Byzantine Fault-Tolerant State-Machine Replication from a Systems Perspective

TOBIAS DISTLER, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Byzantine fault-tolerant (BFT) state-machine replication makes it possible to design systems that are resilient against arbitrary faults, a requirement considered crucial for an increasing number of use cases such as permissioned blockchains, firewalls, and SCADA systems. Unfortunately, the strong fault-tolerance guarantees provided by BFT replication protocols come at the cost of a high complexity, which is why it is inherently difficult to correctly implement BFT systems in practice. This is all the more true with regard to the plethora of solutions and ideas that have been developed in recent years to improve performance, availability, or resource efficiency. This survey aims at facilitating the task of building BFT systems by presenting an overview of state-of-the-art techniques and analyzing their practical implications, for example, with respect to applicability and composability. In particular, this includes problems that arise in the context of concrete implementations, but which are often times passed over in literature. Starting with an in-depth discussion of the most important architectural building blocks of a BFT system (i.e., clients, agreement protocol, execution stage), the survey then focuses on selected approaches and mechanisms addressing specific tasks such as checkpointing and recovery.

CCS Concepts: • **Computer systems organization → Reliability**.

Additional Key Words and Phrases: Byzantine fault tolerance, state-machine replication

ACM Reference Format:
Tobias Distler. 2020. Byzantine Fault-Tolerant State-Machine Replication from a Systems Perspective. *ACM Comput. Surv.* X, X, Article 0 (November 2020), 37 pages. https://doi.org/10.1145/3436728

1 INTRODUCTION

Recent advances in permissioned blockchains [58, 111, 124], firewalls [20, 56], and SCADA systems [9, 92] have shown that Byzantine fault-tolerant (BFT) state-machine replication [106] is not a concept of solely academic interest, but a problem of high relevance in practice. Especially the ability to withstand arbitrary component failures makes BFT replication a powerful tool in building critical systems that have to meet strong reliability and availability requirements. Unfortunately, developing actual BFT system implementations is everything but trivial. On the one hand, this is a direct consequence of the inherent complexity associated with Byzantine fault tolerance itself. On the other hand, the problem is further complicated by the fact that the research of the last two decades (and before) resulted in a wide spectrum of protocols, mechanisms, and techniques whose applicability and side effects do not always become fully clear based on the original literature. Furthermore, with many publications (understandably) focusing on the new ideas they introduce, it is often difficult to determine whether two approaches interfere or whether they can be combined with each other.

Author's address: Tobias Distler, distler@cs.fau.de, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Martensstraße 1, 91058 Erlangen, Germany.

Purpose of this Survey. The goal of this survey is to answer such questions which typically arise when it comes to actually building BFT systems, thereby hopefully fostering and facilitating further implementations. For this purpose, the survey identifies a BFT system's essential parts and mechanisms and gives an overview of existing solutions to the associated problems. In addition, it elaborates on the individual advantages and disadvantages of each technique in order to support informed design decisions. Where possible and suitable, the discussion is enriched with context information that goes beyond the scopes of the corresponding research papers. In particular, this for example includes a novel abstraction for the comparison of agreement protocols and analyses of the impact specific techniques may have on system resilience and performance, as well as on the effectiveness of other mechanisms. Note that despite the fact that this survey targets BFT systems, some of the observations presented here are likely to also be adaptable to crash-tolerant state-machine replication.

Although primarily focusing on practical issues, this survey is not intended to be a tutorial on how to implement BFT systems. Specifically, there are no discussions on topics such as which programming language to use or how to determine an efficient data format to transmit messages in serialized form. Furthermore, the article also does not contain comprehensive step-by-step descriptions of existing BFT replication protocols; please refer to the original publications for these kind of details. Instead, this survey aims at assisting system researchers and developers in finding solutions to concrete problems such as the extent of the client functionality or the efficient creation of checkpoints, which are addressed in dedicated sections. For clarity, each section only discusses a limited set of representative systems that have been selected due to introducing a novel approach to the respective problem. Different sections therefore concentrate on different systems as the purpose of this paper is to contrast techniques, not to provide an exhaustive comparison of specific BFT systems.

Related Surveys. Approaching Byzantine fault tolerance from a systems perspective, this article complements existing surveys covering other BFT-related topics. Correia et al. [36] concentrate on a specific problem in the context of BFT systems: the necessity to reach consensus in the presence of potentially Byzantine replicas. Their survey analyzes different consensus algorithms at a mainly theoretical level. Berger and Reiser [13] also address Byzantine consensus but focus on techniques to improve scalability, for example, by using efficient cryptographic primitives or communication topologies. Their primary application scenario for BFT consensus are blockchains and distributed ledgers. Platania et al. [95] compare a wide spectrum of BFT protocols based on the role they assign to clients, distinguishing between (1) server-side protocols, in which clients in essence are only a means to access the replicated system, and (2) client-side protocols, in which clients are actively involved in the agreement process. Besides discussing the pros and cons of these two main concepts, the study also includes a detailed analysis of the respective vulnerabilities to a variety of performance and correctness attacks.

Organization. In the remainder of this article, a section providing background on BFT state-machine replication (Section 2) is followed by the actual survey which consists of two main parts. The first part analyzes BFT systems from an architectural perspective. Specifically, this part first gives an overview of BFT system architectures (Section 3) and then individually discusses the major building blocks of such architectures: clients (Section 4), the agreement stage (Section 5), as well as the execution stage (Section 6). Next, the second part of the survey focuses on selected mechanisms and techniques that are either essential for the correctness of a BFT system or effectively improve its performance, availability, or resource efficiency. This includes analyses of different approaches for the creation and management of checkpoints (Section 7), the recovery of replicas from faults (Section 8), the application of optimistic protocols during normal-case operation (Section 9), and the use of trusted subsystems (Section 10). Finally, Section 11 concludes the survey.

2 BYZANTINE FAULT-TOLERANT STATE-MACHINE REPLICATION

This survey focuses on BFT systems that provide clients with access to a stateful application. As illustrated in Figure 1, for fault tolerance the server side of such systems comprises multiple instances of the application, which all implement the same state machine [106] and are commonly referred to as *replicas*. Ideally, for consistency the replica group's behavior would be indistinguishable from the behavior of a single correct server providing the same application [1], but most BFT systems do not go to this length. Instead, they guarantee linearizability [64] by making replicas run an agreement protocol to reach consensus on the state-machine inputs before executing them. Specifically, they establish a total order on all inputs that defines how the inputs have to be processed. Most protocols for this purpose appoint a *leader* replica to propose new sequence-number assignments, while the other replicas act as *followers* and accept them; however, there are also BFT systems that circumvent the election of a leader by relying on randomized protocols [44, 89, 90]. This section summarizes the basic concept of BFT replication and the assumptions typically made regarding its implementation.

Common Assumptions. BFT systems are designed to tolerate a maximum number of simultaneous replica faults; in contrast, there is usually no upper bound on the number of faulty clients. Apart from a few exceptions [31, 80], BFT systems do not give any guarantees on how they behave if the number of replica faults exceeds the predefined threshold, which is why for long-running applications it is essential to have means to recover replicas from faults (see Section 8). Faulty nodes (i.e., clients and replicas) are assumed to fail in arbitrary ways, possibly sending messages with incorrect values, conflicting messages with diverging contents, or no messages at all. Typically, no conjectures are made on the specific origins of faults, meaning that there is no distinction between faulty replica behavior caused by a software bug or replicas failing maliciously as the result of a successful intrusion. If multiple nodes are indeed compromised, they are expected to potentially collude with each other.

The BFT systems studied in this survey assume nodes to be connected via an unreliable network that might drop, corrupt, or delay messages. Having been designed for asynchronous environments, most of these systems are able to ensure safety even if there are no upper bounds on network and processing delays. In contrast, to overcome the FLP impossibility [50] synchronous phases are required to guarantee liveness [45]. As participating nodes usually do not have any knowledge on the occurrence and duration of such phases, a node cannot precisely determine whether another node is faulty or simply advances at a slower speed. To enable the recipient of a message to verify the message's origin, nodes authenticate the messages they send. The specific techniques used for this purpose vary between systems, with most approaches relying on signatures (e.g., RSA [99]) or authenticators [26] (i.e., vectors of message authentication codes (MACs) [118], one for each recipient). Faulty nodes are assumed to not be able to break cryptographic primitives, and consequently they cannot impersonate correct nodes. In addition, BFT systems expect the network to be constructed in a way that prevents faulty nodes from suppressing the communication between any two correct nodes.

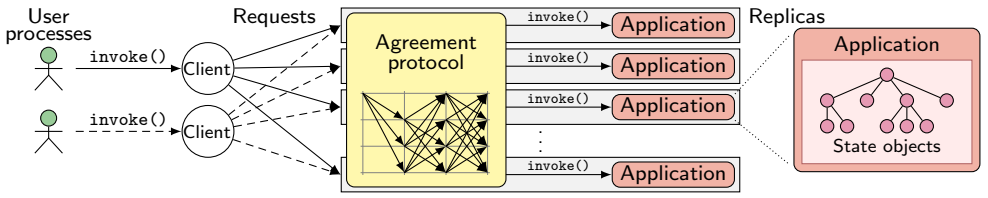


Fig. 1. General overview of BFT state-machine replication. To invoke an operation, a user process relies on a local client library to send a request to the server side. There, replicas run an agreement protocol to establish an order on incoming state-machine inputs and then execute them in the stateful application.

The choice of authentication scheme not only affects computation overhead (i.e., signatures are typically more costly than MACs) but often also has an impact on protocol design, because signatures provide non-repudiation whereas MAC authenticators do not. Without this property, the fact that a correct recipient has validated the authenticity of a message does not automatically mean that other correct recipients will be able to do the same. In general this poses a problem since without counter measures malicious nodes could use the weaker guarantees provided by authenticators to cause inconsistent decisions (e.g., regarding the question whether a client request should be accepted or discarded [33]). One way to address this issue is to combine a signature with a MAC and only check the signature if the MAC is valid and the received message contains relevant information. At the (comparably low) cost of an additional MAC per message, this ensures non-repudiation while minimizing the overhead for unnecessary signature validations [33].

Stateful Application. The state of the replicated application is often modeled as a collection of disjoint *objects* storing all relevant information [26, 28, 41, 46]. The actual content of a state object is typically application specific. Among other things, a state object for example may represent a single file or directory in a file system [26], a data record in a key–value store [46], or a block in a ledger [111]. As further discussed in Section 7, organizing an application's state as a composition of objects makes it possible to efficiently take and synchronize state snapshots. In addition to state objects, the application defines a set of commands that operate on these objects and can be invoked by clients issuing a corresponding request. Once the execution of a command is complete, the application produces a result and forwards it to the client. The particular information contained in a result in general varies between commands. While some results carry a value, others are empty and simply signal the completion of the operation. Depending on a command's effect on the application state, many BFT systems divide commands into two categories: *writes* and *reads* [26]. A write command accesses one or more state objects and potentially modifies at least one of them. Apart from updating object content this also includes the possibility of a write creating new objects or deleting existing ones. Hence, the object composition of the application state may vary over the course of a system's lifetime. In contrast to writes, a read command only retrieves the content of one or more state objects, but in any event leaves the application state unchanged. The distinction between writes and reads is useful as it, for example, offers the opportunity to improve performance by designing a fast path for non-modifying operations, as further discussed in Section 6.2.

Replicas. The minimum size of the replica group differs between BFT systems and depends on the maximum number f of concurrent replica failures a system must be able to tolerate; f is a constant selected prior to system deployment. Given this prerequisites, a BFT system tolerates faults by first processing a request on multiple replicas and then comparing the corresponding results determined by the replicas involved. Once $f + 1$ different replicas have delivered the same opinion on a result, a client considers this result to be correct because, in the presence of at most f faults, at least one of the opinions must have been produced by a non-faulty replica and therefore by definition be correct [26]. This approach follows a general strategy applied throughout all parts of a BFT system. For safety, correct clients and replicas only make irreversible decisions based on *stable* evidence, that is proofs of correctness that are guaranteed to be backed up by a sufficient number of non-faulty replicas.

With the resilience of a BFT system being based on the opinions of different replicas, it is essential that a single fault does not lead to the failure of multiple replicas. One approach to reduce the probability of such common-mode failures is to diversify replica implementations [8, 51], for example based on off-the-shelf software components [28, 53–55, 57]. Another effective technique to make dependent faults less likely is to place replicas at different geographic locations and (possibly) within different administrative domains [14, 47, 110, 122].

Checkpointing. Having only a certain amount of memory at their disposal, replicas solely keep track of a limited window of active consensus instances. The size of this window varies between systems and ranges from two [112] to a configurable constant [26]. To garbage-collect information on completed consensus instances, replicas create and store periodic checkpoints of their state (see Section 7). Apart from application objects, a replica's state also includes replication-protocol data such as the following: (1) *A set of request ids that for each client contains the id of the latest request the replica has processed.* With the request ids of each client monotonically increasing, the set allows a replica to efficiently detect and discard old requests. Among other things, this becomes necessary as many BFT agreement protocols do not have a built-in mechanism for duplicate suppression. (2) *A collection of the latest reply sent to each client.* The purpose of this reply store is to enable replicas to retransmit replies and thereby tolerate scenarios in which the unreliable network drops messages. The limitation to one stored reply per client is a direct consequence of the fact that BFT systems usually assume each correct client to have at most one pending request at a time [26].

