make a similar observation.

6.3 Atomic Broadcast Algorithm and Implementation Design

The TEE-Rider atomic broadcast provides the necessary foundation for building resilient and efficient SMR. The DAG-based approach provides the advantage of inherent parallelism and avoids a leader bottleneck. The DAG allows to order multiple proposals in parallel, thereby increasing efficiency and enabling scaling effects. The efficiency gained, however, merely shifts the "overload point". If NxBFT is burdened with excessively high request rates, metastable failures can still occur. Therefore, proactive rejection mechanisms must also be investigated for Byzantine and hybrid fault-tolerant SMR in order to be able to adequately resolve overload situations without human intervention. Idem [LD24] lays the foundation for this with a proposal for benign fault-tolerant SMR.

Although NxBFT formally assumes partial synchrony, this assumption is only required for state transfer. State transfer, in turn, is only required for reconfiguration and, when using garbage collection, ensuring liveness in the case of faults. The common case operation of NxBFT is equivalent to TEE-Rider and is therefore fully asynchronous. We were

able to show that the asynchronous nature of TEE-Rider achieves significant resilience improvements over established hybrid fault-tolerant protocols. But the asynchrony also introduces challenges, particularly with respect to achievable latencies. When relying on partial synchrony instead, the observed latencies decrease significantly while throughput gains are only slightly reduced [Spi+22]. Typically, this comes at the cost of resilience. Concurrent work has shown that the resilience gap between asynchronous and partially synchronous graph-based approaches can be lowered [Aru+25; Bab+25; Ton+25]. Reputation systems and forensics can further improve the latency by selecting the most suitable replicas for certain tasks [Spi+24; Zha+24b; Gog+25]. Consequently, future work should investigate whether the latency of TEE-Rider and NxBFT can be improved by using a partially synchronous model and reputation systems without sacrificing resilience.

In its current design, NxBFT relies on a fully connected mesh topology for communication and does not use any parallelization within the atomic broadcast logic. As discussed in Chapter 4, the communication topology has a significant impact on the performance and is closely related to parallelization strategies. In Chapter 5, we demonstrated that the cryptographic overhead can be a limiting factor in the performance of NxBFT. Thus, future work should investigate compatible parallelization strategies, e.g., as proposed by Narwhal [Dan+22], that allow to hide latency caused by cryptographic operations and to further increase NxBFT's throughput gains. Moreover, the compatibility of offloading techniques as proposed by Chop Chop [Cam+24] and tree topologies as proposed by Kauri [NMR21] with DAG-based atomic broadcast in general and the hybrid fault model in particular should be explored.

Bibliography

[AAM10] I. Abraham, M. K. Aguilera, and D. Malkhi. "Fast Asynchronous Consensus with Optimal Resilience". In: *Distributed Computing, 24th International Symposium, DISC 2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings*. Vol. 6343. Lecture Notes in Computer Science. Springer, 2010, pp. 4–19. DOI: 10.1007/978-3-642-15763-9_3.

[Abr19a] I. Abraham. *Consensus for State Machine Replication*. 2019. URL: https://decentralizedthoughts.github.io/2019-10-15-consensus-for-state-machine-replication/ (visited on 2025/05/24).

[Abr19b] I. Abraham. *State Machine Replication for Two Servers and One Omission Failure is Impossible even in a lock-step model*. 2019. URL: https://decentralizedthoughts.github.io/2019-11-02-primary-backup-for-2-servers-and-omission-failures-is-impossible/ (visited on 2025/05/09).

[Abr23] I. Abraham. *The CAP Theorem and why State Machine Replication for Two Servers and One Crash Failure is Impossible in Partial Synchrony*. 2023. URL: https://decentralizedthoughts.github.io/2023-07-09-CAP-two-servers-in-psynch/ (visited on 2025/05/21).

[Abr24] I. Abraham. *Flavours of Partial Synchrony*. 2024. URL: https://decentralizedthoughts.github.io/2019-09-13-flavours-of-partial-synchrony/ (visited on 2025/05/11).

[Ack24] D. Ackers. *Account Based Ticketing. Thematische Einordnung des kontenbasierten Tickets in organisatorischer und technischer Sicht*. Tech. rep. Version 1.0. VDV eTicket Service, 2024. URL: https://www.eticket-deutschland.de/media/einordung_accountbasedticketing.pdf (visited on 2025/06/10).

[AEH75] E. A. Akkoyunlu, K. Ekanandham, and R. V. Huber. "Some Constraints and Tradeoffs in the Design of Network Communications". In: *Proceedings of the Fifth Symposium on Operating System Principles, SOSP 1975, The University of Texas at Austin, Austin, Texas, USA, November 19-21, 1975*. ACM, 1975, pp. 67–74. DOI: 10.1145/800213.806523.

[Agu+23] M. K. Aguilera, N. Ben-David, R. Guerraoui, A. Murat, A. Xygkis, and I. Zablotchi. "uBFT: Microsecond-Scale BFT using Disaggregated Memory". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 2023, pp. 862–877. DOI: 10.1145/3575693.3575732.

[AHP25] Y. Amoussou-Guenou, M. Herlihy, and M. Potop-Butucaru. "Asynchronous Byzantine Consensus with Trusted Monotonic Counters". In: *Structural Information and Communication Complexity - 32nd International Colloquium, SIROCCO 2025, Delphi, Greece, June 2-4, 2025, Proceedings*. Vol. 15671. Lecture Notes in Computer Science. Springer, 2025, pp. 23–38. DOI: 10.1007/978-3-031-91736-3_2.

[Akt+23] E. Aktas, C. Cohen, J. Eads, J. Forshaw, and F. Wilhelm. *Intel Trust Domain Extensions (TDX) Security Review*. Tech. rep. Google Cloud Security, 2023. URL: https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf (visited on 2025/05/30).

[Al-+18] A. Al-Momani, F. Kargl, R. K. Schmidt, and C. Bösch. "iRide: A Privacy-Preserving Architecture for Self-Driving Cabs Service". In: *2018 IEEE Vehicular Networking Conference, VNC 2018, Taipei, Taiwan, December 5-7, 2018*. IEEE, 2018, pp. 1–8. DOI: 10.1109/VNC.2018.8628378.

[Ale+22] A. B. Alexandru, E. Blum, J. Katz, and J. Loss. "State Machine Replication Under Changing Network Conditions". In: *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part I*. Vol. 13791. Lecture Notes in Computer Science. Springer, 2022, pp. 681–710. DOI: 10.1007/978-3-031-22963-3_23.

[ALO00] A. Adya, B. Liskov, and P. E. O'Neil. "Generalized Isolation Level Definitions". In: *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*. IEEE Computer Society, 2000, pp. 67–78. DOI: 10.1109/ICDE.2000.839388.

[Ami+22] M. J. Amiri, B. T. Loo, D. Agrawal, and A. E. Abbadi. "Qanaat: A Scalable Multi-Enterprise Permissioned Blockchain System with Confidentiality Guarantees". In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2839–2852. DOI: 10.14778/3551793.3551835.

[Ami+24] M. J. Amiri, C. Wu, D. Agrawal, A. E. Abbadi, B. T. Loo, and M. Sadoghi. "The Bedrock of Byzantine Fault Tolerance: A Unified Platform for BFT Protocols Analysis, Implementation, and Experimentation". In: *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*. USENIX Association, 2024, pp. 371–400. URL: https://www.usenix.org/conference/nsdi24/presentation/amiri.

[Ami+25] Md. Al Amin, H. Tummala, R. Shah, and I. Ray. "Proof of Compliance (PoC): A Consensus Mechanism to Verify the Compliance with Informed Consent Policy in Healthcare". In: *Proceedings of the Fifteenth ACM Conference on Data and Application Security and Privacy, CODASPY 2025, Pittsburgh, PA, USA, June 4-6, 2025*. ACM, 2025, pp. 119–130. DOI: 10.1145/3714393.3726512.

[Amo+18] M. Amoretti, G. Brambilla, F. Medioli, and F. Zanichelli. "Blockchain-Based Proof of Location". In: *2018 IEEE International Conference on Software Quality, Reliability and Security Companion, QRS Companion 2018, Lisbon, Portugal, July 16-20, 2018*. IEEE, 2018, pp. 146–153. DOI: 10.1109/QRS-C.2018.00038.

[Amo+22] M. Amoretti, A. Budianu, G. Caparra, F. D'Agruma, D. Ferrari, G. Penzotti, L. Veltri, and F. Zanichelli. "Enabling Location Based Services with Privacy and Integrity Protection in Untrusted Environments through Blockchain and Secure Computation". In: *4th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications, TPS-ISA 2022, Atlanta, GA, USA, December 14-17, 2022*. IEEE, 2022, pp. 114–123. DOI: 10.1109/TPS-ISA56441.2022.00024.

[AMW25] Z. Avarikioti, M. Maffei, and Y. Wang. "A Security Framework for General Blockchain Layer 2 Protocols". In: *CoRR* abs/2504.14965 (2025). DOI: 10.48550/ARXIV.2504.14965.

[And20] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. 3rd ed. Wiley, 2020. ISBN: 978-1-119-64468-2.

[Ang+23] S. Angel, A. Basu, W. Cui, T. Jaeger, S. Lau, S. T. V. Setty, and S. Singanamalla. "Nimble: Rollback Protection for Confidential Cloud Services". In: *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*. USENIX Association, 2023, pp. 193–208. URL: https://www.usenix.org/conference/osdi23/presentation/angel.

[Ant+18] K. Antoniadis, R. Guerraoui, D. Malkhi, and D.-A. Seredinschi. "State Machine Replication Is More Expensive Than Consensus". In: *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*. Vol. 121. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 7:1–7:18. DOI: 10.4230/LIPICS.DISC.2018.7.

[Ant+23] K. Antoniadis, J. Benhaim, A. Desjardins, E. Poroma, V. Gramoli, R. Guerraoui, G. Voron, and I. Zablotchi. "Leaderless consensus". In: *J. Parallel Distributed Comput.* 176 (2023), pp. 95–113. DOI: 10.1016/J.JPDC.2023.01.009.

[Arn+16] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyers, R. Kapitza, P. R. Pietzuch, and C. Fetzer. "SCONE: Secure Linux Containers with Intel SGX". In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 2016, pp. 689–703. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[Aru+25] B. Arun, Z. Li, F. Suri-Payer, S. Das, and A. Spiegelman. "Shoal++: High Throughput DAG BFT Can Be Fast and Robust!" In: *22nd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2025, Philadelphia, PA, USA, April 28-30, 2025*. USENIX Association, 2025, pp. 813–826. URL: https://www.usenix.org/conference/nsdi25/presentation/arun.

[Aum+13] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. "BLAKE2: Simpler, Smaller, Fast as MD5". In: *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*. Vol. 7954. Lecture Notes in Computer Science. Springer, 2013, pp. 119–135. DOI: 10.1007/978-3-642-38980-1_8.

[Aum17] J.-P. Aumasson. *Serious Cryptography: A Practical Introduction to Modern Encryption*. San Francisco, CA: No Starch Press, 2017. ISBN: 978-1-59327-826-7.

[AV11] R. Almeida and M. Vieira. "Benchmarking the Resilience of Self-Adaptive Software Systems: Perspectives and Challenges". In: *2011 ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, Waikiki, Honolulu , HI, USA, May 23-24, 2011*. ACM, 2011, pp. 190–195. DOI: 10.1145/1988008.1988035.

[AV25] P.-L. Aublin and A. Vogel. "Mitigating Cryptographic Bottlenecks of Low-Latency BFT Protocols". In: *Distributed Applications and Interoperable Systems - 25th IFIP WG 6.1 International Conference, DAIS 2025, Held as Part of the 20th International Federated Conference on Distributed Computing Techniques, DisCoTec 2025, Lille, France, June 16-20, 2025, Proceedings*. Vol. 15730. Lecture Notes in Computer Science. Springer, 2025, pp. 43–63. DOI: 10.1007/978-3-031-95728-4_3.

[Avi+04] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Trans. Dependable Secur. Comput.* 1.1 (2004), pp. 11–33. DOI: 10.1109/TDSC.2004.2.

[AW04] H. Attiya and J. L. Welch. *Distributed computing - fundamentals, simulations, and advanced topics*. 2nd ed. Wiley, 2004. ISBN: 978-0-471-45324-6. DOI: 10.1002/0471478210.

[Bab+25] K. Babel, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, A. Koshy, A. Sonnino, and M. Tian. "Mysticeti: Reaching the Latency Limits with Uncertified DAGs". In: *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025. URL: https://www.ndss-symposium.org/ndss-paper/mysticeti-reaching-the-latency-limits-with-uncertified-dags/.

[Bal+17] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamaric, and L. Ryzhyk. "System Programming in Rust: Beyond Safety". In: *ACM SIGOPS Oper. Syst. Rev.* 51.1 (2017), pp. 94–99. DOI: 10.1145/3139645.3139660.

[Ban+21] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi. "Twins: BFT Systems Made Robust". In: *25th International Conference on Principles of Distributed Systems, OPODIS 2021, December 13-15, 2021, Strasbourg, France*. Vol. 217. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 7:1–7:29. DOI: 10.4230/LIPICS.OPODIS.2021.7.

[Bao+21] J. Bao, D. He, M. Luo, and K.-K. R. Choo. "A Survey of Blockchain Applications in the Energy Sector". In: *IEEE Syst. J.* 15.3 (2021), pp. 3370–3381. DOI: 10.1109/JSYST.2020.2998791.

[Bar+18] E. Barker, L. Chen, A. Roginsky, A. Vassilev, and R. Davis. *Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography*. Tech. rep. NIST Special Publication (SP) 800-56A Rev. 3. National Institute of Standards and Technology, 2018. DOI: 10.6028/NIST.SP.800-56Ar3.

[Bar+25] M. Barbaraci, N. Schmid, O. Alpos, M. Senn, and C. Cachin. "Thetacrypt: A Distributed Service for Threshold Cryptography". In: *CoRR* abs/2502.03247 (2025). DOI: 10.48550/ARXIV.2502.03247.