3 SYSTEM ARCHITECTURES

Although BFT state-machine replication puts specific requirements on the properties of algorithms and protocols, the concept on the other hand also offers some flexibility with regard to the design and structure of systems. This section presents an overview of basic system architectures, thereby focusing on how they assign the responsibilities of performing three important server-side tasks: (1) *Client Handling.* To make the replicated service accessible, replicas need to receive and disseminate incoming requests from clients and, once a request has been processed, return a reply containing the corresponding result. Due to the fact that requests and replies not only have to be transmitted but also authenticated, this exchange of messages between clients and replicas typically involves a significant amount of both network and processing resources. The size of this overhead in general depends on the number of clients that are currently active in the system and the workload they produce. (2) *Agreement.* To ensure consistency, replicas must reach consensus on the decisions they make. The amount and size of the authenticated messages required for this purpose usually varies with the workload, as it does for requests and replies. However, in contrast to the client–replica interaction, the communication between replicas in most use cases is limited to a lower and constant number of participants, and therefore less costly. (3) *Execution.* After replicas have agreed on a request, they process the request based on the current application state. The performance and resource overhead for this task highly depends on the application and may range from being negligible compared with the other tasks (e.g., when the request only increments an in-memory counter) to being the dominant factor in the system (e.g., when the execution of a request takes several seconds or even minutes).

One-Tier Architecture. As illustrated in Figure 2a, the most common approach to BFT system design is to model the server side as a group of monolithic replicas that combine the logic for all three of the tasks listed above [26]. Performing all tasks within the same process, this one-tier architecture enables a system to efficiently forward information between client-handling stage and agreement stage, as well as between agreement stage and execution stage. On the downside, the co-location of stages makes it necessary for each replica to be equipped with enough resources to run the entire server-side protocol and therefore entails the risk of bottlenecks in use-case scenarios with a large number of active clients and/or resource-intensive applications. Despite the restriction to use monolithic replicas, the one-tier architecture still offers the opportunity to apply agreement protocols with different replica-interaction patterns. As a result, the spectrum of proposed one-tier BFT architectures ranges from systems with all-to-all replica communication [26] to systems that organize replicas in the form of a chain [119].

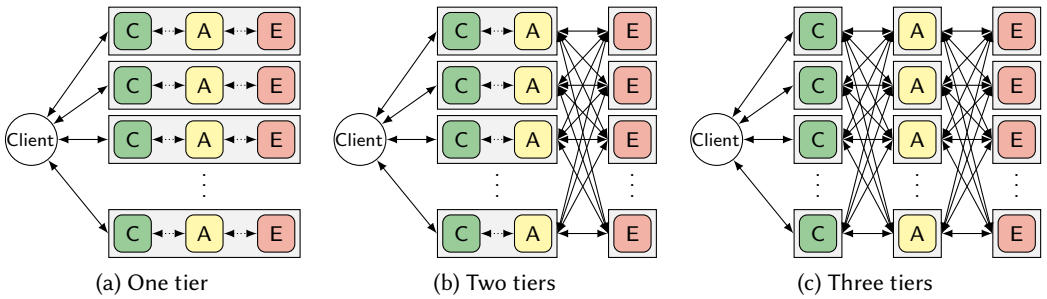


Fig. 2. Basic BFT system architectures. The three main server-side stages client handling (C), agreement (A), and execution (E) may be distributed across one, two, or three groups of processes. More processes facilitate the effective use of additional machines, but also increase the communication overhead between stages.

Two-Tier Architecture. In the two-tier architecture (see Figure 2b), the responsibilities of each replica are split up between two loosely coupled processes, with one process typically hosting the stages for client interaction and agreement, while the other process comprises the execution stage [126]. As a main benefit, this separation creates the possibility of distributing the server side of a BFT system across additional machines, thereby minimizing the probability of performance bottlenecks. This comes at the cost of an increased communication overhead as the interaction between agreement and execution stage now involves the exchange of authenticated messages over a network. Furthermore, it introduces an additional problem with regard to how exactly the agreement stage proves to the execution stage that the consensus process for a request has completed successfully. One option is to allow agreement processes to directly forward such proof in the form of agreement messages to the execution stage [126]. However, this approach makes it necessary for execution processes to be able to verify the correctness of agreement-stage-internal communication. Therefore, if for example the agreement stage relies on MAC authenticators to protect messages [26], the authenticators of agreement messages have to be extended with an additional entry for each execution process. Alternatively, agreement messages can remain small if the execution processes themselves are responsible for assembling the required proof by obtaining a sufficient number of dedicated completion confirmations from different agreement processes [48]. The latter approach also has the advantage that the execution stage does not need any knowledge on the specifics of the protocol used in the agreement stage to reach consensus. Section 6.1 analyzes the separation of agreement and execution in more detail, discussing additional implications of this concept.

Three-Tier Architecture. In BFT systems that are based on the three-tier architecture, each of the three main server-side tasks (i.e., client handling, agreement, and execution) is handled by a separate group of processes (see Figure 2c). This makes it possible to free the agreement stage from the need to spend resources on the potentially costly communication with clients. Furthermore, it enables system designs that further improve agreement-stage efficiency by routing large requests directly from the client-handling stage to the execution stage [32]. More specifically, the agreement stage in such case only sees a hash of the actual request, which it delivers to the execution stage once the consensus process is complete. Using this hash, the execution stage is then able to fetch the full version of the request from the client-handling stage. With the three-tier architecture being an extension of the two-tier architecture, it further increases network traffic and apart from that shares the same difficulties with regard to the separation of agreement and execution.

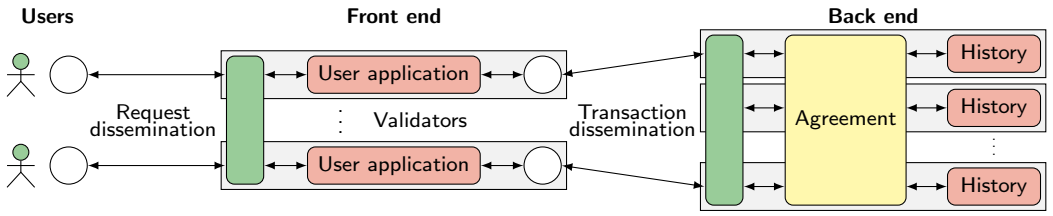


Fig. 3. Composite architecture consisting of a user-facing front-end application and a history back end.

Composite Architectures. Several BFT systems that have been developed in recent years follow the *execute-verify* principle [70], especially in the domain of permissioned blockchains [4, 10, 58]. Many researchers argue that such a design represents a new type of system architecture [4, 70] because compared with the architectures in Figure 2 the tasks of agreement and execution are performed in different order: first the system produces the application outputs and state updates, then it establishes an order on them using a consensus protocol.

Another way to approach these systems is to think of them as a composition of two applications, as shown in Figure 3: (1) a user-facing front-end application that comprises the service-specific logic and (2) a back-end application that verifies and manages state information/history in a ledger [4, 10] or key-value store [58] using a traditional BFT architecture. In this system view, the front-end nodes, which are often referred to as *validators* [10, 23], essentially are BFT clients of the back-end application that themselves maintain a consistent state copy of the user-facing application. When executing a user request, front-end nodes at first only determine the effect the request would have on the front-end application, but they do not directly update their state. Instead, they create a transaction representing the corresponding state modification and instruct the back end to append the transaction to the system's history. Once the back end confirms that the transaction has been agreed on and stored, front-end nodes are allowed to execute the transaction and persist its effects. To tolerate faults in the front end, transactions typically need to be certified by multiple front-end nodes in order to be considered by the back end. To ensure that this is the case the back-end application is not an off-the-shelf storage service but also comprises mechanisms to check the validity of transaction certificates. Still, the back end does not require knowledge on the front-end application itself.

Structuring a system in the described manner has several benefits: (1) With the back end running the consensus process, there is no need for an additional agreement protocol in the front end. As a consequence, it for example becomes easier to achieve scalability by adding front-end nodes. (2) Due to the back end not requiring information on the internals of the front-end application, this part of the system can be implemented in a generic way. This property enables platforms such as Hyperledger Fabric [4] to support different agreement protocols. (3) Since the front-end nodes themselves represent BFT clients, they have proof that the back-end decisions they observe are final and cannot be changed in the future. As pointed out by Bessani et al. [16], this is not the case for individual replicas hosting the user-facing application in traditional BFT system architectures.

4 CLIENTS

Providing an interface to access the replicated application, clients are essential parts of every BFT architecture whose responsibilities in many cases go beyond the task of simply relaying requests and replies between user processes and the server side. This section gives an overview of the main tasks clients are charged with in BFT protocols, thereby illustrating how significantly the client role and its associated complexity can differ between individual systems. Table 1 shows a summary of this analysis based on a selection of representative systems.

Communication with Replicas. The principal service offered by clients in all BFT systems is to handle the interaction with replicas by exchanging request and reply messages over a network. With the network being unreliable, this usually involves the responsibility for taking care of tasks such as the retransmission of messages or the reestablishment of broken connections. The specific protocols used for the client–replica interaction vary between systems, however they all ensure that faulty replicas are unable to prevent the requests of correct clients from being executed. For this purpose, these protocols commonly apply one of the two following general approaches (see Section 9.1 for details): (1) *Request Multicast.* After a user process invokes a command, clients implementing this strategy immediately submit the corresponding request to all replicas in the system that are responsible for handling it [18, 27, 121]. Consequently, non-faulty replicas directly learn about the new request and are therefore, for example, able to detect situations in which the leader tries to suppress a particular client request by not including it in a subsequent proposal. (2) *Use of a Contact Replica.* Relying on this optimistic approach, a client at first only submits its request to a single contact replica, typically the current leader, which in the fault-free case then takes care of further distributing the message to other replicas [26, 40, 67, 72]. To tolerate contact-replica failures, the client in addition sets up a timeout that is canceled once it obtains a stable result. If, on the other hand, this timeout expires, the client resorts to the first strategy of multicasting the request.

The Troxy approach [78] has shown that it is possible to design BFT systems in which the communication with replicas is the only functionality clients need to provide. To achieve this, Troxy introduces trusted proxy components at the server side, integrated into each replica, with which clients interact through secure channels. Due to the trusted proxies handling common client tasks such as result verification, it is sufficient for the actual clients to focus on the exchange of request and reply messages with the proxy. The need for a trusted proxy can be circumvented if the server side protects its messages with threshold signatures [22, 108], that is signatures that are created using inputs from multiple replicas. Among other things, SBFT [58], for example, uses this concept to reduce the communication overhead between clients and replicas during benign conditions.

Result Verification. In contrast to clients in crash-tolerant systems, BFT-system clients in general must not automatically accept the first reply they receive from the server side, because if a replica is faulty its reply may comprise an arbitrary result. To address this problem, a client usually waits for replies from different replicas and then compares them to verify the result before delivering it to the local user process. The number of matching replies necessary for this purpose depends on the specific requirements for a result. In the common case, for example, $f + 1$ matching replies from different replicas are needed to ensure the correctness of a result value in the presence of at most f replica failures [26]. If the corresponding requests take an optimistic fast path through the system and there are additional demands with regard to consistency, the stability threshold may increase to $2f + 1$ (e.g., read-only or tentative replies in PBFT [26]) or even $3f + 1$ (e.g., speculative replies in Zyzzyva [72]). In this context, it is important to note that result verification does not necessarily have to be performed on full replies but can also involve hashes of the result. Section 9.1 discusses this optimization in detail.

Table 1. Comparison of client responsibilities in different BFT systems.

	Troxy [78]	**PBFT** [26]	**Zyzzyva** [72]	**Abstract** [6]	**Archer** [47]
Communication with replicas	✓	✓	✓	✓	✓
Result verification		✓	✓	✓	✓
Detection of replica inconsistencies			✓	✓	
Resolution of replica inconsistencies				✓	
Provision of optimization hints					✓

Detection of Replica Inconsistencies. When performing result verification by comparing replies, clients often are the first to detect inconsistencies between replicas. If such inconsistencies by design can only stem from faulty replicas, clients typically ignore them as the BFT replication protocol guarantees that they will eventually receive a sufficient number of matching replies [2]. However, there are also BFT system designs that deliberately allow correct replicas to temporarily deviate and then rely on clients to trigger mechanisms for reestablishing consistency [6, 72]. Specifically, correct clients must inform the server side about specific inconsistencies they observe. In Zyzzyva [72], for example, a faulty leader can make conflicting proposals that cause the states of correct follower replicas to diverge without these replicas having means to notice the discrepancies themselves. Zyzzyva replicas solve this problem by including information about the order of executed requests in their replies to clients, thereby enabling clients to determine inconsistencies as a byproduct of the result verification process. If a client becomes aware of conflicting order information, the replies serve as a proof of the current leader's misbehavior, which the client can then distribute among follower replicas to convince them to abandon the leader, elect a new leader, and if necessary perform a roll-back of their states in order to repair the existing inconsistencies.