[Bas+25] D. Basin, N. Foster, K. L. McMillan, K. S. Namjoshi, C. Nita-Rotaru, J. M. Smith, P. Zave, and L. D. Zuck. "It Takes a Village: Bridging the Gaps between Current and Formal Specifications for Protocols". In: *Commun. ACM* 68.8 (2025), pp. 50–61. DOI: 10.1145/3706572.

[BCS22] N. Ben-David, B. Y. Chan, and E. Shi. "Revisiting the Power of Non-Equivocation in Distributed Protocols". In: *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*. ACM, 2022, pp. 450–459. DOI: 10.1145/3519270.3538427.

[BDK17] J. Behl, T. Distler, and R. Kapitza. "Hybrids on Steroids: SGX-Based High Performance BFT". In: *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. ACM, 2017, pp. 222–237. DOI: 10.1145/3064176.3064213.

[Bel+23] R. Belen-Saglam, E. Altuncu, Y. Lu, and S. Li. "A systematic literature review of the tension between the GDPR and public blockchain systems". In: *Blockchain: Research and Applications* 4.2 (2023). DOI: https://doi.org/10.1016/j.bcra.2023.100129.

[Ben+20] F. Benhamouda, C. Gentry, S. Gorbunov, S. Halevi, H. Krawczyk, C. Lin, T. Rabin, and L. Reyzin. "Can a Public Blockchain Keep a Secret?" In: *Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part I*. Vol. 12550. Lecture Notes in Computer Science. Springer, 2020, pp. 260–290. DOI: 10.1007/978-3-030-64375-1_10.

[Ben83] M. Ben-Or. "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols (Extended Abstract)". In: *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*. ACM, 1983, pp. 27–30. DOI: 10.1145/800221.806707.

[Ber+06] S. Berger, R. Caceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. "vTPM: Virtualizing the Trusted Platform Module". In: *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*. USENIX Association, 2006. URL: https://www.usenix.org/conference/15t

h-usenix-security-symposium/vtpm-virtualizing-trusted-platform-module.

[Ber+12] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B. Yang. "High-speed high-security signatures". In: *J. Cryptogr. Eng.* 2.2 (2012), pp. 77–89. DOI: 10.1007/S13389-012-0027-1.

[Ber+23] C. Berger, S. Schwarz-Rüsch, A. Vogel, K. Bleeke, L. Jehl, H. P. Reiser, and R. Kapitza. "SoK: Scalability Techniques for BFT Consensus". In: *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2023, Dubai, United Arab Emirates, May 1-5, 2023*. IEEE, 2023, pp. 1–18. DOI: 10.1109/ICBC56567.2023.10174976.

[Ber+24] C. Berger, L. Rodrigues, H. P. Reiser, V. V. Cogo, and A. Bessani. "Chasing Lightspeed Consensus: Fast Wide-Area Byzantine Replication with Mercury". In: *Proceedings of the 25th International Middleware Conference, MIDDLEWARE 2024, Hong Kong, SAR, China, December 2-6, 2024*. ACM, 2024, pp. 158–171. DOI: 10.1145/3652892.3700756.

[Ber+25] N. Bertrand, P. Ghorpade, S. Rubin, B. Scholz, and P. Subotic. "Reusable Formal Verification of DAG-Based Consensus Protocols". In: *NASA Formal Methods - 17th International Symposium, NFM 2025, Williamsburg, VA, USA, June 11-13, 2025, Proceedings*. Vol. 15682. Lecture Notes in Computer Science. Springer, 2025, pp. 138–158. DOI: 10.1007/978-3-031-93706-4_9.

[Bes+08] A. N. Bessani, E. A. Pelinson Alchieri, M. Correia, and J. da Silva Fraga. "DepSpace: a byzantine fault-tolerant coordination service". In: *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*. ACM, 2008, pp. 163–176. DOI: 10.1145/1352592.1352610.

[Bes+13] A. N. Bessani, M. Santos, J. Felix, N. F. Neves, and M. Correia. "On the Efficiency of Durable State Machine Replication". In: *Proceedings of the 2013 USENIX Annual Technical Conference, USENIX ATC 2013, San Jose, CA, USA, June 26-28, 2013*. USENIX Association, 2013, pp. 169–180. URL: https://www.usenix.org/conference/atc13/technical-sessions/presentation/bessani.

[Bes+23] A. Bessani, M. Correia, T. Distler, R. Kapitza, P. E. Verissimo, and J. Yu. "Vivisecting the Dissection: On the Role of Trusted Components in BFT Protocols". In: *CoRR* abs/2312.05714 (2023). DOI: 10.48550/ARXIV.2312.05714.

[BG22] H. Bönisch and M. Grundmann. "Decentralizing Watchtowers for Payment Channels using IPFS". In: *42nd IEEE International Conference on Distributed Computing Systems, ICDCS Workshops, Bologna, Italy, July 10, 2022*. IEEE, 2022, pp. 27–32. DOI: 10.1109/ICDCSW56584.2022.00015.

[Bie+12] M. Biely, Z. Milosevic, N. Santos, and A. Schiper. "S-Paxos: Offloading the Leader for High Throughput State Machine Replication". In: *IEEE 31st Symposium on Reliable Distributed Systems, SRDS 2012, Irvine, CA, USA, October 8-11, 2012*. IEEE Computer Society, 2012, pp. 111–120. DOI: 10.1109/SRDS.2012.66.

[BL07] D. J. Bernstein and T. Lange. "Faster Addition and Doubling on Elliptic Curves". In: *Advances in Cryptology - ASIACRYPT 2007, 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2-6, 2007, Proceedings*. Vol. 4833. Lecture Notes in Computer Science. Springer, 2007, pp. 29–50. DOI: 10.1007/978-3-540-76900-2_3.

[BL20] L. Baird and A. Luykx. "The Hashgraph Protocol: Efficient Asynchronous BFT for High-Throughput Distributed Ledgers". In: *2020 International Conference on Omni-layer Intelligent Systems, COINS 2020, Barcelona, Spain, August 31 - September 2, 2020*. IEEE, 2020, pp. 1–7. DOI: 10.1109/COINS49042.2020.9191430.

[BN05] P. S. L. M. Barreto and M. Naehrig. "Pairing-Friendly Elliptic Curves of Prime Order". In: *Selected Areas in Cryptography, 12th International Workshop, SAC 2005, Kingston, ON, Canada, August 11-12, 2005, Revised Selected Papers*. Vol. 3897. Lecture Notes in Computer Science. Springer, 2005, pp. 319–331. DOI: 10.1007/11693383_22.

[Bot+19] E. Bothos, B. Magoutas, K. Arnaoutaki, and G. Mentzas. "Leveraging Blockchain for Open Mobility-as-a-Service Ecosystems". In: *2019 IEEE/WIC/ACM International Conference on Web Intelligence, WI 2019, Thessaloniki, Greece, October 14-17, 2019 - Companion Volume*. ACM, 2019, pp. 292–296. DOI: 10.1145/3358695.3361844.

[Bou10] J.-Y. L. Boudec. *Performance Evaluation of Computer and Communication Systems*. 1st ed. Computer and communication sciences. CRC Press, 2010. ISBN: 978-1-43-984992-7.

[Bra+19] M. Brandenburger, C. Cachin, R. Kapitza, and A. Sorniotti. "Trusted Computing Meets Blockchain: Rollback Attacks and a Solution for Hyperledger Fabric". In: *38th Symposium on Reliable Distributed Systems, SRDS 2019, Lyon, France, October 1-4, 2019*. IEEE, 2019, pp. 324–333. DOI: 10.1109/SRDS47363.2019.00045.

[Bra87] G. Bracha. "Asynchronous Byzantine Agreement Protocols". In: *Inf. Comput.* 75.2 (1987), pp. 130–143. DOI: 10.1016/0890-5401(87)90054-X.

[BRB21] C. Berger, H. P. Reiser, and A. Bessani. "Making Reads in BFT State Machine Replication Fast, Linearizable, and Live". In: *40th International Symposium on Reliable Distributed Systems, SRDS 2021, Chicago, IL, USA, September 20-23, 2021*. IEEE, 2021, pp. 1–12. DOI: 10.1109/SRDS53918.2021.00010.

[Bre+21] J. Brendel, C. Cremers, D. Jackson, and M. Zhao. "The Provable Security of Ed25519: Theory and Practice". In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1659–1676. DOI: 10.1109/SP40001.2021.00042.

[BSA14] A. N. Bessani, J. Sousa, and E. A. P. Alchieri. "State Machine Replication for the Masses with BFT-SMART". In: *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*. IEEE Computer Society, 2014, pp. 355–362. DOI: 10.1109/DSN.2014.43.

[BSW24] J. Blessing, M. A. Specter, and D. J. Weitzner. "Cryptography in the Wild: An Empirical Analysis of Vulnerabilities in Cryptographic Libraries". In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024*. ACM, 2024. DOI: 10.1145/3634737.3657012.

[BT85] G. Bracha and S. Toueg. "Asynchronous Consensus and Broadcast Protocols". In: *J. ACM* 32.4 (1985), pp. 824–840. DOI: 10.1145/4221.214134.

[BTR24] C. Berger, S. B. Toumia, and H. P. Reiser. "Exploring Scalability of BFT Blockchain Protocols through Network Simulations". In: *Formal Aspects Comput.* 36.4 (2024), 24:1–24:29. DOI: 10.1145/3689343.

[Buc+24] E. Buchholz, A. Abuadbba, S. Wang, S. Nepal, and S. S. Kanhere. "SoK: Can Trajectory Generation Combine Privacy and Utility?" In: *Proc. Priv. Enhancing Technol.* 2024.3 (2024), pp. 75–93. DOI: 10.56553/POPETS-2024-0068.

[Buc16] E. Buchman. "Tendermint: Byzantine Fault Tolerance in the Age of Blockchains". Advised by G. Taylor. Master's Thesis. The University of Guelph, Guelph, Ontario, Canada, 2016. URL: http://hdl.handle.net/10214/9769.

[Buh+21] R. Buhren, H. N. Jacob, T. Krachenfels, and J.-P. Seifert. "One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. ACM, 2021, pp. 2875–2889. DOI: 10.1145/3460120.3484779.

[Bul+18] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. USENIX Association, 2018, pp. 991–1008. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[Bun22] Bundeskartellamt. *Fair competition for digital mobility services – Bundeskartellamt issues statement of objections against Deutsche Bahn*. Accessed: 2025-02-25. 2022. URL: https://www.bundeskartellamt.de/SharedDocs/Meldung/EN/Pressemitteilungen/2022/20_04_2022_Bahn.html.

[Bun23] Bundeskartellamt. *Open markets for digital mobility services – Deutsche Bahn must end restrictions of competition*. Accessed: 2025-02-25. 2023. URL: https://www.bundeskartellamt.de/SharedDocs/Meldung/EN/Pressemitteilungen/2023/28_06_2023_DB_Mobilitaet.html.

[Bun25] Bundesamt für Sicherheit in der Informationstechnik (BSI). *Trusted Platform Module 2.0 Library Part 0: Introduction*. Tech. rep. BSI-SMGW25/001. 2025. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Broschueren/Smart-Meter-Gateway.pdf (visited on 2025/07/02).

[But17] V. Buterin. *The Meaning of Decentralization*. Accessed: 2025-01-25. 2017. URL: https://medium.com/@VitalikButerin/the-meaning-of-decentralization-a0c92b76a274 (visited on 2025/01/27).

[Cac+01] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. "Secure and Efficient Asynchronous Broadcast Protocols". In: *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*. Vol. 2139. Lecture Notes in Computer Science. Springer, 2001, pp. 524–541. DOI: 10.1007/3-540-44647-8_31.

[Cal+17] F. Callegati, M. Gabbrielli, S. Giallorenzo, A. Melis, and M. Prandini. "Smart mobility for all: A global federated market for mobility-as-a-service operators". In: *20th IEEE International Conference on Intelligent Transportation Systems, ITSC 2017, Yokohama, Japan, October 16-19, 2017*. IEEE, 2017, pp. 1–8. DOI: 10.1109/ITSC.2017.8317701.

[Cal+18] F. Callegati, S. Giallorenzo, A. Melis, and M. Prandini. "Cloud-of-Things meets Mobility-as-a-Service: An insider threat perspective". In: *Comput. Secur.* 74 (2018), pp. 277–295. DOI: 10.1016/J.COSE.2017.10.006.

[Cam+24] M. Camaioni, R. Guerraoui, M. Monti, P.-L. Roman, M. Vidigueira, and G. Voron. "Chop Chop: Byzantine Atomic Broadcast to the Network Limit". In: *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*. USENIX Association, 2024, pp. 269–287. URL: https://www.usenix.org/conference/osdi24/presentation/camaioni.

[Cas00] M. Castro. "Practical Byzantine fault tolerance". Advised by B. Liskov. PhD thesis. Massachusetts Institute of Technology, Cambridge, MA, USA, 2000. URL: https://hdl.handle.net/1721.1/86581.

[CD16] V. Costan and S. Devadas. "Intel SGX Explained". In: *IACR Cryptol. ePrint Arch.* (2016). URL: http://eprint.iacr.org/2016/086.

[Cer+20] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto. "SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1416–1432. DOI: 10.1109/SP40000.2020.00061.

[Cer+25] D. Cerdeira, J. Martins, N. Santos, and S. Pinto. "AnyTEE: An Open and Interoperable Software Defined TEE Framework". In: *IEEE Access* 13 (2025), pp. 109983–109998. DOI: 10.1109/ACCESS.2025.3581487.

[CG24] K.-F. Chu and W. Guo. "Privacy-Preserving Federated Deep Reinforcement Learning for Mobility-as-a-Service". In: *IEEE Trans. Intell. Transp. Syst.* 25.2 (2024), pp. 1882–1896. DOI: 10.1109/TITS.2023.3317358.

[CGR11] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)* Springer, 2011. ISBN: 978-3-642-15259-7. DOI: 10.1007/978-3-642-15260-3.