Resolution of Replica Inconsistencies. As discussed above, in Zyzzyva clients are used for the detection of diverging replicas, however at the end the replicas are the ones in charge of fixing the problem. Abstract [6] in this regard follows a similar idea but goes one step further: it also involves clients in the actual resolution of replica inconsistencies. Abstract's underlying concept is to design a BFT system as a composition of specialized replication protocols of which only a single one is active at any given time. The switching procedure that enables the system to transition from one protocol to another is standardized and heavily relies on clients to relay knowledge about the system's history. More precisely, when the current BFT protocol in Abstract aborts, replicas stop their efforts to reach consensus and respond to new client requests by returning details about the state of the agreement process, which among other things include the sequence of requests that have committed up to this point. Having collected such history information from multiple replicas, a client can subsequently use it to initialize the next BFT protocol. Due to histories being protected by digital signatures, faulty clients are unable to trick replicas into accepting arbitrary data as basis for the new protocol. However, they still may leave the system in an inconsistent state by failing to distribute the histories. On the other hand, a single active correct client in Abstract is sufficient to complete the protocol switch and enable the system to make progress again.

Provision of Optimization Hints. Apart from performing tasks that directly contribute to the functionality of the overall system, clients can also be charged with responsibilities affecting non-functional properties. Archer [47], an optimization technique for geo-replicated BFT systems, for example, exploits clients to assist in dynamically finding system configurations that minimize end-to-end response times. The approach is based on the insight that if replicas are distributed across the globe, the position of the leader can have a significant impact on latency [110]. Consequently, Archer aims at determining the leader location that offers the best performance for the current workload conditions. With clients being the only components in the system that are actually able to observe end-to-end response times, Archer relies on them to periodically perform a series of latency measurements to evaluate different leader locations. Whenever a client has completed such measurements, it sends the obtained results attached to its next request to the server side, where the measurement results then are deterministically analyzed and used to select the most suitable leader. In general, mechanisms based on client-provided hints should always include counter measures against Byzantine clients trying to exploit the mechanisms to their advantage. Archer deals with such issues by combining the observations of many clients before reaching a decision, thereby limiting the effect a subset of faulty clients can have on the outcome of the leader selection.

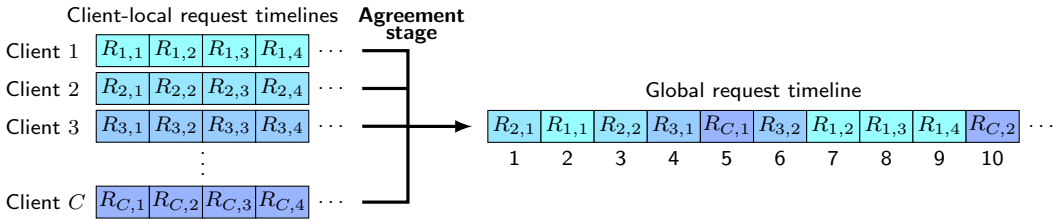


Fig. 4. Overview of the functionality of a BFT agreement stage. Using the client-local request sequences as input, the agreement stage produces a global totally ordered request sequence. That is, from an abstract point of view, the agreement stage merges multiple unstable timelines into a single stable timeline.

5 AGREEMENT STAGE

The purpose of a BFT system's agreement stage is to establish a global and stable total order on incoming client requests, which then serves as a guideline for the execution stage to determine how to apply the requests to the replicated state machine. This section presents a comparison of the different approaches used in BFT systems to solve this problem. In particular, existing solutions differ in the number of intermediate steps involved in ultimately producing the total order, and in the representation of state-machine inputs the actual consensus is performed on.

5.1 Agreement-Stage Building Blocks

To analyze the conceptual differences between BFT agreement stages, it is useful to approach the associated problem from an abstract perspective. As illustrated in Figure 4, every client assigns a unique (logical or physical) timestamp to each submitted request and thereby establishes a local order on its own state-machine inputs. Based on these inputs, it is then the responsibility of the agreement stage to combine the client-specific timelines into a global timeline that all correct replicas in the system adhere to. If all clients were correct and would issue requests at the exact same frequency, achieving this goal would be straightforward as replicas simply could select inputs from client timelines in a round-robin fashion. However, in the presence of faulty clients this approach is infeasible as, for example, a single client could cause the system progress to stall indefinitely. As a consequence, more sophisticated protocols are necessary to reliably agree on state-machine inputs. Typically, the design of such protocols utilizes the fact that there are essentially two ways for a BFT system to maintain information about state-machine inputs:

- *Unstable Timeline ($\mathcal{U}$).* This type of timeline originates from a source that (1) at any time may cease to provide other nodes in the system with information on the inputs contained in this timeline, and (2) possibly expresses conflicting opinions on the timeline's contents to different nodes. The textbook example for an unstable timeline is the sequence of requests submitted by a client that potentially is subject to Byzantine faults.
- *Stable Timeline ($\mathcal{S}$).* In contrast to unstable timelines, the source of a stable timeline is not only a single node but instead a group of nodes with a sufficient number of correct members to ensure that the information on inputs always remains available. Furthermore, it is guaranteed that there are enough correct source nodes providing the same opinions in order for an observer to verify that it has correctly learned the timeline's contents. The textbook example for a stable timeline is the distributed sequence of inputs produced by a BFT agreement stage.

Taking these two types of timelines into account, the problem that needs to be solved by the agreement stage of a BFT system can be described as follows. The agreement stage must transform

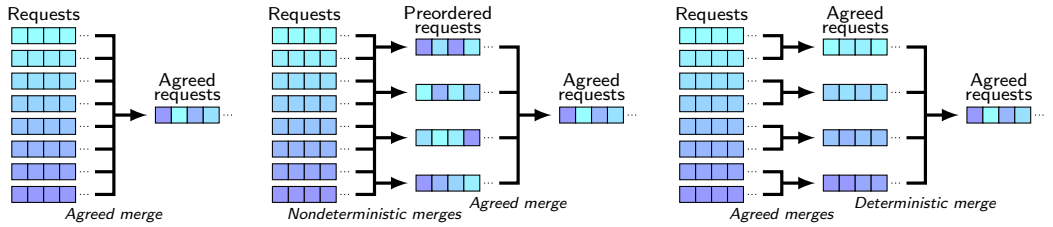
a set of unstable client timelines $\vec{\mathcal{U}}$ into a single stable execution-stage timeline $\mathcal{S}$. This task can be performed in various ways and may be divided into multiple timeline-merging steps. Overall, there are three main building blocks available for the design of an agreement stage:

- *Nondeterministic Merge ($NM : \vec{\mathcal{U}} \rightarrow \mathcal{U}$).* This building block takes a set of unstable timelines and merges them into a combined unstable timeline. An example for such a mechanism is the preordering subprotocol of Prime [2] in which replicas independently of each other assign local sequence numbers to incoming client requests. Without coordination among replicas and due to clients potentially being faulty, the resulting replica-local request sequences are very likely to differ across replicas. Furthermore, as replicas themselves may be faulty and (initially) are the only ones to know their own sequence, the outcome of this nondeterministic merge of unstable timelines once again is an unstable timeline. Due to their simplicity, nondeterministic merges typically do not result in significant performance or resource overhead.
- *Agreed Merge ($AM : \vec{\mathcal{U}} \rightarrow \mathcal{S}$).* Transforming a set of unstable timelines into a stable timeline, agreed merges are essential for BFT state-machine replication and therefore implemented by a wide spectrum of agreement protocols [26, 86, 112]. Specifically, these protocols ensure that all correct replicas reach consensus on the exact interleaving of the unstable input timelines. To prevent starvation, agreement protocols usually guarantee that all contents of an input timeline (e.g., the requests of a specific client) will eventually be included in the resulting stable timeline, provided that the source of the unstable input timeline remains available and behaves according to specification (as a correct client for example does); inputs from faulty sources on the other hand might be ignored. Taking into account the powerful guarantees they offer, agreed merges constitute one of the most complex parts of a BFT system and in general require multiple phases of network communication.
- *Deterministic Merge ($DM : \vec{\mathcal{S}} \rightarrow \mathcal{S}$).* Given a set of stable timelines as input, this building block combines them and outputs a single stable timeline. For the correctness of this procedure, it is essential that the merge algorithm is deterministic and known to all correct replicas, for example due to being configured prior to the start of the system. Examples for BFT systems relying on deterministic merges are COP [11] and Hybster [12], which both perform the agreement on client requests in parallel groups whose outputs are then deterministically merged based on a round-robin scheme. Similar mechanisms also have been proposed to improve scalability in crash-tolerant replicated systems [39, 69, 82, 83, 105]. The fact that the input already consists of stable timelines usually makes it possible to efficiently implement deterministic merges. The major challenge in this context therefore is to ensure that the system does not block if one or more input timelines currently have no new content. A possible solution to address this problem is to fill unused input slots with agreed no-ops [48].

Considering the inputs and outputs of the three building blocks, a BFT agreement stage can be regarded as a chain of merge operations. Adding the concept of partitioning, and thereby allowing merges on disjoint input subsets to take place in parallel, this chain has the following pattern:

$$\vec{\mathcal{U}}_{client} \xrightarrow{\vec{NM}} \ldots \xrightarrow{\vec{NM}} \vec{\mathcal{U}}_{replica} \xrightarrow{\vec{AG}} \vec{\mathcal{S}}_{agreed} \xrightarrow{\vec{DM}} \ldots \xrightarrow{\vec{DM}} \mathcal{S}_{execution}$$

First, a series of nondeterministic merges transforms the set of unstable client-request timelines $\vec{\mathcal{U}}_{client}$ into one or more unstable timelines $\vec{\mathcal{U}}_{replica}$ that are managed at the server side. Next, the agreement protocol produces a set of stable intermediate timelines $\vec{\mathcal{S}}_{agreed}$, which are then deterministically merged into the final stable timeline $\mathcal{S}_{execution}$.



(a) PBFT [26] (b) Prime [2] (c) COP [11]

Fig. 5. Comparison of different agreement stages. Like most BFT systems, PBFT handles the agreement in a single step. In contrast, Prime preorders requests before agreeing on them. COP also relies on a two-step approach by first partitioning the consensus and then deterministically merging the outcomes of all partitions.

As illustrated in Figure 5, existing BFT agreement stages so far do not fully exploit the potential of being able to integrate a variety of building blocks and consequently comprise only short chains. In PBFT [26], for example, the agreement stage directly operates on the client timelines and creates the execution-stage timeline in a single agreed-merge step that involves three phases of message exchange between replicas (see Figure 5a). The agreement stage of Prime [2] in contrast is composed of two blocks: a nondeterministic merge ("preordering") followed by an agreed merge. As shown in Figure 5b, clients in Prime submit their requests to multiple replicas, which then insert them into unstable and replica-specific timelines that serve as basis for the subsequent agreement process afterwards. Performing the agreement on replica timelines instead of client timelines on the one hand results in additional overhead, but on the other hand has one major advantage. With the replica timelines containing the requests of different clients, a faulty leader loses the ability to discriminate against specific clients, for example, by proposing their requests with significant delay. Of course, a faulty leader can still delay the progress of selected replica timelines, however as each request is part of multiple (possibly all) replica timelines this has no negative effect on the requests of a specific client.

The agreement stage of COP [11] also comprises two building blocks; in this case an agreed merge is performed prior to a deterministic merge. Specifically, COP relies on a set of largely independent agreement groups that each are responsible for establishing a stable order on a part of the incoming client requests. In a second step, these agreed timelines are then deterministically merged into the resulting timeline used by the execution. Relying on multiple agreement groups has the benefit of making it possible to parallelize the agreement process, for example by exploiting several cores per replica [11, 12, 79] or extending the agreement stage to a larger number of machines [48]. The effectiveness of the parallelization is tied to a BFT system's ability to distribute the request workload across agreement groups to ensure that all groups advance their timelines at about the same speed, thereby supporting an efficient deterministic merge at the end.

The timeline-centric view on a BFT system's agreement stage used in this section complements the single-consensus view applied in the vast majority of existing publications (e.g., [26, 33, 40, 123]). Rather than only considering the steps necessary for replicas to reach agreement on a single request, the notion of timelines focuses on the relationship between requests of different sources. This approach is particularly beneficial for BFT-system designers that prefer to use an existing agreement algorithm as a black box instead of developing a new algorithm from scratch. Furthermore, the timeline-centric view has the advantage of making it easier to reason about whether existing approaches can be combined or not and, if so, how this could be done. Operating at different positions in the merge chain, Prime's nondeterministic preordering mechanism, for example, may be integrated with COP's parallelized agreement by applying the mechanism to different subsets of client requests.

5.2 Representations of State-Machine Inputs

In order to remain consistent, correct replicas in a BFT system need to agree on a common total order on state-machine inputs. However, this does not necessarily mean that the actual consensus has to be performed on client requests, as the following discussion of input representations shows.

Full Requests. The straightforward approach is to directly include the full client requests in agreement messages and thereby ensure that there cannot be any discrepancies between an agreement stage's data flow and its control flow. That is, if a replica first learns about a specific client request through a valid agreement-protocol message, it is also guaranteed to obtain the corresponding request. As a variant of this approach, it is possible to combine multiple requests into a batch and then agree on this batch within a single consensus instance [52]. Either way, exchanging full requests as part of agreement messages typically has the drawback of significantly increasing message sizes and therefore leading to considerable performance and resource overhead.

Request Hashes. In an effort to minimize the size of agreement messages, BFT systems usually perform the agreement on request hashes, not the full client requests [27]. Although in general more efficient, this approach introduces additional requirements that can be easily overlooked and often increase protocol complexity. In particular, follower replicas are now unable to directly confirm a leader's proposal based on the information contained in the message alone. Instead, a follower first needs to wait until having obtained the full request from a separate source (e.g., the client itself) before being able to send the confirmation. This means that in order to make a decision the agreement stage in this case requires external information from other stages, for example the client-handling stage (see Section 3) if clients directly submit full requests to all replicas in the system [27]. Faulty clients may use this property to their advantage and provide the leader with a different version of the request than the followers, or the followers with no request at all, thereby causing a situation in which correct followers at first might not be able to support the proposal of a correct leader [109]. Resolving such conflicts is possible (e.g., by allowing followers to retroactively fetch the full version of the proposed request from the leader) but involves more complex fault-handling mechanisms. Furthermore, it takes valuable time during which the system cannot make progress. The same is true when agreement messages carry batches of request hashes. Here, a single missing or diverging request from a faulty client can prevent a follower from confirming the entire batch proposal, thereby directly affecting requests submitted by correct clients.

Progress Vectors. Prime [2] shows that a BFT agreement stage can establish a total order on client requests without the consensus protocol seeing the actual requests or their hashes. As discussed in Section 5.1, replicas in Prime preorder incoming client requests, that is, they insert the requests into replica-local timelines and ensure that the requests' contents and assignments to preorder sequence numbers are known to other replicas. With information on replica timelines already being reliably distributed during the preorder step, the consensus protocol in Prime is able to perform the agreement process based on progress vectors, which for each replica timeline contain the highest preordered sequence number. By computing the differences between two consecutive progress vectors in a statically defined manner, the agreement stage can later deterministically translate the sequence of agreed progress vectors into a sequence of client requests. Compared with the traditional approach of agreeing on requests or their hashes, the use of progress vectors offers the major benefit that the size of consensus messages is independent of the number of client requests that replicas agree on within a single consensus instance. That is, the number of elements in the progress vector (i.e., the number of replicas that preorder requests) is the same for a single request or a million requests and therefore throughput independent. This also implies that batching progress vectors is not particularly effective as the approach already performs an inherent batching.

Application Outputs. As discussed in Section 3, composite BFT systems usually first execute client requests in a user-facing application and then use the effects these requests had on the application as inputs for the replicated state machine [4, 24, 70, 111]. Consequently, in such cases the consensus, for example, is performed on state updates and their dependencies [4, 111] or hashes covering both the resulting application state as well as the produced replies [24, 70]. From a consensus perspective, this approach is not significantly different from the agreement on requests or request hashes, however there is one important exception. If the system design allows correct replicas to temporarily have diverging opinions on states and outputs, for example as a result of nondeterministic execution [70], there might be situations in which the agreement stage at first is not able to reach consensus. To ensure eventual system progress under such conditions, replicas must be equipped with a mechanism that enables them to roll back their states and retry request execution, this time for example in a deterministic manner guaranteeing that all correct replicas end up in a consistent state.

6 EXECUTION STAGE

Hosting multiple instances of the service application, the execution stage of a BFT system is responsible for processing client requests and producing the corresponding replies. Given these characteristics, large parts of the execution stage are typically application specific, which is why several works developed approaches to cleanly separate them from the application-independent agreement stage. Apart from such a modularization, most research efforts aimed at preventing the execution stage from becoming a performance bottleneck. This section examines both of these aspects in detail.

6.1 Separate Execution Replicas

Traditionally, replicas in BFT systems participate in both the agreement as well as the execution of client requests [26]. However, as pointed out by Yin et al. [126], it is also possible to separate the responsibilities for the two tasks and implement them in dedicated processes. Several BFT systems have built on this insight as the following examples illustrate. SAfE [126] relies on separate clusters for agreement and execution to provide a privacy firewall that prevents faulty execution replicas (which for this purpose need to run in an isolated environment) from leaking confidential application state to clients. UpRight [32] employs a three-tier system architecture (see Section 3) to enable large requests to bypass the consenus protocol. VM-FIT [98], Spare [42], and ZZ [125] move the application logic into virtual machines and thereby facilitate essential tasks such as the startup, maintenance, and recovery of replicas. Other systems like COP [11] and Omada [48] leverage the fact that the separation significantly simplifies the parallelization of the agreement process.

In addition to these advantages, running separate execution replicas also has the benefit of being able to reduce the number of application instances required for fault tolerance. Specifically, while in the general case at least $3f + 1$ replicas are necessary for the agreement of requests in the presence of at most f faults, $2f + 1$ replicas are sufficient for the execution of requests [126]. With the application usually being one of the most complex parts of a BFT system, a reduction from $3f + 1$ to $2f + 1$ application instances has the potential to lead to major cost savings both before (e.g., development of fewer independent application implementations [8]) and after (e.g., less processing and network resource usage) deployment. While these benefits are generally recognized, the use of $2f+1$ execution replicas also comes with unique subtleties that are sometimes overlooked but can have a decisive impact on the guarantees and performance a BFT system is able to provide. Similar observations can be made with respect to BFT systems that do not separate agreement from execution but use a minimum of $2f+1$ replicas due to relying on trusted components (see Section 10).

State-Transfer Requirements. Comprising only $2f+1$ execution replicas, in order to ensure liveness a BFT system must be designed to make progress (i.e., produce stable results and checkpoints)

based on the active participation of a minimum of $f + 1$ execution replicas. As individual replicas may advance at different speeds, this can lead to scenarios in which up to f execution replicas trail so far behind that they miss requests that have already been garbage-collected by other replicas and therefore require a checkpoint to catch up. Due to the fact that of the $f + 1$ execution replicas that contributed to the latest stable checkpoint up to f may be faulty, a system's state-transfer protocol in such case must enable trailing replicas to verify a checkpoint based on information provided by a single correct source. For this purpose, the proof of checkpoint stability has to be constructed in a way that (1) the proof contains knowledge about the contents of the checkpoint and (2) it is ensured that if one correct replica considers the proof to be valid, all other correct replicas will do the same. The common approach to achieve this is to assemble a checkpoint certificate consisting of $f + 1$ matching checkpoint messages from different replicas that include a hash of the checkpointed state and are protected by digital signatures [126]. Note that the two requirements significantly limit the spectrum of possible design options compared with BFT systems comprising $3f + 1$ execution replicas. There, checkpoints may become stable based on $2f + 1$ execution replicas, of which at least $f + 1$ are correct. Relying on these $f + 1$ correct replicas alone, trailing replicas afterwards are able to retroactively assemble a proof of correctness for a checkpoint, even if no such proof exists yet at the beginning of state transfer. This property offers additional flexibility with regard to the question when and how replicas exchange information about a checkpoint's content. Furthermore, it puts fewer restrictions on the authentication scheme applied for checkpoint messages and, for example, enables the use of MAC-based authenticators for checkpoint certificates [26].

Result Availability Limitations. Reducing the number of execution replicas from $3f + 1$ to $2f + 1$ without taking additional measures, a BFT system usually is no longer able to guarantee that correct clients will eventually obtain stable results to all their requests. This is best illustrated by the following scenario, which might occur as a direct consequence of the state-transfer example discussed above. If f of the correct execution replicas fall behind and apply a checkpoint to bring their states up to date, they thereby skip the execution of requests with sequence numbers between their old state and the checkpoint. This means that the replies to the skipped requests are only known by the remaining $f + 1$ execution replicas, of which f may be faulty and fail to correctly respond to the client. Consequently, a client could be left with a single reply and therefore indefinitely unable to verify the correctness of the result. BFT systems that address this issue, for example, solve the problem by extending checkpoints to include (a hash of) the latest reply to each client [41, 48]. This way, when a correct replica loads a verified checkpoint, it also learns the correct outcomes of all skipped requests that still might be considered pending from a client perspective. In combination with the full reply of at least one correct replica that actually executed the request, this is sufficient to enable a client to verify and consequently accept the correct result. For comparison, in BFT systems with $3f + 1$ execution replicas there is no need for checkpoints to contain information about replies, because at least $f + 1$ of the $2f + 1$ execution replicas reaching a checkpoint are correct, have themselves processed the requests in question, and thus can provide a client with enough matching replies. This shows that a reduction from $3f + 1$ to $2f + 1$ execution replicas comes with a cost that is often hidden: an increase in checkpoint complexity. In particular, checkpoints must not only comprise knowledge about the current state of the application, but also about the results of specific commands that have been executed in the past. For use cases with a large number of active clients this requirement can lead to significantly larger checkpoints than would be necessary with $3f + 1$ execution replicas.

Performance Implications. Going from $3f + 1$ to $2f + 1$ execution replicas typically offers the opportunity to minimize the resource consumption of a BFT system. Unfortunately, under certain circumstances the smaller resource footprint is accompanied with a decrease in performance due

to responsibilities being shared among fewer replicas, thereby on average increasing the load per replica. A textbook example for this effect is the common hash-reply optimization that aims at reducing network traffic by selecting a responder replica to send the full, possibly large result while all other execution replicas only return small result hashes (see Section 9.1 for details). Using different responder replicas for different requests, this optimization has the potential to achieve overall reply throughputs that exceed the capacity of a single replica's outgoing network link. However, the effectiveness of the approach depends on the number of responder-replica candidates and decreases with fewer execution replicas. If there are n replicas in the system, each one of them on average acts as responder replica for $\frac{1}{n}$ requests. As a consequence, the larger the number of replicas n, the more full replies a replica is able to send before its outgoing network link reaches saturation. For use cases with large replies, this may cause BFT systems with $2f + 1$ execution replicas to provide a lower maximum throughput than systems in which $3f + 1$ replicas process requests. Similar observations can be made with regard to state-transfer efficiency in the context of applications with large states. Here, it is usually beneficial if an execution replica is able to download different state parts from different sources, for example $\frac{1}{n-1}$th of the state from each of the other $n-1$ replicas [68].

6.2 Relaxed Execution Order

The ability of a BFT system to tolerate faults originates from the fact that, with regard to the execution of requests, all correct replicas are bound by the order determined in the agreement stage. In most systems, replicas fulfill this requirement by sequentially processing requests which on the one hand is straightforward but on the other hand can significantly limit performance. To address this problem, different techniques have been developed that share the idea of allowing a replica to relax the order in which it executes requests. Based on their potential effects on a replica's state these techniques can be divided into two categories: non-speculative techniques, which when used on a correct replica are guaranteed to produce a consistent state, and speculative techniques, which potentially result in inconsistencies and thus require a correct replica to be able to perform a rollback of its state. In the following each of the categories is presented in detail; Table 2 summarizes this discussion.

Non-speculative Techniques. When it comes to relaxing the request execution order, a key insight is that to keep a replica's state consistent it is sufficient to focus on the order of state-object accesses. Assuming that all such accesses are protected by locks, OptSCORE [59] exploits this idea by relying on a custom scheduler that uses the UDS algorithm [63] to enforce a deterministic order in which application threads are allowed to acquire these locks. That is, instead of sequentializing the execution of entire requests, OptSCORE only sequentializes the execution of critical sections, thereby enabling parallelism in the rest of the application. In order to be effective, this approach requires modifications to the application code, specifically the introduction of tailored lock/unlock primitives that allow the UDS scheduler to control the acquisition and release of locks.

If modifications to the application are undesirable or impossible, the request execution order can also be relaxed by reducing sequential processing to requests that conflict with each other, typically due to accessing at least one common state object. CBASE [73] builds on this principle by introducing

Table 2. Comparison of BFT systems relaxing the execution order of requests.

	OptSCORE [59]	**CBASE** [73]	**ODRC** [41]	**PBFT r/o** [26]	**Zyzzyva** [72]	**Eve** [70]
Parallel execution	✓	✓	Possible	Possible		✓
Request-conflict analysis		✓	✓			✓
Client-assisted retries			✓	✓	✓	✓
Rollbacks due to faults					✓	✓
Rollbacks due to conflicts						✓

a parallelizer component between the agreement and execution stage of each replica. The parallelizer is responsible for analyzing the agreed sequence of requests and deciding which requests can be safely processed in parallel as they are guaranteed to not conflict; Escobar et al. [49] recently showed how to efficiently implement such a parallelizer with reduced synchronization overhead. To conduct the conflict analysis, the parallelizer must be equipped with application-specific knowledge about the structure, characteristics, and possible dependencies of requests. If the parallelizer fails to identify a request, for example due to having been provided with incomplete specifications, the parallelizer has to conservatively assume that it conflicts with every other request and therefore resort to sequential execution. As a consequence, CBASE's approach is particularly effective if (1) it is possible to precisely determine the effects of a command based on its request, (2) the overhead of analyzing a request is negligible, and (3) the fraction of conflicting requests in the workload is small.