[Che+17] L. Chen, S. Thombre, K. Järvinen, E. Simona Lohan, A. Alén-Savikko, H. Leppäkoski, M. Z. H. Bhuiyan, S. Bu-Pasha, G. N. Ferrara, S. Honkala, J. Lindqvist, L. Ruotsalainen, P. Korpisaari, and H. Kuusniemi. "Robustness, Security and Privacy in Location-Based Services for Future IoT: A Survey". In: *IEEE Access* 5 (2017), pp. 8956–8977. DOI: 10.1109/ACCESS.2017.2695525.

[Che+19] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. M. Johnson, A. Juels, A. Miller, and D. Song. "Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts". In: *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*. IEEE, 2019, pp. 185–200. DOI: 10.1109/EUROSP.2019.00023.

[Che+23] L. Chen, D. Moody, K. Randall, A. Regenscheid, and A. Robinson. *Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters*. Tech. rep. NIST Special Publication (SP) 800-186. National Institute of Standards and Technology, 2023. DOI: 10.6028/NIST.SP.800-186.

[Chu+07] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. "Attested append-only memory: making adversaries stick to their word". In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. ACM, 2007, pp. 189–204. DOI: 10.1145/1294261.1294280.

[CKS05] C. Cachin, K. Kursawe, and V. Shoup. "Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography". In: *J. Cryptol.* 18.3 (2005), pp. 219–246. DOI: 10.1007/S00145-005-0318-0.

[CL02] M. Castro and B. Liskov. "Practical byzantine fault tolerance and proactive recovery". In: *ACM Trans. Comput. Syst.* 20.4 (2002), pp. 398–461. DOI: 10.1145/571637.571640.

[CL99] M. Castro and B. Liskov. "Practical Byzantine Fault Tolerance". In: *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*. USENIX Association, 1999, pp. 173–186. URL: https://www.usenix.org/legacy/publications/library/proceedings/osdi99/full_papers/castro/castro.ps.

[Cle+12] A. Clement, F. Junqueira, A. Kate, and R. Rodrigues. "On the (limited) power of non-equivocation". In: *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*. ACM, 2012, pp. 301–308. DOI: 10.1145/2332432.2332490.

[CNV04] M. Correia, N. F. Neves, and P. Verissimo. "How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems". In: *23rd International Symposium on Reliable Distributed Systems (SRDS 2004), 18-20 October 2004, Florianpolis, Brazil*. IEEE Computer Society, 2004, pp. 174–183. DOI: 10.1109/RELDIS.2004.1353018.

[CNV06] M. Correia, N. F. Neves, and P. Verissimo. "From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures". In: *Comput. J.* 49.1 (2006), pp. 82–96. DOI: 10.1093/COMJNL/BXH145.

[Col17] Z. Cole. "Network simulation or emulation?" In: *Network World* (2017). URL: https://www.networkworld.com/article/964353/network-simulation-or-emulation.html (visited on 2025/08/03).

[Con+24] A. Constantinescu, D. Ghinea, J. Sliwinski, and R. Wattenhofer. "Brief Announcement: Unifying Partial Synchrony". In: *38th International Symposium on Distributed Computing, DISC 2024, October 28 to November 1, 2024, Madrid, Spain*. Vol. 319. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 43:1–43:7. DOI: 10.4230/LIPICS.DISC.2024.43.

[Cor+13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. "Spanner: Google's Globally Distributed Database". In: *ACM Trans. Comput. Syst.* 31.3 (2013), p. 8. DOI: 10.1145/2491245.

[Cot20] C. D. Cottrill. "MaaS surveillance: Privacy considerations in mobility as a service". In: *Transportation Research Part A: Policy and Practice*. Developments in Mobility as a Service (MaaS) and Intelligent Mobility 131 (2020), pp. 50–57. DOI: 10.1016/j.tra.2019.09.026.

[CP02] C. Cachin and J. A. Poritz. "Secure Intrusion-tolerant Replication on the Internet". In: *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*. IEEE Computer Society, 2002, pp. 167–176. DOI: 10.1109/DSN.2002.1028897.

[Cra+18] T. Crain, V. Gramoli, M. Larrea, and M. Raynal. "DBFT: Efficient Leaderless Byzantine Consensus and its Application to Blockchains". In: *17th IEEE International Symposium on Network Computing and Applications, NCA 2018, Cambridge, MA, USA, November 1-3, 2018*. IEEE, 2018, pp. 1–8. DOI: 10.1109/NCA.2018.8548057.

[Cro+16] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. E. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, D. Song, and R. Wattenhofer. "On Scaling Decentralized Blockchains - (A Position Paper)". In: *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*. Vol. 9604. Lecture Notes in Computer Science. Springer, 2016, pp. 106–125. DOI: 10.1007/978-3-662-53357-4_8.

[CS20a] B. Y. Chan and E. Shi. "Streamlet: Textbook Streamlined Blockchains". In: *AFT '20: 2nd ACM Conference on Advances in Financial Technologies, New York, NY, USA, October 21-23, 2020*. ACM, 2020, pp. 1–11. DOI: 10.1145/3419614.3423256.

[CS20b] C. O. Cruz and J. M. Sarmento. ""Mobility as a Service" Platforms: A Critical Path towards Increasing the Sustainability of Transportation Systems". In: *Sustainability* 12.16 (2020). DOI: 10.3390/su12166368.

[CT05] C. Cachin and S. Tessaro. "Asynchronous Verifiable Information Dispersal". In: *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*. Vol. 3724. Lecture Notes in Computer Science. Springer, 2005, pp. 503–504. DOI: 10.1007/11561927_42.

[CVL09] M. Correia, G. S. Veronese, and L. C. Lung. *Asynchronous Byzantine Consensus with 2f+1 Processes (extended version)*. Technical Report TR-09-17. Lisboa, Portugal: Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa, 2009. URL: https://repositorio.ulisboa.pt/bitstream/10451/14134/1/09-17.pdf.

[CVL10] M. Correia, G. S. Veronese, and L. C. Lung. "Asynchronous Byzantine consensus with 2f+1 processes". In: *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*. ACM, 2010, pp. 475–480. DOI: 10.1145/1774088.1774187.

[Dan+22] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman. "Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus". In: *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. ACM, 2022, pp. 34–50. DOI: 10.1145/3492321.3519594.

[Dat19] Berliner Beauftrage für Datenschutz und Informationsfreiheit. *Datenschutz und Informationsfreiheit. Jahresbericht 2019*. Ed. by Berliner Beauftrage für Datenschutz und Informationsfreiheit. 2019. URL: https://www.datenschutz-berlin.de/fileadmin/user_upload/pdf/jahresbericht/BlnBDI-Jahresbericht-2019-Web.pdf (visited on 2025/06/13).

[DCK16] T. Distler, C. Cachin, and R. Kapitza. "Resource-Efficient Byzantine Fault Tolerance". In: *IEEE Trans. Computers* 65.9 (2016), pp. 2807–2819. DOI: 10.1109/TC.2015.2495213.

[Dec+22a] J. Decouchant, D. Kozhaya, V. Rahli, and J. Yu. "DAMYSUS: streamlined BFT consensus leveraging trusted components". In: *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. ACM, 2022, pp. 1–16. DOI: 10.1145/3492321.3519568.

[Dec+22b] J. Decouchant, D. Kozhaya, V. Rahli, and J. Yu. *DAMYSUS: streamlined BFT consensus leveraging trusted components [Extended Version]*. 2022. URL: https://github.com/vrahli/damysus/blob/main/doc/damysus-extended.pdf (visited on 2025/08/13).

[Dec+24] J. Decouchant, D. Kozhaya, V. Rahli, and J. Yu. "OneShot: View-Adapting Streamlined BFT Protocols with Trusted Execution Environments". In: *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2024, San Francisco, CA, USA, May 27-31, 2024*. IEEE, 2024, pp. 1022–1033. DOI: 10.1109/IPDPS57955.2024.00095.

[Din+17] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan. "BLOCKBENCH: A Framework for Analyzing Private Blockchains". In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 2017, pp. 1085–1100. DOI: 10.1145/3035918.3064033.

[Dis21] T. Distler. "Byzantine Fault-tolerant State-machine Replication from a Systems Perspective". In: *ACM Comput. Surv.* 54.1 (2021), 24:1–24:38. DOI: 10.1145/3436728.

[DLS88] C. Dwork, N. A. Lynch, and L. J. Stockmeyer. "Consensus in the presence of partial synchrony". In: *J. ACM* 35.2 (1988), pp. 288–323. DOI: 10.1145/42282.42283.

[DPL24] K. Davis, B. Peabody, and P. Leach. *Universally Unique IDentifiers (UUIDs)*. RFC 9562. 2024. DOI: 10.17487/RFC9562.

[Dra+19] K. Drakonakis, P. Ilia, S. Ioannidis, and J. Polakis. "Please Forget Where I Was Last Summer: The Privacy Risks of Public Location (Meta)Data". In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. DOI: 10.14722/ndss.2019.23151. URL: https://www.ndss-symposium.org/ndss-paper/please-forget-where-i-was-last-summer-the-privacy-risks-of-public-location-metadata/.

[DZ16] S. Duan and H. Zhang. "Practical State Machine Replication with Confidentiality". In: *35th IEEE Symposium on Reliable Distributed Systems, SRDS 2016, Budapest, Hungary, September 26-29, 2016*. IEEE Computer Society, 2016, pp. 187–196. DOI: 10.1109/SRDS.2016.031.

[EC16] European Parliament and Council of the European Union. *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)*. 2016. URL: https://eur-lex.europa.eu/eli/reg/2016/679/oj (visited on 2025/06/13).

[EC22] European Parliament and Council of the European Union. *Directive (EU) 2022/2557 of the European Parliament and of the Council of 14 December 2022 on the resilience of critical entities and repealing Council Directive 2008/114/EC (Directive on the Resilience of Critical Entities)*. 2022. URL: https://eur-lex.europa.eu/eli/dir/2022/2557/oj (visited on 2025/07/02).

[EC25] M. Esmaili and K. J. Christensen. "Performance Modeling of Public Permissionless Blockchains: A Survey". In: *ACM Comput. Surv.* 57.7 (2025), 174:1–174:35. DOI: 10.1145/3715094.

[Eck+18] J. Eckhardt, A. Aaapaoja, L. Nykänen, and J. Sochor. "The European Roadmap 2025 for Mobility as a Service". In: *7th Transport Research Arena TRA 2018 (Vienna, April 16-19, 2018)*. 2018. URL: https://www.researchgate.net/publication/321586613_The_European_Roadmap_2025_for_Mobility_as_a_Service (visited on 2025/06/08).

[ED21] M. Eischer and T. Distler. "Egalitarian Byzantine Fault Tolerance". In: *26th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2021, Perth, Australia, December 1-4, 2021*. IEEE, 2021, pp. 1–10. DOI: 10.1109/PRDC53464.2021.00019.

[Edd22] W. Eddy. *Transmission Control Protocol (TCP)*. RFC 9293. 2022. DOI: 10.17487/RFC9293.

[Ein24] A. Einarsdóttir. "Enabling Parallel Voting in Streamlined Consensus Protocols". Advised by J. Decouchant. Master's Thesis. Delft University of Technology, Delft, The Netherlands, 2024. URL: https://resolver.tudelft.nl/uuid:8b40ec25-155e-4c92-b746-290baacab577.

[Eis+25] J. Eisoldt, A. Galanou, A. Ruzhanskiy, N. Küchenmeister, Y. Baburkin, T. Dai, I- Gudymenko, S. Köpsell, and R. Kapitza. "SoK: A cloudy view on trust relationships of CVMs - How Confidential Virtual Machines are falling short in Public Cloud". In: *CoRR* abs/2503.08256 (2025). DOI: 10.48550/ARXIV.2503.08256.

[End25a] M. Endler. *Prototyping in Rust*. 2025. URL: https://corrode.dev/blog/prototyping/ (visited on 2025/08/04).

[End25b] M. Endler. *Rust for Foundational Software*. 2025. URL: https://corrode.dev/blog/foundational-software/ (visited on 2025/08/04).

[Ern+23] J. Ernstberger, J. Lauinger, F. Elsheimy, L. Zhou, S. Steinhorst, R. Canetti, A. Miller, A. Gervais, and D. Song. "SoK: Data Sovereignty". In: *8th IEEE European Symposium on Security and Privacy, EuroS&P 2023, Delft, Netherlands, July 3-7, 2023*. IEEE, 2023, pp. 122–143. DOI: 10.1109/EUROSP57164.2023.00017.

[Eur21] Eurostat. *Passenger mobility statistics*. 2021. URL: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Passenger_mobility_statistics (visited on 2025/06/14).

[Eur23] European Commission. *Proposal for a Regulation of the European Parliament and of the Council on the establishment of the digital euro*. 2023. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:52023PC0369 (visited on 2025/07/02).

[Eur24] European Central Bank. *Progress on the preparation phase of a digital euro. Second progress report*. 2024. URL: https://www.ecb.europa.eu/euro/digital_euro/progress/shared/pdf/ecb.deprp202412.en.pdf (visited on 2025/07/02).

[Fau+25] D. Faut, V. Fetzer, J. Müller-Quade, M. Raiber, and A. Rupp. "POBA: Privacy-Preserving Operator-Side Bookkeeping and Analytics". In: *IACR Commun. Cryptol.* 2.2 (2025), p. 7. DOI: 10.62056/AV11ZO-3Y.

[FCV25] J. Ferré-Queralt, J. Castellà-Roca, and A. Viejo. "Blockchain-based hierarchical smart contracts to prevent user profiling in decentralized energy trading systems". In: *Sustainable Energy, Grids and Networks* 43 (2025). DOI: 10.1016/j.segan.2025.101762.

[Fet+22] V. Fetzer, M. Keller, S. Maier, M. Raiber, A. Rupp, and R. Schwerdt. "PUBA: Privacy-Preserving User-Data Bookkeeping and Analytics". In: *Proc. Priv. Enhancing Technol.* 2022.2 (2022), pp. 447–516. DOI: 10.2478/POPETS-2022-0054.

[Fet24] V. Fetzer. "Towards Solving Real-World Challenges for Privacy-Preserving Systems". Advised by A. Rupp. PhD thesis. Karlsruher Institut für Technologie (KIT), Karlsruhe, Germany, 2024.