While CBASE exploits knowledge about non-conflicting commands to relax the execution order of requests in space, ODRC [41] has shown that such information can also be used to relax the execution order in time. Specifically, ODRC enables a replica to defer the processing of a subset of requests until (if ever) it is actually necessary for system progress. The subset of requests that a replica is allowed to skip is determined for each replica individually and selected in such a way that each request is still guaranteed to be processed by $f + 1$ replicas, which in the normal case is enough for the client to obtain a stable result. In case of faults, replicas retroactively execute buffered requests to supply the client with additional replies, thereby essentially completing the reordering of requests. Otherwise, buffered requests are eventually garbage-collected once their effects are part of stable checkpoints, which offers the benefit of replicas not needing to spend resources on executing them.

While ensuring a consistent application state for writes usually involves additional efforts, for reads this is straightforward as by definition they do not modify state. As a consequence, each replica may independently choose the point in time at which it performs a read without being at risk of becoming inconsistent. PBFT [26] relies on this principle for its read-only optimization that aims at minimizing read response times by bypassing the agreement protocol. More specifically, instead of first assigning a sequence number to an incoming read request, replicas using this optimization immediately execute a read after having received it. Although unproblematic with regard to the resulting state, this technique requires additional measures to be applicable in a safe and useful manner: (1) Due to replicas possibly executing the same read at different points in time relative to writes, the optimization cannot guarantee that correct replicas produce matching replies. In particular, it is possible that there are fewer matching replies than are needed by a client to verify a result. This means that when using this optimization clients must be prepared for a read to fail and in such case, for example, repeat the operation as a regular read that is sent through the agreement protocol. The likelihood of this to happen increases with the number of faulty replicas in the system as well as the frequency of conflicting writes to state objects accessed by the read. (2) As another byproduct of the inability of correct replicas to provide matching replies for optimized reads, it is often necessary, for both reads and writes, to increase the voting threshold beyond which a client considers a result stable (e.g., to $2f + 1$ [26]). Otherwise, a combination of replies from faulty replicas and trailing correct replicas could lead to a client accepting a stale result. (3) If used in BFT systems that relax the execution order of writes in a speculative way (see below), immediately executed reads might observe state that replicas will never agree on. Accepting such state at the client would mean to potentially provide users with incorrect information on application progress and therefore must be prevented. In PBFT, for example, replicas solve this problem by delaying their replies to optimized reads until having obtained confirmation from the agreement protocol that the observed state indeed has become stable. Replicas in Eve [70], in contrast, directly return their results and shift the responsibility for suppressing unconfirmed information to clients, which for this purpose employ an increased voting threshold that causes only non-speculative results to be delivered to users.

Speculative Techniques. The key requirement of having to keep the state of correct replicas consistent at all times, on the one hand, simplifies the reasoning about the correctness of non-speculative techniques but, on the other hand, also limits what they can do to improve performance. Speculative techniques address this issue by going one step further and allowing temporary inconsistencies between correct replicas, as long as it is ensured that these inconsistencies are eventually resolved and never become visible to users or other external processes. As a consequence, such techniques typically rely on some form of rollback mechanism that enables a replica to reset its state to a consistent version. Furthermore, speculative techniques introduce additional prerequisites for the application. In particular, the execution of a request must not involve irreversible actions such as remote calls to external processes or the transmission of control signals to physical actuators.

One of the first examples of a speculative execution technique proposed for BFT systems is PBFT's support for tentative execution [26] which allows replicas to process a request once it has been prepared, but before it is actually committed. Zyzzyva [72] extends this idea by already executing requests after the initial proposal phase of the agreement protocol. In both cases, only an increasing prefix of the sequence of requests provided by the agreement stage is stable, the rest is speculative. Hence, the speculative part may change in the course of a leader switch and consequently require a replica to rollback its state to a version that reflects a point in time in the stable prefix. In PBFT and Zyzzyva, replicas perform such a rollback by loading a stable checkpoint and thereby resetting the contents of all state objects to the values they had when the checkpoint was created. Erasing all inconsistencies introduced by speculatively executed (write) requests, this procedure allows a correct replica to reach a state from which it is able to make progress.

Using PBFT's tentative execution or Zyzzyva's normal-case protocol, the extent to which the execution order of requests can differ between correct replicas is limited to the speculative part of the agreed request sequence. Furthermore, in case the use of inconsistent execution orders causes correct replicas to actually diverge, both systems ensure that a rollback allows them to fall back to the globally agreed execution order. In contrast to these two techniques, Eve [70] in the normal case does not enforce a specific execution order on all correct replicas as long as they produce the same replies and end up in the same state. As a main benefit, this enables replicas to relax the execution order of requests by processing them in parallel. If the outputs and states determined by individual replicas match, the system continues to make progress. Otherwise, correct replicas in Eve roll back their states to the latest stable checkpoint and afterwards resort to sequentially processing the affected requests in the same order. Therefore, rollbacks in Eve are not always a consequence of faults (e.g., a faulty leader proposing different requests for the same sequence number to different followers) but may also be caused by the nondeterministic parallel execution of requests accessing the same state objects. To minimize the probability of the latter, Eve offers means to detect and separately execute requests based on a priori information about conflicts. That is, although not necessary for the correctness of the system, Eve also benefits from application-specific knowledge.

7 CHECKPOINTING

BFT systems tolerate arbitrary faults based on the principle that all correct replicas reach consistent decisions due to executing the same state machine with the same inputs. As the input sequence grows with every request submitted by clients, in practice the size of this sequence at some point usually exceeds the amount of memory available to each replica. Consequently, in order to be able to proceed over the entire lifetime of a system, replicas must comprise some form of garbage-collection mechanism for removing state-machine information about already processed entries. On the downside, the necessity to discard entries decisively limits the extent to which a replica can assist other replicas in making progress. Without additional measures, this would pose a problem for BFT systems deployed in asynchronous environments in which replicas are connected through

an unreliable network and therefore cannot exactly determine whether another replica is faulty or simply advances at a slower speed (see Section 2). Specifically, it could lead to scenarios in which correct replicas fall so far behind that the other replicas in the system have garbage-collected all information about the entries next required by the trailing replica, thereby causing the trailing replica to get stuck indefinitely. To avoid this problem, replicas in BFT systems periodically create checkpoints of all relevant state and only garbage-collect data that is covered by a stable checkpoint and therefore stored at a sufficient number of replicas in order to not get lost. This allows trailing replicas to catch up by fetching a checkpoint from another replica and update their states accordingly. Besides, similar checkpoint-based mechanisms can also be used to recover replicas from faults (more on this topic in Section 8) or integrate new replicas into an already running system. This section gives an overview of the differences between BFT systems with regard to the contents, representations, as well as the creation of checkpoints. A summary of this discussion can be found in Table 3.

7.1 Checkpoint Contents

To be of use for garbage-collecting a prefix of the sequence of state-machine inputs, a checkpoint must comprise all effects the inputs in the prefix had on a correct replica. That is, a checkpoint has to reflect the replica state at a specific point in time, more precisely: after the execution of the input at a particular sequence number s. In the ideal case, a checkpoint for a sequence number s contains all information necessary for a correct replica to reach a state that is indistinguishable from the one the replica would have if it had processed all inputs up to sequence number s. In general the state parts relevant for checkpointing can be divided into three categories:

- *Application State.* First and foremost, a checkpoint must provide information on the objects representing the state of the replica's application instance. By nature, this part of a checkpoint is use-case specific and its internal semantics may be opaque to the replication protocol. Without the application state in the checkpoint, replicas after loading a checkpoint would execute requests based on their old state and thus very likely produce incorrect results.
- *Protocol State.* This category consists of state that is application independent and required by a system's BFT replication protocol to offer certain guarantees. Most of this data therefore is protocol specific. The textbook example for such information is the set of request ids referencing the latest request of each client that the replica has executed. As discussed in Section 2, replicas rely on this set to perform duplicate suppression and prevent the same client request from being processed more than once. If this information is missing from a checkpoint, replicas that applied the checkpoint afterwards might mistakenly decide to execute requests that at this point are ignored by other correct replicas, thereby introducing inconsistencies.
- *Replies.* A third major part of a replica's state is the reply store containing the results to processed requests, one for each client (see Section 2). Maintaining this store enables a replica to retransmit a reply in case the previous attempt failed due to an unreliable network. In case checkpoints do not include the reply store, replicas are unable to provide responses for requests they skipped when bringing themselves up to date via a checkpoint.

Table 3. Comparison of checkpoint contents, representations, and creation techniques. The symbol (✓) indicates that a feature is required by the protocol but not mentioned in the corresponding publication.

	PBFT [27]	**Omada** [48]	**UpRight** [32]	**Dura-SMaRt** [17]	**DFC** [46]
Application state	✓	✓	✓	✓	✓
Protocol state	(✓)	✓	(✓)	(✓)	✓
Replies		✓			
Representation	State	State	Hybrid	Hybrid	State
Creation technique	Copy on write	Stop and copy	(Configurable)	Rotation	Fuzzy

Interestingly, the checkpoints in most BFT systems do not comprise all the information mentioned above and therefore are unable to exactly reproduce the state a correct replica would have if it had processed all previous inputs. Instead, for reasons of efficiency, checkpoint-protocol designs often aim at creating a state that is "good enough" to not endanger the guarantees provided by the overall system. PBFT's checkpoint protocol [27], which represents the basis for checkpointing in many other BFT systems, for example, involves checkpoints that lack the snapshot of a replica's reply store. For PBFT, this does not pose a problem, because the protocol ensures that a checkpoint only becomes stable if $2f + 1$ of the $3f + 1$ replicas reach it, of which at least $f + 1$ are correct and thus able to supply the client with enough valid replies. However, while correct in the presence of $3f + 1$ replicas, as discussed in Section 6.1, this approach may have unintended consequences when used in BFT systems with fewer replicas. Including result hashes in the checkpoint [41], instead of the full results, also does not allow replicas to entirely restore their state. Relying on this approach, it is therefore essential to have at least one correct replica remaining in the system at all times that has obtained the full results by actually executing the requests in question. Finally, the vast majority of BFT protocol descriptions do not mention the latest-request-id set being part of the checkpoint. Presumably, this is an imprecision in the protocol specifications, not an explicit design decision.

7.2 Checkpoint Representations

Independent of the particular contents of a checkpoint, there is usually some flexibility with regard to how this information is represented. Typically, BFT systems use one of two main approaches for this purpose. While *state checkpoints* for a specific sequence number include a snapshot of all relevant state, *hybrid checkpoints* combine an earlier snapshot with a list of subsequent state-machine inputs.

State Checkpoints. In most BFT systems, checkpoints directly comprise the values application-state objects and other important data structures had at the sequence number for which the checkpoint was created [27, 28, 46, 48]. Maintaining a checkpoint this way is comparably straightforward and offers the key benefit that to load a checkpoint a replica can simply copy the checkpoint's contents to the actual locations where they are needed. To further improve the efficiency of this process, checkpoints often carry additional information on their internal structure as well as metadata such as content hashes and last-modified sequence numbers. Inspired by Merkle trees [88], this approach enable a replica to quickly distinguish between (1) outdated state parts the replica must replace with the corresponding values in the checkpoint and (2) the parts of the replica's state that already reflect the latest version and therefore can be left unmodified [27].

Hybrid Checkpoints. For some use cases, capturing the entire replica state at the relatively high rate required by BFT systems is prohibitively expensive. Hybrid checkpoints [32] address this problem by allowing a replica to create frequent checkpoints based on infrequent replica snapshots. To do so, a replica stores the latest snapshot and afterwards adds a sequence of deltas to it; a delta is a list of all requests the replica has executed between two checkpoints. That is, a hybrid checkpoint for a sequence number s contains a replica-state snapshot for sequence number $s_{snapshot} < s$ as well as the deltas covering all sequence numbers between $s_{snapshot} + 1$ and s. To apply such a checkpoint, a replica first loads the snapshot and then executes the requests from the deltas in the order of their sequence numbers, thereby eventually reaching the state reflected by the checkpoint. Compared with state checkpoints, the major advantage of hybrid checkpoints is the fact that their creation typically involves less overhead, in particular because a replica already has the relevant information to create deltas (i.e., the output sequence of the agreement stage) at its disposal. However, the approach also has drawbacks that need to be considered [46]: (1) Loading a hybrid checkpoint usually takes more time than loading a state checkpoint, as a replica cannot just copy snapshot contents but in addition has to process the requests included in the checkpoint. (2) Reproducing a

certain state by executing requests in general consumes more processing resources than directly copying the associated state modifications [42]. (3) Most importantly, in the context of Byzantine faults updating replica state via request execution introduces a potential vulnerability into the system that originates from a crucial difference between state checkpoints and hybrid checkpoints. While state checkpoints contain the effects all previous requests had on the replicated state machine, the deltas in hybrid checkpoints represent state-machine inputs. This distinction becomes important in scenarios such as the following. If a malicious client manages to send a manipulated request that compromises a replica (e.g., through a buffer overflow or a SQL injection [62]) but does not manifest in the state parts covered by a checkpoint, this has no effect on another replica that applies a verified state checkpoint. In contrast, successfully verifying deltas solely guarantees that a replica actually executes the requests committed by the agreement stage; specifically, it does not validate the effects those requests have on the replica. As a result, executing the manipulated request in the course of applying a checkpoint entails the risk of propagating the fault to newly added replicas or reintroducing already handled faults in case the checkpoint is used as part of a recovery mechanism. A possible solution to this problem is to resort to deltas that, instead of requests, are based on state updates reflecting the actual changes requests made to the relevant parts of the replica state [43].