[Feu+22] S. Feulner, J. Sedlmeir, V. Schlatt, and N. Urbach. "Exploring the use of self-sovereign identity for event ticketing systems". In: *Electron. Mark.* 32.3 (2022), pp. 1759–1777. DOI: 10.1007/S12525-022-00573-9.

[FFF07] M. Farsi, A. Fetz, and M. Filippini. "Economies of Scale and Scope in Local Public Transportation". In: *Journal of Transport Economics and Policy (JTEP)* 41.3 (2007), pp. 345–361.

[FH16] P. De Filippi and S. Hassan. "Blockchain technology as a regulatory technology: From code is law to law is code". In: *First Monday* 21.12 (2016). DOI: 10.5210/FM.V21I12.7113.

[FHF20] F. Fatz, P. Hake, and P. Fettke. "Confidentiality-preserving Validation of Tax Documents on the Blockchain". In: *Entwicklungen, Chancen und Herausforderungen der Digitalisierung: Proceedings der 15. Internationalen Tagung Wirtschaftsinformatik, WI 2020, Potsdam, Germany, March 9-11, 2020. Zentrale Tracks*. GITO Verlag, 2020, pp. 1262–1277. DOI: 10.30844/WI_2020_L1-FATZ.

[FLP83] M. J. Fischer, N. A. Lynch, and M. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 21-23, 1983, Colony Square Hotel, Atlanta, Georgia, USA*. ACM, 1983, pp. 1–7. DOI: 10.1145/588058.588060.

[FNL22] X. Fan, B. Niu, and Z. Liu. "Scalable blockchain storage systems: research progress and models". In: *Computing* 104.6 (2022), pp. 1497–1524. DOI: 10.1007/S00607-022-01063-8.

[FSZ18] S. Friebe, I. Sobik, and M. Zitterbart. "DecentID: Decentralized and Privacy-Preserving Identity Storage System Using Smart Contracts". In: *17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications / 12th IEEE International Conference On Big Data Science And Engineering, TrustCom/BigDataSE 2018, New York, NY, USA, August 1-3, 2018*. IEEE, 2018, pp. 37–42. DOI: 10.1109/TRUSTCOM/BIGDATASE.2018.00016.

[Fu+22] X. Fu, H. Wang, P. Shi, and X. Zhang. "Teegraph: A Blockchain consensus algorithm based on TEE and DAG for data sharing in IoT". In: *J. Syst. Archit.* 122 (2022), p. 102344. DOI: 10.1016/J.SYSARC.2021.102344.

[Gao+24] S. Gao, B. Zhan, Z. Wu, and L. Zhang. "Verifying Randomized Consensus Protocols with Common Coins". In: *54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2024, Brisbane, Australia, June 24-27, 2024*. IEEE, 2024, pp. 403–415. DOI: 10.1109/DSN58291.2024.00047.

[Gar+23] Z. Garroussi, A. Legrain, S. Gambs, V. Gautrais, and B. Sansò. "Data privacy for Mobility as a Service". In: *CoRR* abs/2310.10663 (2023). DOI: 10.48550/ARXIV.2310.10663.

[Geb+22] L. Gebhardt, M. Leinweber, F. Jacob, and H. Hartenstein. "Grasping the Concept of Decentralized Systems for Instant Messaging". In: *WiPSCE '22: The 17th Workshop in Primary and Secondary Computing Education, Morschach, Switzerland, 31 October 2022 - 2 November 2022*. ACM, 2022, 10:1–10:6. DOI: 10.1145/3556787.3556864.

[Gel+23] R. Gelashvili, A. Spiegelman, Z. Xiang, G. Danezis, Z. Li, D. Malkhi, Y. Xia, and R. Zhou. "Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing". In: *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023*. ACM, 2023, pp. 232–244. DOI: 10.1145/3572848.3577524.

[Gep+22] T. Geppert, S. Deml, D. Sturzenegger, and N. Ebert. "Trusted Execution Environments: Applications and Organizational Challenges". In: *Frontiers Comput. Sci.* 4 (2022). DOI: 10.3389/FCOMP.2022.930741.

[Gir+24] N. Giridharan, F. Suri-Payer, I. Abraham, L. Alvisi, and N. Crooks. "Autobahn: Seamless high speed BFT". In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024*. ACM, 2024, pp. 1–23. DOI: 10.1145/3694715.3695942.

[GK20] J. A. Garay and A. Kiayias. "SoK: A Consensus Taxonomy in the Blockchain Era". In: *Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings*. Vol. 12006. Lecture Notes in Computer Science. Springer, 2020, pp. 284–318. DOI: 10.1007/978-3-030-40186-3_13.

[GKP11] S. Gambs, M.-O. Killijian, and M. Núñez del Prado Cortez. "Show Me How You Move and I Will Tell You Who You Are". In: *Trans. Data Priv.* 4.2 (2011), pp. 103–126. URL: http://www.tdp.cat/issues11/abs.a078a11.php.

[GL05] S. Goldwasser and Y. Lindell. "Secure Multi-Party Computation without Agreement". In: *J. Cryptol.* 18.3 (2005), pp. 247–287. DOI: 10.1007/S00145-005-0319-Z.

[GLH19] M. Grundmann, M. Leinweber, and H. Hartenstein. "Banklaves: Concept for a Trustworthy Decentralized Payment Service for Bitcoin". In: *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2019, Seoul, Korea (South), May 14-17, 2019*. IEEE, 2019, pp. 268–276. DOI: 10.1109/BLOC.2019.8751394.

[GLM23] L. Gebhardt, M. Leinweber, and T. Michaeli. "Investigating the Role of Computing Education for Informed Usage Decision-Making". In: *Proceedings of the 18th WiPSCE Conference on Primary and Secondary Computing Education Research, WiPSCE 2023, Cambridge, United Kingdom, September 27-29, 2023*. ACM, 2023, 29:1–29:2. DOI: 10.1145/3605468.3609776.

[Gog+25] H. Gogada, C. Berger, L. Jehl, H. P. Reiser, and H. Meling. "SmartLog: Metrics-driven Role Assignment for Byzantine Fault-tolerant Protocols". In: *CoRR* abs/2502.15428 (2025). DOI: 10.48550/ARXIV.2502.15428.

[Gol+19] G. Golan-Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu. "SBFT: A Scalable and Decentralized Trust Infrastructure". In: *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*. IEEE, 2019, pp. 568–580. DOI: 10.1109/DSN.2019.00063.

[Gra+23] V. Gramoli, R. Guerraoui, A. Lebedev, C. Natoli, and G. Voron. "Diablo: A Benchmark Suite for Blockchains". In: *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*. ACM, 2023, pp. 540–556. DOI: 10.1145/3552326.3567482.

[Gra+24] V. Gramoli, R. Guerraoui, A. Lebedev, and G. Voron. "Stabl: Blockchain Fault Tolerance". In: *CoRR* abs/2409.13142 (2024). DOI: 10.48550/ARXIV.2409.13142.

[Gra22] V. Gramoli. *Blockchain Scalability and its Foundations in Distributed Systems*. Springer, 2022. ISBN: 978-3-031-12577-5. DOI: 10.1007/978-3-031-12578-2.

[Gra78] J. Gray. "Notes on Data Base Operating Systems". In: *Operating Systems, An Advanced Course*. Vol. 60. Lecture Notes in Computer Science. Springer, 1978, pp. 393–481. DOI: 10.1007/3-540-08755-9_9.

[Gre+24] R. W. Gregory, R. Beck, O. Henfridsson, and N. Yaraghi. "Cooperation Among Strangers: Algorithmic Enforcement of Reciprocal Exchange with Blockchain-Based Smart Contracts". In: *Academy of Management Review* (2024). DOI: 10.5465/amr.2023.0023.

[Gud+20] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais. "SoK: Layer-Two Blockchain Protocols". In: *Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers*. Vol. 12059. Lecture Notes in Computer Science. Springer, 2020, pp. 201–226. DOI: 10.1007/978-3-030-51280-4_12.

[Gue+19] R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovic, and D.-A. Seredinschi. "The Consensus Number of a Cryptocurrency". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*. ACM, 2019, pp. 307–316. DOI: 10.1145/3293611.3331589.

[Gun+16] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar. "Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*. ACM, 2016, pp. 1–16. DOI: 10.1145/2987550.2987583.

[Guo+23] Y. Guo, Z. Wan, H. Cui, X. Cheng, and F. Dressler. "Vehicloak: A Blockchain-Enabled Privacy-Preserving Payment Scheme for Location-Based Vehicular Services". In: *IEEE Trans. Mob. Comput.* 22.11 (2023), pp. 6830–6842. DOI: 10.1109/TMC.2022.3193165.

[Gup+25] S. Gupta, D. Kang, D. Malkhi, and M. Sadoghi. "Carry the Tail in Consensus Protocols". In: *CoRR* abs/2508.12173 (2025). DOI: 10.48550/ARXIV.2508.12173.

[GZH22] M. Grundmann, O. v. Zastrow-Marcks, and H. Hartenstein. "On the Applicability of Payment Channel Networks for Allocation of Transport Ticket Revenues". In: *31st International Conference on Computer Communications and Networks, ICCCN 2022, Honolulu, HI, USA, July 25-28, 2022*. IEEE, 2022, pp. 1–7. DOI: 10.1109/ICCCN54977.2022.9868881.

[Hal+21] A. Haleem, M. Javaid, R. Pratap Singh, R. Suman, and S. Rab. "Blockchain technology applications in healthcare: An overview". In: *Int. J. Intell. Networks* 2 (2021), pp. 130–139. DOI: 10.1016/J.IJIN.2021.09.005.

[Hal25] M. Haller. "Design and Analysis of TEE-based Common Coins for Asynchronous Agreement Primitives". Advised by M. Leinweber. Master's Thesis. Karlsruher Institut für Technologie (KIT), Karlsruhe, Germany, 2025.

[Has+19] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic. "SoK: General Purpose Compilers for Secure Multi-Party Computation". In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 1220–1237. DOI: 10.1109/SP.2019.00028.

[HH24] F. J. Hauck and A. Heß. "Linearizability and State-Machine Replication: Is it a match?" In: *CoRR* abs/2407.01720 (2024). DOI: 10.48550/ARXIV.2407.01720.

[HHM24] A. Heß, F. J. Hauck, and E. Meißner. "Consensus-Agnostic State-Machine Replication". In: *Proceedings of the 25th International Middleware Conference, MIDDLEWARE 2024, Hong Kong, SAR, China, December 2-6, 2024*. ACM, 2024, pp. 341–353. DOI: 10.1145/3652892.3700776.

[Hil+23] B. Hildebrand, M. Baza, T. Salman, S. Tabassum, B. Konatham, F. Amsaad, and A. Razaque. "A comprehensive review on blockchains for Internet of Vehicles: Challenges and directions". In: *Comput. Sci. Rev.* 48 (2023), p. 100547. DOI: 10.1016/J.COSREV.2023.100547.

[How+23] H. Howard, F. Alder, E. Ashton, A. Chamayou, S. Clebsch, M. Costa, A. Delignat-Lavaud, C. Fournet, A. Jeffery, M. Kerner, F. Kounelis, M. A. Kuppe, J. Maffre, M. Russinovich, and C. M. Wintersteiger. "Confidential Consortium Framework: Secure Multiparty Applications with Confidentiality, Integrity, and High Availability". In: *Proc. VLDB Endow.* 17.2 (2023), pp. 225–240. DOI: 10.14778/3626292.3626304.

[HT94] V. Hadzilacos and S. Toueg. *A modular approach to fault-tolerant broadcasts and related problems*. Tech. rep. 94-1425. Cornell University, 1994. URL: https://ecommons.cornell.edu/bitstream/1813/6207/1/94-1425.pdf.

[Hua+22] L. Huang, M. Magnusson, A. B. Muralikrishna, S. Estyak, R. Isaacs, A. Aghayev, T. Zhu, and A. Charapko. "Metastable Failures in the Wild". In: *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 2022, pp. 73–90. URL: https://www.usenix.org/conference/osdi22/presentation/huang-lexiang.

[HW90] M. Herlihy and J. M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), pp. 463–492. DOI: 10.1145/78969.78972.

[Hyl+24] D. Hyland, J. Sousa, G. Voron, A. Bessani, and V. Gramoli. "Ten Myths About Blockchain Consensus". In: *Blockchains: A Handbook on Fundamentals, Platforms and Applications*. Springer International Publishing, 2024, pp. 3–24. ISBN: 978-3-031-32146-7. DOI: 10.1007/978-3-031-32146-7_1.

[Int15] International Electrotechnical Commission. *International Electrotechnical Vocabulary – Part 192: Dependability*. 2015. URL: https://www.electropedia.org/iev/iev.nsf/display?openform&ievref=192-01-22.

[Int21] International Organization for Standardization. *ISO/TS 23258:2021 - Blockchain and distributed ledger technologies — Taxonomy and Ontology*. 2021. URL: https://www.iso.org/standard/75094.html.

[Int24] International Organization for Standardization. *ISO 22739:2024 - Blockchain and distributed ledger technologies — Vocabulary*. 2024. URL: https://www.iso.org/standard/82208.html.

[JGS83] S. R. Ames Jr., M. Gasser, and R. R. Schell. "Security Kernel Design and Implementation: An Introduction". In: *Computer* 16.7 (1983), pp. 14–22. DOI: 10.1109/MC.1983.1654439.

[Jia+22] H. Jiang, J. Li, P. Zhao, F. Zeng, Z. Xiao, and A. Iyengar. "Location Privacy-preserving Mechanisms in Location-based Services: A Comprehensive Survey". In: *ACM Comput. Surv.* 54.1 (2022), 4:1–4:36. DOI: 10.1145/3423165.

[Jit+17] P. Jittrapirom, V. Caiati, A.-M. Feneri, S. Ebrahimigharehbaghi, M. J. A. González, and J. Narayan. "Mobility as a Service: A Critical Review of Definitions, Assessments of Schemes, and Key Challenges". In: *Urban Planning* 2.2 (2017), pp. 13–25. DOI: 10.17645/up.v2i2.931.

[JM25] F. Javed and J. Mangues-Bafalluy. “An Empirical Smart Contracts Latency Analysis on Ethereum Blockchain for Trustworthy Inter-Provider Agreements”. In: *CoRR* abs/2503.01397 (2025). DOI: 10.48550/ARXIV.2503.01397.

[Jol+24] A. A. Jolfaei, A. Rupp, S. Schiffner, and T. Engel. “Why Privacy-Preserving Protocols Are Sometimes Not Enough: A Case Study of the Brisbane Toll Collection Infrastructure”. In: *Proc. Priv. Enhancing Technol.* 2024.1 (2024), pp. 232–257. DOI: 10.56553/POPETS-2024-0014.

[JSS14] J. L. Dautrich Jr., E. Stefanov, and E. Shi. “Burst ORAM: Minimizing ORAM Response Times for Bursty Access Patterns”. In: *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. USENIX Association, 2014, pp. 749–764. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/dautrich.

[JSS20] P. Jauernig, A.-R. Sadeghi, and E. Stapf. “Trusted Execution Environments: Properties, Applications, and Challenges”. In: *IEEE Secur. Priv.* 18.2 (2020), pp. 56–60. DOI: 10.1109/MSEC.2019.2947124.

[Jun+18] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. “RustBelt: securing the foundations of the rust programming language”. In: *Proc. ACM Program. Lang.* 2.POPL (2018), 66:1–66:34. DOI: 10.1145/3158154.

[Kan+21] N. Kannengießer, S. Lins, T. Dehling, and A. Sunyaev. “Trade-offs between Distributed Ledger Technology Characteristics”. In: *ACM Comput. Surv.* 53.2 (2021), 42:1–42:37. DOI: 10.1145/3379463.

[Kan+25a] D. Kang, S. Gupta, D. Malkhi, and M. Sadoghi. “HotStuff-1: Linear Consensus with One-Phase Speculation”. In: *Proc. ACM Manag. Data* 3.3 (2025), 171:1–171:29. DOI: 10.1145/3725308.

[Kan+25b] D. Kang, S. Gupta, D. Malkhi, and M. Sadoghi. “HotStuff-1: Linear Consensus with One-Phase Speculation”. In: *Proc. ACM Manag. Data* 3.3 (2025), 171:1–171:29. DOI: 10.1145/3725308.

[Kap22] A. Kapp. “Collection, usage and privacy of mobility data in the enterprise and public administrations”. In: *Proc. Priv. Enhancing Technol.* 2022.4 (2022), pp. 440–456. DOI: 10.56553/POPETS-2022-0117.

[KCG25] S. Keshavarzi, G. V. Chockler, and A. Gotsman. “TEE is not a Healer: Rollback-Resistant Reliable Storage”. In: *CoRR* abs/2505.18648 (2025). DOI: 10.48550/ARXIV.2505.18648.

[Kei+21] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman. “All You Need is DAG”. In: *PODC ’21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*. ACM, 2021, pp. 165–175. DOI: 10.1145/3465084.3467905.

[Kel20] M. Keller. “MP-SPDZ: A Versatile Framework for Multi-Party Computation”. In: *CCS ’20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. ACM, 2020, pp. 1575–1590. DOI: 10.1145/3372297.3417872.

[KH18] A. Karinsalo and K. Halunen. "Smart Contracts for a Mobility-as-a-Service Ecosystem". In: *2018 IEEE International Conference on Software Quality, Reliability and Security Companion, QRS Companion 2018, Lisbon, Portugal, July 16-20, 2018*. IEEE, 2018, pp. 135–138. DOI: 10.1109/QRS-C.2018.00036.

[Kil+24] Q. Kilbourn, S. Bellemare, J. Passerat-Palmbach, and A. Miller. *Zero Trust Execution Environments*. 2024. URL: https://writings.flashbots.net/ZTEE (visited on 2025/08/19).

[KJH15] J. Köhler, K. Jünemann, and H. Hartenstein. "Confidential database-as-a-service approaches: taxonomy and survey". In: *J. Cloud Comput.* 4 (2015), p. 1. DOI: 10.1186/S13677-014-0025-1.

[KK25] A. Kida and H. Kawashima. "Cataphract: A Batch Processing Method Specialized for BFT Databases". In: *Int. J. Netw. Comput.* 15.1 (2025), pp. 32–50. URL: http://www.ijnc.org/index.php/ijnc/article/view/328.

[KL14] J. Katz and Y. Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014. ISBN: 9781466570269.

[Kle+19] M. Kleppmann, A. Wiggins, P. v. Hardenberg, and M. McGranaghan. "Local-first software: you own your data, in spite of the cloud". In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019, Athens, Greece, October 23-24, 2019*. ACM, 2019, pp. 154–178. DOI: 10.1145/3359591.3359737.

[Kle16] M. Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly, 2016. ISBN: 978-1-4493-7332-0.

[Kom+25] J. Komatovic, A. Lewis-Pye, J. Neu, T. Roughgarden, and E. N. Tas. "From Permissioned to Proof-of-Stake Consensus". In: *CoRR* abs/2506.14124 (2025). DOI: 10.48550/ARXIV.2506.14124.

[Kot+07] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong. "Zyzzyva: speculative byzantine fault tolerance". In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. ACM, 2007, pp. 45–58. DOI: 10.1145/1294261.1294267.

[KPW21] D. Kaplan, J. Powell, and T. Woller. *AMD Memory Encryption*. Tech. rep. Accessed: 2025-05-30. Advanced Micro Devices, Inc., 2021. URL: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf.

[Kuz+23] O. Kuznetsov, O. Peliukh, N. Poluyanenko, S. Bohucharskyi, and I. Kolovanova. "Comparative Analysis of Cryptographic Hash Functions in Blockchain Systems". In: *Proceedings of the Cybersecurity Providing in Information and Telecommunication Systems II co-located with International Conference on Problems of Infocommunications. Science and Technology (PICST 2023), Kyiv, Ukraine, October 26, 2023 (online)*. Vol. 3550. CEUR Workshop Proceedings. CEUR-WS.org, 2023, pp. 81–94. URL: https://ceur-ws.org/Vol-3550/paper7.pdf.

[KVS25] M. Kuhne, S. Volos, and S. Shinde. "Dorami: Privilege Separating Security Monitor on RISC-V TEEs". In: *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*. USENIX Association, 2025, pp. 1149–1166. URL: https://www.usenix.org/conference/usenixsecurity25/presentation/kuhne.

[Lam+19] R. Lamberti, C. Fries, M. Lücking, R. Manke, N. Kannengießer, B. Sturm, M. M. Komarov, W. Stork, and A. Sunyaev. "An Open Multimodal Mobility Platform Based on Distributed Ledger Technology". In: *Internet of Things, Smart Spaces, and Next Generation Networks and Systems - 19th International Conference, NEW2AN 2019, and 12th Conference, ruSMART 2019, St. Petersburg, Russia, August 26-28, 2019, Proceedings*. Vol. 11660. Lecture Notes in Computer Science. Springer, 2019, pp. 41–52. DOI: 10.1007/978-3-030-30859-9_4.

[Lam78] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), pp. 558–565. DOI: 10.1145/359545.359563.

[Lam98] L. Lamport. "The Part-Time Parliament". In: *ACM Trans. Comput. Syst.* 16.2 (1998), pp. 133–169. DOI: 10.1145/279227.279229.

[Lat+24] A. Lattuada, T. Hance, J. Bosamiya, M. Brun, C. Cho, H. LeBlanc, P. Srinivasan, R. Achermann, T. Chajed, C. Hawblitzel, J. Howell, J. R. Lorch, O. Padon, and B. Parno. "Verus: A Practical Foundation for Systems Verification". In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024*. ACM, 2024, pp. 438–454. DOI: 10.1145/3694715.3695952.

[Law15] A. M. Law. *Simulation Modeling and Analysis*. 5th ed. McGraw-Hill Education, 2015. ISBN: 978-0-07-340132-4.

[LD24] L. Lawniczak and T. Distler. "Targeting Tail Latency in Replicated Systems with Proactive Rejection". In: *Proceedings of the 25th International Middleware Conference, MIDDLEWARE 2024, Hong Kong, SAR, China, December 2-6, 2024*. ACM, 2024, pp. 327–340. DOI: 10.1145/3652892.3700775.

[LD25] L. Lawniczak and T. Distler. "Hard Shell, Reliable Core: Improving Resilience in Replicated Systems with Selective Hybridization". In: *CoRR* abs/2508.10141 (2025). DOI: 10.48550/ARXIV.2508.10141.

[LeB+25] H. LeBlanc, J. R. Lorch, C. Hawblitzel, C. Huang, Y. Tao, N. Zeldovich, and V. Chidambaram. "PoWER Never Corrupts: Tool-Agnostic Verification of Crash Consistency and Corruption Detection". In: *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*. USENIX Association, 2025, pp. 839–857. URL: https://www.usenix.org/conference/osdi25/presentation/leblanc.

[Lee+20] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song. "Keystone: an open framework for architecting trusted execution environments". In: *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. ACM, 2020, 38:1–38:16. DOI: 10.1145/3342195.3387532.

[Lei+19] M. Leinweber, M. Grundmann, L. Schönborn, and H. Hartenstein. "TEE-Based Distributed Watchtowers for Fraud Protection in the Lightning Network". In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2019 International Workshops, DPM 2019 and CBT 2019, Luxembourg, September 26-27, 2019, Proceedings*. Vol. 11737. Lecture Notes in Computer Science. Springer, 2019, pp. 177–194. DOI: 10.1007/978-3-030-31500-9_11.

[Lei+23] M. Leinweber, N. Kannengießer, H. Hartenstein, and A. Sunyaev. "Leveraging Distributed Ledger Technology for Decentralized Mobility-as-a-Service Ticket Systems". In: *Towards the New Normal in Mobility: Technische und betriebswirtschaftliche Aspekte*. Springer Fachmedien Wiesbaden, 2023, pp. 547–567. ISBN: 978-3-658-39438-7. DOI: 10.1007/978-3-658-39438-7_32.

[Lev+09] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. "TrInc: Small Trusted Hardware for Large Distributed Systems". In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*. USENIX Association, 2009, pp. 1–14. URL: https://www.usenix.org/legacy/event/nsdi09/tech/full_papers/levin/levin.pdf.

[Lew22] A. Lewis-Pye. "Quadratic worst-case message complexity for State Machine Replication in the partial synchrony model". In: *CoRR* abs/2201.01107 (2022).

[LG24] A. Lebedev and V. Gramoli. "On the Relevance of Blockchain Evaluations on Bare Metal". In: *Distributed Ledger Technology*. Springer Nature, 2024, pp. 22–38. DOI: 10.1007/978-981-97-0006-6_2.

[LH23] M. Leinweber and H. Hartenstein. "Brief Announcement: Let It TEE: Asynchronous Byzantine Atomic Broadcast with n ≥ 2f+1". In: *37th International Symposium on Distributed Computing, DISC 2023, October 10-12, 2023, L'Aquila, Italy*. Vol. 281. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 43:1–43:7. DOI: 10.4230/LIPIcs.DISC.2023.43.

[LH25] M. Leinweber and H. Hartenstein. "Not eXactly Byzantine: Efficient and Resilient TEE-Based State Machine Replication". In: *CoRR* abs/2501.11051 (2025), pp. 1–13. DOI: 10.48550/arXiv.2501.11051.

[Li+18] B. Li, N. Weichbrodt, J. Behl, P.-L. Aublin, T. Distler, and R. Kapitza. "Troxy: Transparent Access to Byzantine Fault-Tolerant Systems". In: *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*. IEEE Computer Society, 2018, pp. 59–70. DOI: 10.1109/DSN.2018.00019.

[Li+21] W. Li, C. Meese, M. Nejad, and H. Guo. "P-CFT: A Privacy-preserving and Crash Fault Tolerant Consensus Algorithm for Permissioned Blockchains". In: *2021 4th International Conference on Hot Information-Centric Networking (HotICN)*. 2021, pp. 26–31. DOI: 10.1109/HotICN53262.2021.9680829.

[Li+23] X. Li, B. Zhao, G. Yang, T. Xiang, J. Weng, and R. H. Deng. "A Survey of Secure Computation Using Trusted Execution Environments". In: *CoRR* abs/2302.12150 (2023). DOI: 10.48550/ARXIV.2302.12150.

[Lia+24] Z. Liang, V. Jabrayilov, A. Charapko, and A. Aghayev. “The Cost of Garbage Collection for State Machine Replication”. In: *CoRR* abs/2405.11182 (2024). DOI: 10.48550/ARXIV.2405.11182.

[Lin+17] J. Lind, C. Priebe, D. Muthukumaran, D. O’Keeffe, P.-L. Aublin, F. Kelbert, T- Reiher, D. Goltzsche, D. M. Eyers, R. Kapitza, C. Fetzer, and P. R. Pietzuch. “Glamdring: Automatic Application Partitioning for Intel SGX”. In: *Proceedings of the 2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. USENIX Association, 2017, pp. 285–298. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind.

[Liu+19] J. Liu, W. Li, G. O. Karame, and N. Asokan. “Scalable Byzantine Consensus via Hardware-Assisted Secret Sharing”. In: *IEEE Trans. Computers* 68.1 (2019), pp. 139–151. DOI: 10.1109/TC.2018.2860009.

[LLR02] Y. Lindell, A. Lysyanskaya, and T. Rabin. “On the composition of authenticated byzantine agreement”. In: *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*. ACM, 2002, pp. 514–523. DOI: 10.1145/509907.509982.

[LML14] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. “A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments”. In: *J. Grid Comput.* 12.4 (2014), pp. 559–592. DOI: 10.1007/S10723-014-9314-7.

[LMR12] K. Li, M. Maass, and M. Ralph. *A Type-safe, TPM Backed TLS Infrastructure*. 2012. URL: https://www.cs.cmu.edu/~mmaass/tpm_tls/report.html (visited on 2025/05/30).

[LNS25] A. Lewis-Pye, K. Nayak, and N. Shrestha. “The Pipes Model for Latency and Throughput Analysis”. In: *IACR Cryptol. ePrint Arch.* (2025), p. 1116. URL: https://eprint.iacr.org/2025/1116.