7.3 Checkpoint Creation

For applications with a large state, the need to periodically generate checkpoints can result in significant overhead, both in terms of performance as well as resource consumption. To address this issue, several checkpoint-creation techniques have been developed that typically trade off an increase in complexity for an improvement in efficiency. The four most widely used of these techniques are discussed below. They share the requirement that in order for states to be verifiable by comparing data from different replicas, all correct replicas must have a consistent view on the sequence numbers at which checkpoints are due. Note that this prerequisite is unique to BFT state-machine replication. In crash-tolerant systems, replicas trust each other and consequently there is no need for checkpoints to be comparable, allowing each replica to individually decide when to create a checkpoint.

Stop and Copy. The straightforward approach to produce a consistent checkpoint for a sequence number s is for all replicas to suspend the execution of requests after having processed the request with sequence number s, then create the checkpoint, and afterwards resume request execution [32]. Temporarily stopping execution during checkpoint creation offers the benefit of leaving a correct replica in a valid state, thereby greatly simplifying the task of capturing a consistent checkpoint by copying all relevant data. Another advantage of this technique is the fact that it does not make any assumptions on how or where an application maintains its state. For legacy applications, this often means that already existing checkpoint mechanisms can be reused for BFT replication. On the downside, for large states the stop-and-copy approach may lead to major service disruptions, which are a direct result of the need to copy the entire state once every checkpoint interval. As a consequence, efficient stop-and-copy use cases generally either involve large checkpoint intervals or small applications for which request execution only has to be suspended for a negligible amount of time.

Copy on Write. To minimize the downtime associated with checkpoint creation, BFT systems may rely on copy-on-write mechanisms to limit the performance overhead of state capture to the parts that actually changed since the last checkpoint [27, 28]. This way, the overall duration for which a replica needs to suspend request execution is not tied to the size of the replica's state, but instead depends on the amount of data modified between two checkpoints. In the worst case, this still forces a replica to copy the entire state, however for short checkpoint intervals in practice it is much more likely that the changes only affect a small portion of it. To determine the parts to capture, copy-on-write requires a replica to be able to identify and store the

modifications performed by requests. A common way to make such information available to the replica is to (manually) insert callbacks into the application code that trigger when the value of a state object is about to be updated [28]. This allows the replica to copy the old version of the object into the checkpoint. Alternatively, a replica may rely on copy-on-write mechanisms that automatically create a new state-object copy for use in the application if an object is modified while the checkpoint capture is in progress [29, 32]. Either way, both variants enable the interleaving of checkpoint creation with request execution and therefore, in contrast to the stop-and-copy approach, distribute the performance overhead over a longer period of time.

Checkpoint Rotation. When all replicas in a BFT system capture their states at the same sequence number, the associated performance penalty can lead to an overall service disruption. A possible solution to mitigate this effect is to rotate the responsibility for creating an application-state snapshot among replicas [17]. That is, instead of all replicas producing a snapshot once every checkpoint interval CPI, each replica $r \in \{0, \ldots, n-1\}$ only performs this task for sequence numbers $(n \cdot i + r) \cdot CPI$, with $i \in \mathbb{N}$ and n being the number of replicas in the system. Although limiting state capture to every nth checkpoint interval, a replica using this technique nevertheless is able to create checkpoints for all sequence numbers $i \cdot CPI$ in the form of hybrid checkpoints (see Section 7.2). Consequently, a checkpoint in this case consists of the latest replica-state snapshot as well as all requests executed since the capture of this snapshot. Relying on hybrid checkpoints, the approach comes with the drawbacks discussed in Section 7.2 that generally arise when the loading of a checkpoint involves the active execution of requests. Furthermore, checkpoint rotation introduces an additional vulnerability that might be exploited by malicious replicas to compromise another replica through a manipulated state transfer [46]. The problem is rooted in the fact that due to replicas capturing their states at different sequence numbers, the representations of their hybrid checkpoints for the same sequence number are not identical and therefore cannot be directly verified by comparison. For example, while a checkpoint cp_A for a sequence number s may be based on a snapshot for sequence number $s - CP$ and thus also include the requests from $s - CP + 1$ to s, another checkpoint cp_B can directly contain the snapshot for sequence number s. As a consequence, based on a byte-to-byte comparison of the two checkpoints it is impossible for a replica to determine whether cp_A and cp_B refer to the same resulting replica state or not. For this reason, to verify cp_A with the help of checkpoint cp_B the checkpoint-rotation technique requires a replica to first load cp_A's snapshot and then execute cp_A's requests. Only at this point, the replica then is able to retroactively verify the correctness of checkpoint cp_A by comparing its current state to the snapshot of checkpoint cp_B. Unfortunately, the need to load unverified snapshots opens the possibility for malicious replicas to launch successful attacks by supplying other replicas with hybrid checkpoints whose snapshots have been tampered with. To prevent such scenarios, it is crucial to ensure that correct replicas cannot become compromised before having the chance to verify the state. One way to achieve this is to perform the critical steps (i.e., loading/execution of unverified snapshots and requests) in a sandbox and only continue with the resulting checkpoint after having verified it.

Deterministic Fuzzy Checkpoints. Although checkpoints have to be deterministic in order to be verifiable by comparison, the same does not necessarily have to apply to the checkpoint-creation process itself. Using fuzzy checkpointing [46], correct replicas for example may individually select when to begin the capturing of data for the next snapshot and still end up producing identical checkpoints. Among other things, this offers the possibility for slower replicas to start early in an effort to complete the checkpoint at a similar point in time as faster replicas. Relying on this checkpointing technique, a replica already starts to copy state objects for an upcoming snapshot before having actually reached the corresponding sequence number. While the state capture is in progress, the replica in parallel continues to execute requests and tracks the resulting state

modifications that occur after the beginning of the state-capture procedure. When arriving at the target sequence number, this preparation enables the replica to finalize the checkpoint by applying the recorded state modifications to the affected state-object copies, thereby generating a checkpoint that is consistent and comparable across all correct replicas in the system. With most of the work being done in advance, this approach typically enables replicas to complete a checkpoint shortly after having reached the sequence number for which the checkpoint is due.

8 REPLICA RECOVERY

BFT systems provide resilience against arbitrary faults as long as the total number of faulty replicas does not exceed the threshold a system has been designed for. Beyond this threshold, these systems in general do not make any guarantees with regard to the availability, correctness, or consistency of results and replica states. For long-running BFT systems this poses a problem because without additional measures faults accumulate over time, thereby increasing the probability that the number of faults eventually passes the predefined tolerance threshold. The common approach to prevent such scenarios is to enable systems to recover replicas from faults by applying software rejuvenation techniques [37, 66, 94]. This solution makes it possible to specify the fault-tolerance threshold based on the frequency of recoveries instead of a BFT system's entire lifetime. This section presents different strategies of integrating rejuvenation techniques with BFT systems and discusses the impact recovery mechanisms can have on availability (see Table 4 for a summary). Primarily, the section focuses on approaches that allow a replica to recover based on information available within its own replica group. For some fault scenarios such as permanent hardware failures, additional mechanisms are needed to replace faulty replicas, typically requiring a reconfiguration of the replica group.

8.1 Recovery Strategies

Independent of the particular mechanism used to recover a replica, there are two general strategies for deciding when to perform the rejuvenation. A replica may be recovered (1) *reactively* once a fault has been detected or (2) *proactively* at predefined points in time. Although most BFT systems opt for either one or the other, it is also possible to combine both approaches [113].

Reactive Recovery. The underlying idea of reactive recovery is to only initiate recovery procedures when they are actually considered necessary due to a replica having shown a behavior that violates its specification. As a consequence, to be able to apply this approach it is essential for a BFT system to comprise mechanisms for detecting faulty replica behavior [60]. Typically, such mechanisms analyze the outputs of a replica and compare them to the corresponding outputs of other replicas. If after the execution of a committed client request, for example, one replica responds with a different result than the other replicas, in most BFT systems this is a clear indication that the replica is indeed faulty and therefore must be recovered. Furthermore, a client in this case may use the set of diverging replies as evidence to convince other replicas that did not directly observe the faulty behavior themselves.

Unfortunately, assembling this kind of definitive proof is solely possible for a very limited spectrum of faulty behavior. The main reason for this is the fact that in a distributed system it is often impossible to unequivocally determine in the first place whether a replica is actually faulty or not. In

Table 4. Comparison of BFT systems recovering replicas from faults.

	PBFT [27]	**PRRW** [113]	**VM-FIT** [98]
Strategy	Proactive	Proactive + reactive	Proactive
Granularity	Replicas: one by one	Replicas: group by group	All replicas at once
Recovered parts	Agreement + execution	Agreement + execution	Execution
Trusted component	Secure coprocessor	Synchronous wormhole	Virtualization layer
Availability improvements	Immediate resume after reboot	Additional replicas	Shadow replicas

asynchronous environments in which there are no guaranteed upper bounds on transmission times, for example, clients cannot distinguish between faulty replicas that omitted to send a reply and correct replicas whose replies have been delayed by the network. Reactive-recovery mechanisms usually address this issue by relying on two levels of certainty [61, 113]: scenarios in which faulty behavior has been provably detected (e.g., due to a replica providing an incorrect result) and cases in which it only has been suspected (e.g., due to a replica's reply not having arrived within a certain amount of time). While for detected faults it is generally most effective to immediately start the recovery of the affected replicas in order to quickly handle the problem, for suspected faults recoveries should be scheduled in a coordinated fashion [113]. Otherwise, if a correct replica is wrongfully suspected to be faulty, its recovery may cause the system to become temporarily unavailable because of too few correct replicas remaining and actively participating in the replication protocol.

Proactive Recovery. While reactive recovery by design is limited to replica faults that are observable by other system parts, proactive recovery can also be used to effectively remove faults that do not manifest in a replica's output. This is due to the fact that proactive-recovery mechanisms trigger replica recoveries in periodic time intervals, not based on evidence of faulty behavior [27]. Compared with reactive recovery, this approach has the benefit of not requiring means for fault detection and signaling, but it typically comes at the cost of unnecessarily recovering correct replicas on a regular basis. As the frequency with which replicas are rejuvenated is usually a configurable value, the overhead of proactive recovery can be reduced by increasing the duration of the time interval between two recoveries. However, this also means that on average it takes a BFT system longer to eliminate potential non-observable faults. An additional possibility to influence the tradeoff between efficiency and resilience is the selection of the number of replicas to recover in parallel. Here, the options range from rejuvenating replicas one by one [27, 98], to refreshing replicas in groups [113], to recovering all replicas in the system simultaneously [43].

8.2 Recovery-specific Requirements

In order to be both effective and safe, a recovery mechanism should be tightly integrated within the replication logic [74]. This approach in general puts additional requirements on a BFT system that go beyond the common assumptions discussed in Section 2. Without further restrictions it would be impossible to build potent recovery mechanisms because an adversary could simply prevent a compromised replica from ever initiating the rejuvenation process. The common approach to solve this problem is to assume that each replica is equipped with a trusted component that remains fully operational even if the rest of the replica is under the control of an attacker. The specifics and size of this trusted component vary between systems, ranging from a secure cryptographic coprocessor [27] to an entire virtualization layer [42, 43, 98]. To support reactive recovery, trusted components on all replicas must be able to reliably communicate with each other [113]. In particular, this is necessary to exchange proofs of faulty behavior (see Section 8.1) with the goal of convincing another trusted component that its local replica has to be recovered. For proactive recovery, on the other hand, there is no essential need for the trusted components of different replicas to interact directly. Instead, to provide guarantees on the timeliness of replica recoveries, in this case it is crucial that there are limits on the duration of recovery procedures as well as upper bounds on the drift rates of local clocks [115, 116]. This insight led to the concept of wormholes [120], which are synchronous trusted subsystems that are, for example, responsible for triggering and executing the proactive recovery of a replica's asynchronous state-machine-replication subsystem [115].