[LR25] A. Lewis-Pye and T. Roughgarden. “Beyond Optimal Fault Tolerance”. In: *CoRR* abs/2501.06044 (2025). DOI: 10.48550/ARXIV.2501.06044.

[LSP82] L. Lamport, R. E. Shostak, and M. C. Pease. “The Byzantine Generals Problem”. In: *ACM Trans. Program. Lang. Syst.* 4.3 (1982), pp. 382–401. DOI: 10.1145/357172.357176.

[LV17] Y. Li and T. Voege. “Mobility as a Service (MaaS): Challenges of Implementation and Policy Required”. In: *Journal of Transportation Technologies* 7.2 (2017), pp. 95–106. DOI: 10.4236/jtts.2017.72007.

[LWS21] F. Lumineau, W. Wang, and O. Schilke. “Blockchain Governance—A New Way of Organizing Collaborations?” In: *Organization Science* 32.2 (2021), pp. 500–521. DOI: 10.1287/orsc.2020.1379.

[MAK13] I. Moraru, D. G. Andersen, and M. Kaminsky. “There is more consensus in Egalitarian parliaments”. In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP ’13, Farmington, PA, USA, November 3-6, 2013*. ACM, 2013, pp. 358–372. DOI: 10.1145/2517349.2517350.

[Mat+17] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. M. Sommer, A. Gervais, A. Juels, and S. Capkun. "ROTE: Rollback Protection for Trusted Execution". In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. USENIX Association, 2017, pp. 1289–1306. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic.

[Mat25] N. Matsakis. *Rust in 2025: Targeting foundational software*. 2025. URL: https://smallcultfollowing.com/babysteps/blog/2025/03/10/rust-2025-intro/ (visited on 2025/08/04).

[McC+08] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. "Flicker: an execution infrastructure for tcb minimization". In: *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*. ACM, 2008, pp. 315–328. DOI: 10.1145/1352592.1352625.

[McK+13] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. "Innovative instructions and software model for isolated execution". In: *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*. ACM, 2013, p. 10. DOI: 10.1145/2487726.2488368.

[Mer25] Merriam-Webster. "equivocal". In: *Merriam-Webster.com Dictionary*. Merriam-Webster, Incorporated, 2025. URL: https://www.merriam-webster.com/dictionary/equivocal (visited on 2025/05/02).

[Mes+22] I. Messadi, M. H. Becker, K. Bleeke, L. Jehl, S. B. Mokhtar, and R. Kapitza. "SplitBFT: Improving Byzantine Fault Tolerance Safety Using Trusted Compartments". In: *Middleware '22: 23rd International Middleware Conference, Quebec, QC, Canada, November 7 - 11, 2022*. ACM, 2022, pp. 56–68. DOI: 10.1145/3528535.3531516.

[Mes+25] I. Messadi, M. E. Gerber, T. Distler, and R. Kapitza. "TEE-Assisted Recovery and Upgrades for Long-Running BFT Services". In: *Availability, Reliability and Security*. 2025, pp. 83–105. DOI: 10.1007/978-3-032-00627-1_5.

[MHS11] Z. Milosevic, M. Hutle, and A. Schiper. "On the Reduction of Atomic Broadcast to Consensus with Byzantine Faults". In: *30th IEEE Symposium on Reliable Distributed Systems (SRDS 2011), Madrid, Spain, October 4-7, 2011*. IEEE Computer Society, 2011, pp. 235–244. DOI: 10.1109/SRDS.2011.36.

[Mil+16] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. "The Honey Badger of BFT Protocols". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, 2016, pp. 31–42. DOI: 10.1145/2976749.2978399.

[Mir+23] A. Miranda-Pascual, P. Guerra-Balboa, J. Parra-Arnau, J. Forné, and T. Strufe. "SoK: Differentially Private Publication of Trajectory Data". In: *Proc. Priv. Enhancing Technol.* 2023.2 (2023), pp. 496–516. DOI: 10.56553/POPETS-2023-0065.

[MK25] S. Mssassi and A. Abou El Kalam. "The Blockchain Trilemma: A Formal Proof of the Inherent Trade-Offs Among Decentralization, Security, and Scalability". In: *Applied Sciences* 15.1 (2025). DOI: 10.3390/app15010019.

[MMS25] A. A. Messaoud, S. B. Mokhtar, and A. Simonet-Boulogne. "Tee-based key-value stores: a survey". In: *VLDB J.* 34.1 (2025), p. 10. DOI: 10.1007/S00778-024-00877-6.

[MN22] D. Malkhi and O. Naor. *The Latest View on View Synchronization*. 2022. URL: https://blog.chain.link/view-synchronization/ (visited on 2025/07/10).

[MN23] D. Malkhi and K. Nayak. "Extended Abstract: HotStuff-2: Optimal Two-Phase Responsive BFT". In: *IACR Cryptol. ePrint Arch.* (2023), p. 397. URL: https://eprint.iacr.org/2023/397.

[Mof+18] S. Mofrad, F. Zhang, S. Lu, and W. Shi. "A comparison study of intel SGX and AMD memory encryption technology". In: *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2018, Los Angeles, CA, USA, June 02-02, 2018*. ACM, 2018, 9:1–9:8. DOI: 10.1145/3214292.3214301.

[Mol+21] M. B. Mollah, J. Zhao, D. Niyato, K.-Y. Lam, X. Zhang, A. M. Y. M. Ghias, L. H. Koh, and L. Yang. "Blockchain for Future Smart Grid: A Comprehensive Survey". In: *IEEE Internet Things J.* 8.1 (2021), pp. 18–43. DOI: 10.1109/JIOT.2020.2993601.

[Mon+13] Y.-A. d. Montjoye, C. A. Hidalgo, M. Verleysen, and V. D. Blondel. "Unique in the Crowd: The privacy bounds of human mobility". In: *Scientific Reports* 3.1 (2013). DOI: 10.1038/srep01376.

[MPP20] G. Del Monte, D. Pennino, and M. Pizzonia. "Scaling blockchains without giving up decentralization and security: a solution to the blockchain scalability trilemma". In: *CryBlock@MOBICOM 2020: Proceedings of the 3rd Workshop on Cryptocurrencies and Blockchains for Distributed Systems, London, UK, September 25, 2020*. ACM, 2020, pp. 71–76. DOI: 10.1145/3410699.3413800.

[MPW24] A. Mostéfaoui, M. Perrin, and J. Weibel. "Brief Announcement: Randomized Consensus: Common Coins Are not the Holy Grail!" In: *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing, PODC 2024, Nantes, France, June 17-21, 2024*. ACM, 2024, pp. 36–39. DOI: 10.1145/3662158.3662824.

[Mur+25] A. D. Murtas, D. C. D'Elia, G. A. D. Luna, P. Felber, L. Querzoni, and V. Schiavoni. "ConfBench: A Tool for Easy Evaluation of Confidential Virtual Machines". In: *55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2025, Naples, Italy, June 23-26, 2025*. IEEE, 2025, pp. 279–288. DOI: 10.1109/DSN64029.2025.00038.

[Nab+21] M. Nabil, A. B. T. Sherif, M. Mahmoud, A. Alsharif, and M. M. Abdallah. "Efficient and Privacy-Preserving Ridesharing Organization for Transferable and Non-Transferable Services". In: *IEEE Trans. Dependable Secur. Comput.* 18.3 (2021), pp. 1291–1306. DOI: 10.1109/TDSC.2019.2920647.

[Nak+23] T. Nakai, A. Sakurai, S. Hironaka, and K. Shudo. "The Blockchain Trilemma Described by a Formula". In: *IEEE International Conference on Blockchain, Blockchain 2023, Danzhou, China, December 17-21, 2023*. IEEE, 2023, pp. 41–46. DOI: 10.1109/BLOCKCHAIN60715.2023.00016.

[Nas+22] B. Nasrulin, M. D. Vos, G. Ishmaev, and J. Pouwelse. "Gromit: Benchmarking the Performance and Scalability of Blockchain Systems". In: *2022 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*. 2022, pp. 56–63. DOI: 10.1109/DAPPS55202.2022.00015.

[Nat15a] National Institute of Standards and Technology. *FIPS 180-4: Secure Hash Standard (SHS)*. Federal Information Processing Standards Publication 180-4. U.S. Department of Commerce, 2015. DOI: 10.6028/NIST.FIPS.180-4.

[Nat15b] National Institute of Standards and Technology. *FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Federal Information Processing Standards Publication 202. U.S. Department of Commerce, 2015. DOI: 10.6028/NIST.FIPS.202.

[Nat23a] National Institute of Standards and Technology. *FIPS 186-5: Digital Signature Standard (DSS)*. Federal Information Processing Standards Publication 186-5. U.S. Department of Commerce, 2023. DOI: 10.6028/NIST.FIPS.186-5.

[Nat23b] National Institute of Standards and Technology. *FIPS 197: Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication 197. U.S. Department of Commerce, 2023. DOI: 10.6028/NIST.FIPS.197-upd1.

[Ngo+19] T. D. Ngoc, B. Bui, S. Bitchebe, A. Tchana, V. Schiavoni, P. Felber, and D. Hagimont. "Everything You Should Know About Intel SGX Performance on Virtualized Systems". In: *Proc. ACM Meas. Anal. Comput. Syst.* 3.1 (2019), 5:1–5:21. DOI: 10.1145/3322205.3311076.

[Niu+22] J. Niu, W. Peng, X. Zhang, and Y. Zhang. "NARRATOR: Secure and Practical State Continuity for Trusted Execution in the Cloud". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. ACM, 2022, pp. 2385–2399. DOI: 10.1145/3548606.3560620.

[Niu+25] J. Niu, X. Wen, G. Wu, S. Liu, J. Yu, and Y. Zhang. "Achilles: Efficient TEE-Assisted BFT Consensus via Rollback Resilient Recovery". In: *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*. ACM, 2025, pp. 193–210. DOI: 10.1145/3689031.3717457.

[NL18] Y. Nir and A. Langley. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 8439. 2018. DOI: 10.17487/RFC8439.

[NMR21] R. Neiheiser, M. Matos, and L. E. T. Rodrigues. "Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation". In: *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. ACM, 2021, pp. 35–48. DOI: 10.1145/3477132.3483584.

[NO25] J. M. L. Neto and B. K. Ozkan. "A Benchmark Framework for Byzantine Fault Tolerance Testing Algorithms (Tool Paper)". In: *6th International Workshop on Formal Methods for Blockchains, FMBC 2025, May 4, 2025, Hamilton, Canada*. Vol. 129. OASIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, 13:1–13:11. DOI: 10.4230/OASICS.FMBC.2025.13.

[Nos+20] M. R. Nosouhi, K. Sood, S. Yu, M. Grobler, and J. Zhang. "PASPORT: A Secure and Private Location Proof Generation and Verification Framework". In: *IEEE Trans. Comput. Soc. Syst.* 7.2 (2020), pp. 293–307. DOI: 10.1109/TCSS.2019.2960534.

[NPP19] T. H. Nguyen, J. Partala, and S. Pirttikangas. "Blockchain-Based Mobility-as-a-Service". In: *28th International Conference on Computer Communication and Networks, ICCCN 2019, Valencia, Spain, July 29 - August 1, 2019*. IEEE, 2019, pp. 1–6. DOI: 10.1109/ICCCN.2019.8847027.

[NPR99] M. Naor, B. Pinkas, and O. Reingold. "Distributed Pseudo-random Functions and KDCs". In: *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*. Vol. 1592. Lecture Notes in Computer Science. Springer, 1999, pp. 327–346. DOI: 10.1007/3-540-48910-X_23.

[OCo+21] J. O'Connor, J.-P. Aumasson, S. Neves, and Z. Wilcox-O'Hearn. *BLAKE3. one function, fast everywhere*. Version 20211102173700. 2021. URL: https://raw.githubusercontent.com/BLAKE3-team/BLAKE3-specs/master/blake3.pdf (visited on 2025/08/14).

[OO14] D. Ongaro and J. K. Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. USENIX Association, 2014, pp. 305–319. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[PBS22] G. D. Pasquale, J. d. Bie, and J. Singh. *Ticketing in Mobility as a Service*. Tech. rep. International Association of Public Transport (UITP), 2022. URL: https://www.smart-ticketing.org/_files/ugd/fcd3b9_2bed2a36a39c437faac2a7dcc069255b.pdf (visited on 2025/06/10).

[PE19] J. D. Preece and J. M. Easton. "Blockchain Technology as a Mechanism for Digital Railway Ticketing". In: *2019 IEEE International Conference on Big Data (IEEE BigData), Los Angeles, CA, USA, December 9-12, 2019*. IEEE, 2019, pp. 3599–3606. DOI: 10.1109/BIGDATA47090.2019.9006293.

[Pet22] D. Petersen. "Automating governance: Blockchain delivered governance for business networks". In: *Industrial Marketing Management* 102 (2022), pp. 177–189. DOI: 10.1016/j.indmarman.2022.01.017.

[PFR20] P. Poux, P. De Filippi, and S. Ramos. "Blockchains for the Governance of Common Goods". In: *DICG@Middleware 2020, Proceedings of the 2020 1st International Workshop on Distributed Infrastructure for Common Good, Delft, The Netherlands, December 7-11, 2020*. ACM, 2020, pp. 7–12. DOI: 10.1145/3428662.3428793.

[PME24] J. D. Preece, C. R. Morris, and J. M. Easton. "Leveraging ontochains for distributed public transit ticketing: An investigation with the system for ticketing ubiquity with blockchains". In: *IET Blockchain* 4.4 (2024), pp. 456–469. DOI: 10.1049/BLC2.12071.

[Pol+22] E. Politou, E. Alepiss, M. Virvou, and C. Patsakis. "Privacy in Blockchain". In: *Privacy and Data Protection Challenges in the Distributed Era*. Springer International Publishing, 2022, pp. 133–149. ISBN: 978-3-030-85443-0. DOI: 10.1007/978-3-030-85443-0_7.