In addition to the trusted component, a second important assumption required for recovery is the existence of read-only memory whose contents persist across machine reboots and cannot be manipulated by an attacker. This memory is used to store all information that is necessary for

bootstrapping a correct replica after restart. In this context two categories of data are of particular relevance: (1) To establish and refresh session keys for the communication with other replicas, the read-only memory should contain their public keys [27]. Of course, the local replica's own private key also has to reside in read-only storage, however this must be in a section from which it is only accessible by the trusted component. (2) The read-only memory should comprise the code required to run a replica [96, 100], including the operating system, the replication logic, the application software, as well as the recovery mechanism. As an alternative, it is possible to solely maintain hashes of these code parts in read-only storage and the actual code elsewhere [27]. In this case, prior to execution the trusted component first needs to verify the correctness of the code based on the hashes.

8.3 Recovery Steps

The goal of recovery is to transform a replica that may be subject to (detected or undetected) Byzantine faults into a replica that is free of faults. In general, this task requires the following steps:

- *Reboot.* Having decided that a rejuvenation of its local replica is due, the trusted component initiates a restart of the replica's machine to erase existing faulty processes and other potential effects of a successful intrusion. If demanded by the recovery protocol, prior to the reboot the trusted component saves all relevant state to persistent storage [27]. After the restart, leveraging information contained in its local read-only memory (see Section 8.2), the trusted component ensures that the replica's operating system, replication middleware, and application are loaded from a clean code base. Consequently, at this point the replica's logic can be considered correct, however its state still might be corrupted, missing, or out of date.
- *Rejoining the Replica Group.* Once the reboot is complete, a replica needs to reestablish the communication with clients and other replicas. As an attacker might have previously gained access to session keys, as part of this step the trusted component usually generates and distributes new keys [27]. In addition, the trusted component on each replica typically increments a local epoch counter whose current value can be included in messages to be able to distinguish old from new messages and prevent replay attacks [102].
- *State Update.* Being able to communicate, a recovering replica now can assess whether it has fallen behind with respect to other replicas, and if so how far. Based on the outcome of this assessment the replica then decides on its next actions. If the replica's recovery-induced period of unavailability was short, it for example may be sufficient for the replica to verify its current state by obtaining a stable hash of the correct state from other replicas. In contrast, in case the replica was faulty prior to recovery or has been unavailable for a long time, a full state transfer could be necessary to bring it up to date. Either way, the purpose of this phase is to enable the replica to once again actively participate in the replication protocol as a correct member.

Depending on the individual design of the recovery mechanism in use, the order and specifics of the steps presented above may vary between BFT systems. In PBFT [27], for example, the reboot phase includes a restart of the entire machine, whereas other systems keep the physical server running and instead create a new virtual machine for the replica from scratch [42, 98]. Besides, the creation and exchange of new session keys for the communication with other replicas, for instance, may be interleaved with a replica's efforts to rejoin the group and update its state [27].

8.4 Availability Implications

While recovering, a replica in general is unable to fulfill its usual responsibilities as it cannot continuously participate in the replication protocol throughout the entire rejuvenation process. Some BFT systems therefore aim at keeping the period of replica unavailability as short as possible. In PBFT [27], for example, a replica only suspends its active involvement in the agreement protocol

while the machine reboot and the subsequent code verification are in progress. That is, once the replica has been restarted it immediately resumes the processing of agreement messages based on the protocol state that existed before the start of the recovery. As a consequence, if the replica was faulty prior to the recovery and left behind an incorrect state, after reboot the replica temporarily still might show faulty behavior. However, the extent to which this can occur in PBFT is bounded by the size of the window that defines for which sequence numbers the replica currently participates in the consensus process. Specifically, this means that it is infeasible for faulty replicas to leave behind false information for an arbitrary number of consensus instances and expect the recovering replica to later resume based on this state. If they nevertheless do, a window-bounds plausibility check run by the recovering replica (relying on progress information provided by other replicas) will later identify and discard consensus instances that cannot yet have been active when the recovery started. Whenever a recovering PBFT replica detects that a state object is faulty or outdated, it initiates a state transfer that replaces the affected object with the latest version supplied by another replica.

In contrast to PBFT, VM-FIT [43, 98] does not minimize the period of unavailability by resuming a replica with its old state and (if necessary) repairing incorrect state parts afterwards. Relying on virtualization technology, the recovery process in VM-FIT instead involves the start of a second replica instance ("shadow replica") on the same physical machine that hosts the replica to be rejuvenated. As part of the recovery process, the shadow replica is created from a clean code base and initialized with an up-to-date state, which is transferred from the old replica and verified or corrected based on checksums and data provided by other replicas. Until the shadow replica is set up, the old replica continues to operate and thereby remains available. This way, replica unavailability is limited to the short duration of the switch between old replica and shadow replica.

As an alternative to the solutions discussed above, the availability of a BFT system during recovery procedures can also be improved by increasing the total number of replicas participating in the replication protocol. A naïve solution, for example, would be to model recovering replicas as arbitrarily faulty and consequently increase the value for the system parameter f that specifies the maximum number of Byzantine faults to tolerate. However, the more efficient approach is to treat recovering replicas as nodes that have (temporarily) crashed and therefore on the one hand do not respond to requests, but on the other hand also themselves do not distribute faulty messages. As a result, a total of $3f + 2k + 1$ replicas are sufficient in order to be able to recover up to k replicas at the same time [113]. When increasing the number of replicas this way, it is guaranteed that there are always enough correct replicas available to tolerate f faults even if k replicas are currently recovering. This property frees recovering replicas from the need to participate in the agreement process while rejuvenation is still in progress. On the downside, to account for the increased number of total replicas it becomes necessary to use larger agreement quorums [87, 96] compared with the normal $3f+1$ setting. This means that the improved availability during recovery is likely to come at the cost of an overall decrease in system performance.

8.5 Replica Evolution

Simply recovering replicas to ensure that they have the correct state often is not sufficient to offer resilience for long-running services. If an attacker, for example, is able to compromise a replica by exploiting a vulnerability in the application code, the sole recovery of a replica temporarily can remove the effects of the successful intrusion, however it does not prevent the attacker from reusing the same vulnerability afterwards to take over the replica once again. Even worse, with the attacker gaining experience subsequent attacks commonly require less time than the initial attempts. Building on these insights, an effective strategy for an attacker therefore is to accumulate information on how to attack different replicas and eventually apply this knowledge to compromise multiple replicas within a short period of time, thereby exceeding the fault-tolerance threshold between two recoveries.

To solve this problem, a BFT system should ensure that any replica-specific information potentially obtained by an attacker becomes worthless after rejuvenation. For this purpose, a system needs to combine replica recovery with a second concept: replica evolution [19, 91]. That is, instead of recreating the same replica as before, the goal of a recovery mechanism should be to produce a replica that is equivalent with regard to functionality, but sufficiently differs from its previous version. Several techniques have been proposed to achieve this goal. The secure-secret distribution systems COCA [127] and CODEX [84], for example, rely on a proactive secret sharing protocol [128] that on each recovery invalidates any confidential information an attacker could have previously obtained by refreshing keys. The FOREVER service [114] performes upgrades in the course of a recovery, which include the application of security patches, the installation of new software versions, and the randomization of program data and code [21]. As a consequence, existing vulnerabilities can be dynamically removed from the system over time. In addition, FOREVER targets the modification of port numbers, usernames, and passwords in an effort to make repeated attacks more difficult. Roeder and Schneider [102] presented mechanisms that allow a BFT system to comprise obfuscated replica executables and to automatically substitute these executables with newly generated versions during recovery. Examples of obfuscation techniques that may be used for this purpose include heap randomization [15], system call reordering [30], and instruction set randomization [71]. Finally, Platania et al. [96] applied the MultiCompiler [65] to periodically recompile the bitcode of replicas in Prime [2] in order to diversify their implementations. In summary, by ensuring that replicas evolve as part of the recovery process, all these works aim at minimizing the time span during which an attacker may benefit from the accumulation of knowledge on how to take over multiple replicas.

8.6 Replica-Group Reconfiguration

Not all replica failures that may occur in a BFT system can be addressed by the software-based approaches discussed in previous sections. For example, if a hardware component breaks down, often times the only effective solution is to replace the entire server with another machine. Unfortunately, substituting an existing replica for a new one is not straightforward in BFT systems as the reconfiguration of the replica group impacts various system parts. Among other things, the agreement protocol needs to switch to the updated replica-group configuration in a safe manner [75], clients must learn about the new replica [18], retiring replicas have to lose the capability to further participate in the system [16, 85], and for some use cases even the application requires adjustment to membership changes in the replica group [101]. To achieve a consistent reconfiguration, a common technique is to model a reconfiguration operation as a special command that is ordered in the same way as regular client requests, thereby determining a well-defined point in time at which the reconfiguration should take place across replicas [75]. In BFT-SMaRt [18], for example, administrators may issue such reconfiguration commands through a trusted client whose instructions are verifiable by all replicas. Once a reconfiguration command has been agreed on and is ready for execution, instead of forwarding it to the application, a replica locally installs the new configuration.

Apart from replacing faulty replicas, mechanisms for replica-group reconfiguration can also play an important role in minimizing the vulnerability of a BFT system, as shown by Lazarus [54], a control pane supporting the automatic management of diverse replicas. In contrast to the approaches discussed in Section 8.5, Lazarus does not implement replica evolution as part of the recovery mechanism, but instead based on replica-group reconfiguration. Specifically, the system continuously gathers information on replica-implementation vulnerabilities from public databases such as NVD [93] and uses this data to estimate the exploitability of the current replica group. If the risk of keeping the active configuration is considered too high, Lazarus substitutes vulnerable replicas with spare replicas from a pool of different implementations. Removed vulnerable replicas are put under quarantine and only used again after the necessary patches have been applied.

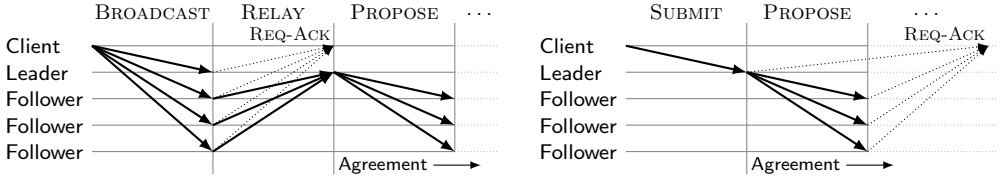
8.7 Durability

Tolerating the simultaneous failures of a subset of replicas is not always sufficient. For example, if all replicas reside in the same building, a power outage may render the entire group unavailable. Such scenarios are not included by the standard fault model which is why to address them the fault model needs to be extended to also cover the simultaneous crash and recovery of more than f replicas. In practice, this requirement can be fulfilled with the introduction of durability techniques ensuring that after a recovery the system state reflects all requests that previously have been completed [17]. Specifically, this means that replicas may only provide a client with a reply once the effects of the corresponding request have been logged to stable storage where the information survives a crash.

9 SEPARATION OF NORMAL CASE AND WORST CASE

While essential for correctness in the presence of faults, the redundancy provided by a BFT system in general comes with significant additional costs in the absence of faults. To minimize such computational and network overhead, several techniques have been proposed that, at their core, all follow the same fundamental principle of computer system design: the separate handling of the normal case and the worst case [76]. For BFT systems, this typically translates to approaches consisting of an optimistic first phase that, if necessary, is followed by a pessimistic second phase. In the optimistic phase, to save resources only a subset of replicas (sometimes a single one) is entrusted with a task. Under benign conditions, when these replicas are correct and the network delay is acceptable, the optimistic phase is sufficient to make progress. To address scenarios where this is not the case (e.g., as the result of a replica failure), the subsequent pessimistic phase triggered by a timeout is then responsible for including the remaining replicas and thereby preventing the system from getting stuck.

This section presents several applications of this general idea. The commonality of all these optimistic techniques is that they offer improvements in efficiency at the cost of an increase in protocol complexity, which is a direct consequence of the need to implement two phases with diverging objectives. Before applying these techniques, it is crucial to take into account their potential impact on system availability, stability, and performance, especially in the presence of faults. Having been designed for benign conditions, most optimistic mechanisms for example cannot guarantee progress if even a single replica fails. Eventually, the pessimistic phase solves this problem, however, depending on the timeout used, a substantial amount of time may pass until the switch actually occurs. Besides, knowing the timeout length, during the optimistic phase a malicious replica can slow down system progress by delaying its actions until shortly before the timeout would expire; studies have revealed that for some BFT protocols this behavior results in the entire system essentially becoming unusable [2, 33]. Considerations such as this show that optimistic techniques should be used only if their consequences are understood and acceptable for the particular application scenario. Furthermore, they underline the importance of increasing the robustness of BFT systems to failures [2, 7, 32, 33].



(a) Pessimistic approach (b) Optimistic approach

Fig. 6. Strategies for conveying a request from a client to the leader. The figures use Req-Ack messages to illustrate the respective earliest point in time at which the client may learn about a sufficient dissemination of the request and therefore cease its transmission. In general BFT systems do not send explicit Request-Acks.