[PP24] D. Pennino and M. Pizzonia. "Virtual Private Blockchains for GDPR: Cheap Private Blockchains out of Public Ones". In: *Proceedings of the Sixth Distributed Ledger Technology Workshop (DLT 2024), Turin, Italy, May 14-15, 2024*. Vol. 3791. CEUR Workshop Proceedings. CEUR-WS.org, 2024. URL: https://ceur-ws.org/Vol-3791/paper6.pdf.

[PPT20] A. Polydoropoulou, I. Pagoni, and A. Tsirimpa. "Ready for Mobility as a Service? Insights from stakeholders and end-users". In: *Travel Behaviour and Society* 21 (2020), pp. 295–306. DOI: 10.1016/j.tbs.2018.11.003.

[PS19] S. Pinto and N. Santos. "Demystifying Arm TrustZone: A Comprehensive Survey". In: *ACM Comput. Surv.* 51.6 (2019), 130:1–130:36. DOI: 10.1145/3291047.

[PSL80] M. C. Pease, R. E. Shostak, and L. Lamport. "Reaching Agreement in the Presence of Faults". In: *J. ACM* 27.2 (1980), pp. 228–234. DOI: 10.1145/322186.322188.

[PT86] K. J. Perry and S. Toueg. "Distributed Agreement in the Presence of Processor and Communication Faults". In: *IEEE Trans. Software Eng.* 12.3 (1986), pp. 477–482. DOI: 10.1109/TSE.1986.6312888.

[PVC18] C. Priebe, K. Vaswani, and M. Costa. "EnclaveDB: A Secure Database Using SGX". In: *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 264–278. DOI: 10.1109/SP.2018.00025.

[Rab83] M. O. Rabin. "Randomized Byzantine Generals". In: *24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7-9 November 1983*. IEEE Computer Society, 1983, pp. 403–409. DOI: 10.1109/SFCS.1983.48.

[Ray02] M. Raynal. "Consensus in Synchronous Systems: A Concise Guided Tour". In: *9th Pacific Rim International Symposium on Dependable Computing (PRDC 2002), 16-18 December 2002, Tsukuba-City, Ibarski, Japan*. IEEE Computer Society, 2002, pp. 221–228. DOI: 10.1109/PRDC.2002.1185641.

[RB11] S. Dani R. Bhamra and K. Burnard. "Resilience: the concept, a literature review and future directions". In: *International Journal of Production Research* 49.18 (2011), pp. 5375–5393. DOI: 10.1080/00207543.2011.563826.

[Res18] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. 2018. DOI: 10.17487/RFC8446.

[Rez+23] F. Rezabek, K. Glas, R. v. Seck, A. Aroua, T. Leonhardt, and G. Carle. "Multilayer Environment and Toolchain for Holistic NetwOrk Design and Analysis". In: *CoRR* abs/2310.16190 (2023). DOI: 10.48550/ARXIV.2310.16190.

[Rez+25] F. Rezabek, J. Passerat-Palmbach, M. Mahhouk, F. Erdmann, and A. Miller. "Narrowing the Gap between TEEs Threat Model and Deployment Strategies". In: *CoRR* abs/2506.14964 (2025). DOI: 10.48550/ARXIV.2506.14964.

[RGV25] M. Raikwar, T. Garrett, and R. Vitenberg. "Deconstruction of Knowledge-Building in DAG-Based DLTs". In: *Distrib. Ledger Technol.* (2025). DOI: 10.1145/3746647.

[Ric24] F. Ricciato. *JOCONDE: Joint On-Demand Computation with No Data Exchange*. 2024. URL: https://cros.ec.europa.eu/joconde (visited on 2025/07/02).

[Ros+21] R. Ross, V. Pillitteri, R. Graubart, D. Bodeau, and R. McQuaid. *Developing Cyber Resilient Systems: A Systems Security Engineering Approach*. Tech. rep. NIST Special Publication (SP) 800-160, Volume 2, Revision 1. National Institute of Standards and Technology, 2021. DOI: 10.6028/NIST.SP.800-160v2r1.

[Roz+21] D. Rozas, A. Tenorio-Fornés, S. Díaz-Molina, and S. Hassan. "When Ostrom Meets Blockchain: Exploring the Potentials of Blockchain for Commons Governance". In: *SAGE Open* 11.1 (2021). DOI: 10.1177/21582440211002526.

[Rua+21] P. Ruan, T. T. A. Dinh, D. Loghin, M. Zhang, G. Chen, Q. Lin, and B. C. Ooi. "Blockchains vs. Distributed Databases: Dichotomy and Fusion". In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2021, pp. 1504–1517. DOI: 10.1145/3448016.3452789.

[Rüs+22] S. Rüsch, K. Bleeke, I. Messadi, S. Schmidt, A. Krampf, K. Olze, S. Stahnke, R. Schmid, L. Pirl, R. Kittel, A. Polze, M. Franz, M. Müller, L. Jehl, and R. Kapitza. "ZugChain: Blockchain-Based Juridical Data Recording in Railway Systems". In: *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2022, Baltimore, MD, USA, June 27-30, 2022*. IEEE, 2022, pp. 67–78. DOI: 10.1109/DSN53405.2022.00019.

[SAB15] M. Sabt, M. Achemlal, and A. Bouabdallah. "Trusted Execution Environment: What It is, and What It is Not". In: *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*. IEEE, 2015, pp. 57–64. DOI: 10.1109/TRUSTCOM.2015.357.

[Sar+18] V. A. Sartakov, N. Weichbrodt, S. Krieter, T. Leich, and R. Kapitza. "STANlite - A Database Engine for Secure Data Processing at Rack-Scale Level". In: *2018 IEEE International Conference on Cloud Engineering, IC2E 2018, Orlando, FL, USA, April 17-20, 2018*. IEEE Computer Society, 2018, pp. 23–33. DOI: 10.1109/IC2E.2018.00024.

[SBK22] V. Sridhar, E. Blum, and J. Katz. "Musings on the HashGraph Protocol: Its Security and Its Limitations". In: *CoRR* abs/2210.13682 (2022). DOI: 10.48550/ARXIV.2210.13682.

[Sch+21] J. Schiffl, M. Grundmann, M. Leinweber, O. Stengele, S. Friebe, and B. Beckert. "Towards Correct Smart Contracts: A Case Study on Formal Verification of Access Control". In: *SACMAT '21: The 26th ACM Symposium on Access Control Models and Technologies, Virtual Event, Spain, June 16-18, 2021*. ACM, 2021, pp. 125–130. DOI: 10.1145/3450569.3463574.

[Sch+22a] M. Schneider, R. J. Masti, S. Shinde, S. Capkun, and R. Perez. "SoK: Hardware-supported Trusted Execution Environments". In: *CoRR* abs/2205.12742 (2022). DOI: 10.48550/ARXIV.2205.12742.

[Sch+22b] S. Schwarz-Rüsch, M. Behlendorf, M. H. Becker, R. Kudlek, H. H. E. Mohamed, F. Schoenitz, L. Jehl, and R. Kapitza. "EventChain: a blockchain framework for secure, privacy-preserving event verification". In: *Middleware '22: 23rd International Middleware Conference, Quebec, QC, Canada, November 7 - 11, 2022*. ACM, 2022, pp. 174–187. DOI: 10.1145/3528535.3565243.

[Sch+24] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale Ebrahim, N. Bissantz, M. Muench, and T. Holz. "SoK: Prudent Evaluation Practices for Fuzzing". In: *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 2024, pp. 1974–1993. DOI: 10.1109/SP54263.2024.00137.

[Sch90] F. B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial". In: *ACM Comput. Surv.* 22.4 (1990), pp. 299–319. DOI: 10.1145/98163.98167.

[Sec+24] R. v. Seck, F. Rezabek, S. Gallenmüller, and G. Carle. "On the Impact of Network Transport Protocols on Leader-Based Consensus Communication". In: *Proceedings of the 6th ACM International Symposium on Blockchain and Secure Critical Infrastructure, BSCI 2024, Singapore, 2 July 2024*. ACM, 2024, pp. 1–11. DOI: 10.1145/3659463.3660030.

[Sha+22] A. Shamis, P. R. Pietzuch, B. Canakci, M. Castro, C. Fournet, E. Ashton, A. Chamayou, S. Clebsch, A. Delignat-Lavaud, M. Kerner, J. Maffre, O. Vrousgou, C. M. Wintersteiger, M. Costa, and M. Russinovich. "IA-CCF: Individual Accountability for Permissioned Ledgers". In: *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*. USENIX Association, 2022, pp. 467–491. URL: https://www.usenix.org/conference/nsdi22/presentation/shamis.

[Sha79] A. Shamir. "How to Share a Secret". In: *Commun. ACM* 22.11 (1979), pp. 612–613. DOI: 10.1145/359168.359176.

[She+20] H. Shen, J. Zhou, Z. Cao, X. Dong, and K.-K. R. Choo. "Blockchain-Based Lightweight Certificate Authority for Efficient Privacy-Preserving Location-Based Service in Vehicular Social Networks". In: *IEEE Internet Things J.* 7.7 (2020), pp. 6610–6622. DOI: 10.1109/JIOT.2020.2974874.

[She+21] P. Sheng, G. Wang, K. Nayak, S. Kannan, and P. Viswanath. "BFT Protocol Forensics". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. ACM, 2021, pp. 1722–1743. DOI: 10.1145/3460120.3484566.

[Sin+08] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. "BFT Protocols Under Fire". In: *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*. USENIX Association, 2008, pp. 189–204. URL: http://www.usenix.org/events/nsdi08/tech/full_papers/singh/singh.pdf.

[SLL14] J. Shao, R. Lu, and X. Lin. "FINE: A fine-grained privacy-preserving location-based service framework for mobile devices". In: *2014 IEEE Conference on Computer Communications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014*. IEEE, 2014, pp. 244–252. DOI: 10.1109/INFOCOM.2014.6847945.

[SLM20] D. Saingre, T. Ledoux, and J.-M. Menaud. "BCTMark: a Framework for Benchmarking Blockchain Technologies". In: *17th IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 2020, Antalya, Turkey, November 2-5, 2020*. IEEE, 2020, pp. 1–8. DOI: 10.1109/AICCSA50499.2020.9316536.

[SMF21] M. U. Sardar, S. Musaev, and C. Fetzer. "Demystifying Attestation in Intel Trust Domain Extensions via Formal Verification". In: *IEEE Access* 9 (2021), pp. 83067–83079. DOI: 10.1109/ACCESS.2021.3087421.

[Spa+23] T. Spannagel, M. Leinweber, A. Castro, and H. Hartenstein. "ABCperf: Performance Evaluation of Fault Tolerant State Machine Replication Made Simple: Demo Abstract". In: *Proceedings of the 24th International Middleware Conference Demos, Posters and Doctoral Symposium, Bologna, Italy, December 11-15, 2023*. ACM, 2023, pp. 35–36. DOI: 10.1145/3626564.3629101.

[Spa22] T. Spannagel. "Benchmarking Byzantine Fault-Tolerant Consensus Algorithms with Focus on Varying Network Conditions at the Example of MinBFT". Advised by M. Leinweber. Bachelor's Thesis. Karlsruher Institut für Technologie (KIT), Karlsruhe, Germany, 2022.

[Spi+22] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias. "Bullshark: DAG BFT Protocols Made Practical". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. ACM, 2022, pp. 2705–2718. DOI: 10.1145/3548606.3559361.

[Spi+24] A. Spiegelman, B. Arun, R. Gelashvili, and Z. Li. "Shoal: Improving DAG-BFT Latency and Robustness". In: *Financial Cryptography and Data Security - 28th International Conference, FC 2024, Willemstad, Curaçao, March 4-8, 2024, Revised Selected Papers, Part I*. Vol. 14744. Lecture Notes in Computer Science. Springer, 2024, pp. 92–109. DOI: 10.1007/978-3-031-78676-1_6.

[SPV22] C. Stathakopoulou, M. Pavlovic, and M. Vukolic. "State machine replication scalability made simple". In: *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. ACM, 2022, pp. 17–33. DOI: 10.1145/3492321.3519579.

[SRC24] R. v. Seck, F. Rezabek, and G. Carle. "Thresh-Hold: Assessment of Threshold Cryptography in Leader-Based Consensus". In: *2024 IEEE 49th Conference on Local Computer Networks (LCN)*. 2024, pp. 1–8. DOI: 10.1109/LCN60385.2024.10639786.

[Sri+24] S. Sridhara, A. Bertschi, B. Schlüter, and S. Shinde. "SIGY: Breaking Intel SGX Enclaves with Malicious Exceptions & Signals". In: *CoRR* abs/2404.13998 (2024). DOI: 10.48550/ARXIV.2404.13998.

[SS12] N. Santos and A. Schiper. "Tuning Paxos for High-Throughput with Batching and Pipelining". In: *Distributed Computing and Networking - 13th International Conference, ICDCN 2012, Hong Kong, China, January 3-6, 2012. Proceedings*. Vol. 7129. Lecture Notes in Computer Science. Springer, 2012, pp. 153–167. DOI: 10.1007/978-3-642-25959-3_11.

[SS13] N. Santos and A. Schiper. "Optimizing Paxos with batching and pipelining". In: *Theor. Comput. Sci.* 496 (2013), pp. 170–183. DOI: 10.1016/J.TCS.2012.10.002.

[Sta+22] C. Stathakopoulou, T. David, M. Pavlovic, and M. Vukolic. "[Solution] Mir-BFT: Scalable and Robust BFT for Decentralized Networks". In: *J. Syst. Res.* 2.1 (2022). DOI: 10.5070/SR32159278.

[Sta25a] Statista Market Insights. *Public Transportation – EU-27*. 2025. URL: https://www.statista.com/outlook/mmo/shared-mobility/public-transportation/eu-27?currency=EUR (visited on 2025/06/14).

[Sta25b] Statista Market Insights. *Shared Mobility – EU-27*. 2025. URL: https://www.statista.com/outlook/mmo/shared-mobility/eu-27 (visited on 2025/06/14).

[Ste+21] O. Stengele, M. Raiber, J. Müller-Quade, and H. Hartenstein. "ETHTID: Deployable Threshold Information Disclosure on Ethereum". In: *Third International Conference on Blockchain Computing and Applications, BCCA 2021, Tartu, Estonia, November 15-17, 2021*. IEEE, 2021, pp. 127–134. DOI: 10.1109/BCCA53669.2021.9657019.

[Str12] L. Strigini. "Fault Tolerance and Resilience: Meanings, Measures and Assessment". In: *Resilience Assessment and Evaluation of Computing Systems*. Ed. by K. Wolter, A. Avritzer, M. Vieira, and A. P. A. v. Moorsel. Springer, 2012, pp. 3–24. DOI: 10.1007/978-3-642-29032-9_1.

[Sur+21] F. Suri-Payer, M. Burke, Z. Wang, Y. Zhang, L. Alvisi, and N. Crooks. "Basil: Breaking up BFT with ACID (transactions)". In: *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. ACM, 2021, pp. 1–17. DOI: 10.1145/3477132.3483552.

[Taf+20] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis. "CockroachDB: The Resilient Geo-Distributed SQL Database". In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 2020, pp. 1493–1509. DOI: 10.1145/3318464.3386134.

[Ton+25] A. Tonkikh, B. Arun, Z. Xiang, Z. Li, and A. Spiegelman. "Raptr: Prefix Consensus for Robust High-Performance BFT". In: *CoRR* abs/2504.18649 (2025). DOI: 10.48550/ARXIV.2504.18649.

[Tru25] Trusted Computing Group. *Trusted Platform Module 2.0 Library Part 0: Introduction*. Version 184. 2025. URL: https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-0-Version-184_pub.pdf (visited on 2025/05/30).

[Van+25] D. Vanoverloop, A. Sánchez, F. Toffalini, F. Piessens, M. Payer, and J. v. Bulck. "TLBlur: Compiler-Assisted Automated Hardening against Controlled Channels on Off-the-Shelf Intel SGX Platforms". In: *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*. USENIX Association, 2025, pp. 1167–1186. URL: https://www.usenix.org/conference/usenixsecurity25/presentation/vanoverloop.

[VBB25] VBB Verkehrsverbund Berlin-Brandenburg GmbH. *Wichtigste VBB-Kennzahlen (Stand: 2025)*. 2025. URL: https://unternehmen.vbb.de/fileadmin/user_upload/VBB/Dokumente/Verkehrsverbund/2025-wichtigste-vbb-kennzahlen.pdf (visited on 2025/06/14).

[Ver+11] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo. "Efficient Byzantine Fault-Tolerance". In: *IEEE Trans. Computers* 62.1 (2011), pp. 16–30. DOI: 10.1109/TC.2011.221.

[VG19] G. Voron and V. Gramoli. "Dispel: Byzantine SMR with Distributed Pipelining". In: *CoRR* abs/1912.10367 (2019).

[Vie+12] M. Vieira, H. Madeira, K. Sachs, and S. Kounev. "Resilience Benchmarking". In: *Resilience Assessment and Evaluation of Computing Systems*. Ed. by K. Wolter, A. Avritzer, M. Vieira, and A. P. A. v. Moorsel. Springer, 2012, pp. 283–301. DOI: 10.1007/978-3-642-29032-9_14.

[Vit+25] R. De Viti, I. Sheff, N. Glaeser, B. Dinis, R. Rodrigues, B. Bhattacharjee, A. Hithnawi, D. Garg, and P. Druschel. "CoVault: Secure, Scalable Analytics of Personal Data". In: *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*. USENIX Association, 2025, pp. 567–587. URL: https://www.usenix.org/conference/usenixsecurity25/presentation/de-viti.

[Wah+24] A. Wahrstätter, J. Ernstberger, A. Yaish, L. Zhou, K. Qin, T. Tsuchiya, S. Steinhorst, D. Svetinovic, N. Christin, M. Barczentewicz, and A. Gervais. "Blockchain Censorship". In: *Proceedings of the ACM on Web Conference 2024, WWW 2024, Singapore, May 13-17, 2024*. ACM, 2024, pp. 1632–1643. DOI: 10.1145/3589334.3645431.

[WAK18] N. Weichbrodt, P.-L. Aublin, and R. Kapitza. "sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves". In: *Proceedings of the 19th International Middleware Conference, Middleware 2018, Rennes, France, December 10-14, 2018*. ACM, 2018, pp. 201–213. DOI: 10.1145/3274808.3274824.

[Wan+19] G. Wang, Z. Jerry Shi, M. Nixon, and S. Han. "SoK: Sharding on Blockchain". In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October 21-23, 2019*. ACM, 2019, pp. 41–61. DOI: 10.1145/3318041.3355457.

[Wan+20] Y. Wang, J. Li, S. Zhao, and F. Yu. "Hybridchain: A Novel Architecture for Confidentiality-Preserving and Performant Permissioned Blockchain Using Trusted Execution Environment". In: *IEEE Access* 8 (2020). DOI: 10.1109/ACCESS.2020.3031889.

[Wan+22] P.-L. Wang, T.-W. Chao, C.-C. Wu, and H.-C. Hsiao. "Tool: An Efficient and Flexible Simulator for Byzantine Fault-Tolerant Protocols". In: *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2022, Baltimore, MD, USA, June 27-30, 2022*. IEEE, 2022, pp. 287–294. DOI: 10.1109/DSN53405.2022.00038.

[Wan+24] B. Wang, S. Liu, H. Dong, X. Wang, W. Xu, J. Zhang, P. Zhong, and Y. Zhang. "Bandle: Asynchronous State Machine Replication Made Efficient". In: *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 2024, pp. 265–280. DOI: 10.1145/3627703.3650091.

[Wei+20] S. Weiser, D. Schrammel, L. Bodner, and R. Spreitzer. "Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. USENIX Association, 2020, pp. 1767–1784. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/weiser.

[WG18] K. Wüst and A. Gervais. "Do you Need a Blockchain?" In: *Crypto Valley Conference on Blockchain Technology, CVCBT 2018, Zug, Switzerland, June 20-22, 2018*. IEEE, 2018, pp. 45–54. DOI: 10.1109/CVCBT.2018.00011.

[Whi+02] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. "An Integrated Experimental Environment for Distributed Systems and Networks". In: *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*. USENIX Association, 2002. URL: http://www.usenix.org/events/osdi02/tech/white.html.

[Wik25a] Wikipedia contributors. *Foreshadow — Wikipedia, The Free Encyclopedia*. 2025. URL: https://en.wikipedia.org/w/index.php?title=Foreshadow&oldid=1304461991 (visited on 2025/08/13).

[Wik25b] Wikipedia contributors. *Hypergeometric distribution — Wikipedia, The Free Encyclopedia*. 2025. URL: https://en.wikipedia.org/w/index.php?title=Hypergeometric_distribution&oldid=1300528082 (visited on 2025/07/21).

[Wil+25] A. Wilde, T. N. Gruel, C. Soriente, and G. Karame. "The Forking Way: When TEEs Meet Consensus". In: *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025. URL: https://www.ndss-symposium.org/ndss-paper/the-forking-way-when-tees-meet-consensus/.

[WJP24] P. Walker, E. Joia, and J. P. A. Pereira. *Central Bank Digital Currencies: Accelerating a Digital Economy with Advanced Technology*. 2024. URL: https://www.microsoft.com/en-us/industry/blog/financial-services/2024/06/20/central-bank-digital-currencies-accelerating-a-digital-economy-with-advanced-technology/ (visited on 2025/08/21).

[WK18] Y. Wang and A. Kogan. "Designing confidentiality-preserving Blockchain-based transaction processing systems". In: *Int. J. Account. Inf. Syst.* 30 (2018), pp. 1–18. DOI: 10.1016/J.ACCINF.2018.06.001.

[WKM24] D. Wong, D. Kolegov, and I. Mikushin. "Beyond the Whitepaper: Where BFT Consensus Protocols Meet Reality". In: *IACR Cryptol. ePrint Arch.* (2024), p. 1242. URL: https://eprint.iacr.org/2024/1242.

[Wu+25] F. Wu, A. Ye, Y. Diao, Y. Zhang, J. Chen, and C. Huang. "TrustChain: A privacy protection smart contract model with Trusted Execution Environment". In: *Blockchain: Research and Applications* (2025). DOI: 10.1016/j.bcra.2025.100296.

[Wüs+22] K. Wüst, K. Kostiainen, N. Delius, and S. Capkun. "Platypus: A Central Bank Digital Currency with Unlinkable Transactions and Privacy-Preserving Regulation". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. ACM, 2022, pp. 2947–2960. DOI: 10.1145/3548606.3560617.

[WZ21] D. Wang and X. Zhang. "Secure Ride-Sharing Services Based on a Consortium Blockchain". In: *IEEE Internet Things J.* 8.4 (2021), pp. 2976–2991. DOI: 10.1109/JIOT.2020.3023920.

[XC07] T. Xu and Y. Cai. "Location anonymity in continuous location-based services". In: *15th ACM International Symposium on Geographic Information Systems, ACM-GIS 2007, November 7-9, 2007, Seattle, Washington, USA, Proceedings*. ACM, 2007, p. 39. DOI: 10.1145/1341012.1341062.

[Xia+20] Y. Xiao, N. Zhang, J. Li, W. Lou, and Y. T. Hou. "PrivacyGuard: Enforcing Private Data Usage Control with Blockchain and Attested Off-Chain Contract Execution". In: *Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14-18, 2020, Proceedings, Part II*. Vol. 12309. Lecture Notes in Computer Science. Springer, 2020, pp. 610–629. DOI: 10.1007/978-3-030-59013-0_30.

[Xie+25] S. Xie, D. Kang, H. Lyu, J. Niu, and M. Sadoghi. "Fides: Scalable Censorship-Resistant DAG Consensus via Trusted Components". In: *CoRR* abs/2501.01062 (2025). DOI: 10.48550/ARXIV.2501.01062.

[Xu+18] W. Xu, S. Rüsch, B. Li, and R. Kapitza. "Hybrid Fault-Tolerant Consensus in Asynchronous and Wireless Embedded Systems". In: *22nd International Conference on Principles of Distributed Systems, OPODIS 2018, December 17-19, 2018, Hong Kong, China*. Vol. 125. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 15:1–15:16. DOI: 10.4230/LIPICS.OPODIS.2018.15.

[Xu21] W. Xu. "Hybrid Fault Tolerant Consensus in Wireless Embedded Systems". Advised by R. Kapitza. PhD thesis. Braunschweig University of Technology, Braunschweig, Germany, 2021. URL: https://publikationsserver.tu-braunschweig.de/receive/dbbs_mods_00069686.

[XXZ18] J. Xue, C. Xu, and Y. Zhang. "Private Blockchain-Based Secure Access Control for Smart Home Systems". In: *KSII Trans. Internet Inf. Syst.* 12.12 (2018), pp. 6057–6078. DOI: 10.3837/TIIS.2018.12.024.

[Yan+20] Y. Yan, C. Wei, X. Guo, X. Lu, X. Zheng, Q. Liu, C. Zhou, X. Song, B. Zhao, H. Zhang, and G. Jiang. "Confidentiality Support over Financial Grade Consortium Blockchain". In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 2020, pp. 2227–2240. DOI: 10.1145/3318464.3386127.

[Yin+03] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. "Separating agreement from execution for byzantine fault tolerant services". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*. ACM, 2003, pp. 253–267. DOI: 10.1145/945445.945470.

[Yin+19] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham. "HotStuff: BFT Consensus with Linearity and Responsiveness". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*. ACM, 2019, pp. 347–356. DOI: 10.1145/3293611.3331591.

[Yu+20] G. Yu, X. Wang, K. Yu, W. Ni, J. A. Zhang, and R. P. Liu. "Survey: Sharding in Blockchains". In: *IEEE Access* 8 (2020), pp. 14155–14181. DOI: 10.1109/ACCESS.2020.2965147.

[Yua+18] R. Yuan, Y. Xia, H. Chen, B. Zang, and J. Xie. "ShadowEth: Private Smart Contract on Public Blockchain". In: *J. Comput. Sci. Technol.* 33.3 (2018), pp. 542–556. DOI: 10.1007/S11390-018-1839-Y.

[YZ22] Y. YuanJiang and J. T. Zhou. "Ticketing System Based On NFT". In: *24th IEEE International Workshop on Multimedia Signal Processing, MMSP 2022, Shanghai, China, September 26-28, 2022*. IEEE, 2022, pp. 1–5. DOI: 10.1109/MMSP55362.2022.9948706.

[Zha+22] Y. Zhang, Y. Zhang, G. Portokalidis, and J. Xu. "Towards Understanding the Runtime Performance of Rust". In: *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, 140:1–140:6. DOI: 10.1145/3551349.3559494.

[Zha+24a] G. Zhang, F. Pan, Y. Mao, S. Tijanic, M. Dang'ana, S. Motepalli, S. Zhang, and H.-A. Jacobsen. "Reaching Consensus in the Byzantine Empire: A Comprehensive Review of BFT Consensus Algorithms". In: *ACM Comput. Surv.* 56.5 (2024), 134:1–134:41. DOI: 10.1145/3636553.

[Zha+24b] G. Zhang, F. Pan, S. Tijanic, and H.-A. Jacobsen. "PrestigeBFT: Revolutionizing View Changes in BFT Consensus Algorithms with Reputation Mechanisms". In: *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 2024, pp. 1930–1943. DOI: 10.1109/ICDE60146.2024.00156.

[Zha+24c] L. Zhao, J. Decouchant, J. K. Liu, Q. Lu, and J. Yu. "Trusted Hardware-Assisted Leaderless Byzantine Fault Tolerance Consensus". In: *IEEE Trans. Dependable Secur. Comput.* 21.6 (2024), pp. 5086–5097. DOI: 10.1109/TDSC.2024.3357521.

[ZMR18] M. Zamani, M. Movahedi, and M. Raykova. "RapidChain: Scaling Blockchain via Full Sharding". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. ACM, 2018, pp. 931–948. DOI: 10.1145/3243734.3243853.

List of Algorithms

List of Definitions, Theorems, and Lemmas

List of Figures

List of Tables