9.1 Dedicated Contact and Responder Replicas

One of the first use cases of optimistic techniques in BFT systems was the interaction between clients and replicas. In this context two specific problems are of interest: the transmission of requests from a client to the server side and the delivery of replies in the opposite direction.

Request Transmission. In order for a new client request to be considered in a leader-based agreement protocol, the leader first has to learn about it, which can be accomplished in one of two ways: (1) As illustrated in Figure 6a, the pessimistic approach to achieve this is for a client to submit its request to all replicas in the system and then have each replica relay the request to the leader. With replicas directly receiving the request from the client, this strategy offers the possibility to implement all necessary leader-monitoring logic at the server side. Specifically, upon learning a new valid client request, a replica may immediately respond with an acknowledgment notifying the client that from now on the replica takes care of ensuring that the request eventually will be agreed on and executed, independent of whether the current leader is faulty or not. As a consequence, a client can focus on (re)transmitting its request until having obtained $f + 1$ acknowledgments from different replicas, that is, until learning that the request has reached at least one correct replica. (2) The optimistic counterpart for providing the leader with a new client request is shown in Figure 6b. Here, the client solely submits its request to the current leader, its dedicated contact replica, whose identity the client typically has learned through a reply to an earlier request [26, 40, 67, 72]. Due to the other replicas not being involved, the client in this case may be the only one besides the leader to know that the request exists. Therefore, the client itself is responsible for ensuring progress in the presence of a faulty leader. This problem is usually addressed by setting up a timeout that expires if the client fails to obtain a stable result to its request within a certain amount of time. As a reaction to the timeout, the client switches to the pessimistic approach and distributes its request to further replicas. In comparison to the pessimistic variant, the optimistic case requires fewer messages to be sent from clients to the server side. However, this comes at the cost of increased network traffic between replicas, because now the leader is the node in charge of disseminating the client request to other replicas in order to enable them to eventually execute it. This puts additional pressure on the leader's outgoing link and represents a bottleneck that has the potential to limit maximum system performance, especially in scenarios with large client requests [40].

Reply Transmission. When result verification is performed at the client, as it is in most BFT systems (see Section 4), clients require replies from different replicas to determine a stable result. The pessimistic approach to meet this requirement is for each replica that has executed the request to include the full result in its reply to the client. For large results, this can lead to significant network overhead which is why BFT systems typically offer an optimistic mechanism that involves a dedicated responder replica sending a full result while the other replicas minimize the size of their replies by each only providing a hash of the result. Depending on the system, the dedicated responder replica is either chosen by the client [26] or determined by the server side, for example, by rotating the role among replicas [40]. Applying this optimization, clients must be prepared to deal with situations in which they have learned and successfully verified the hash of the result based on a set of hash replies, but still do not know the actual result as the full reply is missing. Interestingly, root causes for such scenarios are not limited to faulty replicas (i.e., the correct full result is not provided in the first place) and network problems (i.e., the full reply has been dropped or delayed). Another reason for this optimization failing, for example, can be a correct replica that had fallen behind and subsequently applied a checkpoint to catch up, thereby skipping the execution of one or more client requests for which the replica therefore is unable to supply full results. Either way, clients commonly address these issues by requesting the full result from a different replica after a timeout.

9.2 Resource-efficient Replication

Limiting active replication-protocol participation during normal-case operation to only a subset of replicas is an effective approach to reduce a BFT system's resource footprint. Different variants of this basic concept have been applied to both the agreement stage as well as the execution stage.

Mode-specific Agreement. Presenting a generic mechanism for substituting the agreement protocol of a BFT system at runtime, Abstract [6] can be used as a foundation to design systems with different protocols for normal mode and fault handling. CheapBFT [67] and ReBFT [40] are examples for such mode-specific protocols targeting resource efficiency. In both protocols only a subset of replicas actively contribute to the agreement process by sending consensus messages. The size of the active subset is selected in such a way that the overall agreement stage is able to make progress if all participating agreement replicas behave according to specification and network conditions are benign. As a consequence, a single faulty replica can cause a system to switch to fault-handling mode and activate additional agreement replicas. Apart from improving resource efficiency, the general idea of an optimistic normal mode has also been implemented in other agreement protocols [38].

Optimistic Execution. With request processing in many cases being one of the most resource-intensive steps of state-machine replication, some systems aim at minimizing the number of execution replicas that are active during normal-case operation. When configured to tolerate up to f faults, ZZ [125] for example only operates with $f + 1$ execution replicas, thereby enabling clients to obtain stable results as long as all replicas provide correct replies and all of these replies actually reach the client side. If a monitoring mechanism hosted by the agreement stage suspects or detects that this is no longer the case, ZZ starts f additional execution replicas to tolerate the fault. Compared with other BFT systems, the startup of a replica in ZZ is a relatively efficient procedure as (1) execution replicas run in virtual machines and therefore have significantly shorter boot times than physical machines and (2) replicas start with a limited state and later fetch additional state objects on demand. Like ZZ, Spare [42] also relies on virtualization to maintain $f + 1$ active execution replicas in the normal case, but for an efficient fault handling draws on a set of prepared passive replicas whose states are periodically synchronized via state updates.

10 HYBRID FAULT MODELS

Assuming a wide spectrum of faulty behavior, the Byzantine fault model makes it inherently difficult to develop and operate systems tolerating such faults. In recent years, several works have aimed at facilitating these tasks by moving to hybrid fault models [117] with weaker guarantees. This section discusses the underlying idea of this approach and gives an overview of proposed solutions.

Distinction Between Fault Types. BFT systems designed for the assumptions presented in Section 2 provide correctness despite a powerful adversary or harsh network conditions, and even a combination of both. This comes at the cost of complexity and resource usage, which some practitioners have argued are unnecessarily high for relatively secure and predictable environments such as data centers [74, 97]. To address this issue, several works in recent years have abandoned the goal of offering full-fledged Byzantine fault tolerance and instead aimed at leveraging hybrid fault models. UpRight [32], for example, distinguishes between crashes ("omission failures") and incorrect behavior ("commission failures") and enables the configuration of separate upper bounds for each of the two categories. As a consequence, compared with the traditional Byzantine fault model, UpRight requires fewer replicas in scenarios in which not all of the replica failures are expected to involve incorrect behavior. VFT [97] extends this concept and further minimizes the replica-group size by exploiting the assumption that there is always a minimum number of well-connected replicas that are able to communicate with each other in a timely manner. If this assumption does not hold in practice,

depending on the specific protocol implementation, the system may either become unsafe or unavailable. Like VFT, XFT [81] also relies on a hybrid fault model but only requires the configuration of a single fault threshold that combines crashed, incorrectly behaving, and partitioned replicas.

Trusted Subsystems. A different direction of applying hybrid fault models is to design BFT systems as a composition of parts of two different types: untrusted and trusted subsystems. The untrusted parts may be subject to Byzantine faults and therefore can fail in arbitrary, possibly malicious ways. In contrast, the trusted parts are assumed to either work according to specification or to fail by crashing. This assumption must hold at all times, even for example if an attacker has successfully managed to compromise the untrusted parts of a replica. As a consequence of the hybrid fault model, whenever a trusted subsystem makes decisions or sends messages, other (untrusted and trusted) subsystems can be sure that these are correct by definition. Among other things, this property can be used to simplify system design [78, 103, 120] or increase the efficiency of protocols [31, 123].

In order to justify the trust put in them, trusted subsystems should have a small trusted computing base [104] as this makes it easier to reason about their correctness and identify bugs during development. Furthermore, the interfaces between trusted and untrusted subsystems should be designed in a way that attackers cannot exploit these interfaces to manipulate trusted components. To better isolate trusted components from the rest of the system, it is possible to implement them in the form of dedicated hardware modules such as a smart card [77], a trusted platform module [122, 123], or an FPGA [40, 67]. Other approaches [12, 78, 103] for this purpose rely on trusted execution environments provided by modern processors through technologies like Intel's Software Guard Extensions [3] or ARM's TrustZone [5]. Finally, there are hybrid BFT systems where the trusted parts are primarily implemented in software and for example include a proxy [107], a multicast ordering service [34, 35], a configuration service [119], or a virtualization layer [42, 54, 98, 122, 125].

As discussed in Section 8.2, trusted subsystems are essential to ensure that replicas are recovered even if they are compromised [27]. In addition, trusted components on several occasions have been used as part of a system's agreement stage. TrInc [77], for example, has shown that a small trusted counter for message attestation is sufficient to prevent a faulty leader from successfully performing equivocation, that is proposing different requests for the same sequence number to different followers without being detected. Building on this principle, other works reduced the minimum number of replicas required in the agreement stage and/or the number of protocol phases involved in reaching consensus [12, 31, 40, 67, 122, 123]. Relying on more complex trusted subsystems, Correia et al. [34, 35] presented a system design in which client requests are asynchronously distributed by untrusted system parts while their sequence number is determined by a synchronous trusted subsystem.

11 CONCLUSION

BFT state-machine replication is a powerful tool to provide resilience against arbitrary faults, but building actual BFT systems is inherently difficult due to the high number and complexity of alternatives and techniques [25]. In an effort to facilitate future research and implementations, this survey analyzed essential building blocks and mechanisms of BFT systems and discussed their respective benefits and limitations. Presenting a novel timeline-centric view of the agreement stage, the survey for example introduced a means to assess the composability of existing (and future) protocols. With regard to the execution stage, the analysis has shown that the common approach of imprecisely restoring replica states via checkpoints can easily lead to unintended consequences when checkpoint mechanisms are used in settings for which they were not specifically designed, such as systems with fewer replicas. Similar observations were made for optimistic techniques, which should only be applied if their advantages indeed justify the associated increase in complexity.

ACKNOWLEDGMENTS

The author thanks the reviewers for their valuable feedback, which has substantially helped to improve the quality of this article. In addition, he also thanks Michael Eischer and Laura Lawniczak for many insightful discussions on different aspects of Byzantine fault-tolerant systems. This work was partially supported by the German Research Council (DFG) under grant no. DI 2097/1-2.

REFERENCES

[1] Remzi Can Aksoy and Manos Kapritsos. 2019. Aegean: Replication beyond the Client-Server Model. In *Proceedings of the 27th Symposium on Operating Systems Principles (SOSP '19)*. 385–398.

[2] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. 2011. Prime: Byzantine Replication Under Attack. *IEEE Transactions on Dependable and Secure Computing* 8, 4 (2011), 564–577.

[3] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative Technology for CPU based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '13)*. 1–7.

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the 13th EuroSys Conference (EuroSys '18)*. Article 30, 15 pages.

[5] ARM. 2009. Security Technology Building a Secure System Using TrustZone Technology (White Paper).

[6] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2015. The Next 700 BFT Protocols. *ACM Transactions on Computer Systems* 32, 4, Article 12 (2015), 45 pages.

[7] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. 2013. RBFT: Redundant Byzantine Fault Tolerance. In *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS '13)*. 297–306.

[8] Algirdas Avižienis. 1985. The N-version Approach to Fault-tolerant Software. *IEEE Transactions on Software Engineering* 11, 12 (1985), 1491–1501.

[9] Amy Babay, John Schultz, Thomas Tantillo, Samuel Beckley, Eamon Jordan, Kevin Ruddell, Kevin Jordan, and Yair Amir. 2019. Deploying Intrusion-Tolerant SCADA for the Power Grid. In *Proceedings of the 49th International Conference on Dependable Systems and Networks (DSN '19)*. 328–335.

[10] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. 2019. *State Machine Replication in the Libra Blockchain*. Technical Report. Calibra.

[11] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2015. Consensus-Oriented Parallelization: How to Earn Your First Million. In *Proceedings of the 16th Middleware Conference (Middleware '15)*. 173–184.

[12] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2017. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys '17)*. 222–237.

[13] Christian Berger and Hans P. Reiser. 2018. Scaling Byzantine Consensus: A Broad Analysis. In *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers (SERIAL '18)*. 13–18.

[14] Christian Berger, Hans P. Reiser, João Sousa, and Alysson Bessani. 2019. Resilient Wide-Area Byzantine Consensus Using Adaptive Weighted Replication. In *Proceedings of the 38th International Symposium on Reliable Distributed Systems (SRDS '19)*. 183–192.

[15] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. *ACM SIGPLAN Notices* 41, 6 (2006), 158–168.

[16] Alysson Bessani, Eduardo Alchieri, João Sousa, André Oliveira, and Fernando Pedone. 2020. From Byzantine Replication to Blockchain: Consensus is only the Beginning. In *Proceedings of the 50th International Conference on Dependable Systems and Networks (DSN '20)*. 424–436.

[17] Alysson Bessani, Marcel Santos, João Felix, Nuno Neves, and Miguel Correia. 2013. On the Efficiency of Durable State Machine Replication. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC '13)*. 169–180.

[18] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMaRt. In *Proceedings of the 44th International Conference on Dependable Systems and Networks (DSN '14)*. 355–362.

[19] Alysson Neves Bessani, Hans P. Reiser, Paulo Sousa, Ilir Gashi, Vladimir Stankovic, Tobias Distler, Rüdiger Kapitza, Alessandro Daidone, and Rafael Obelheiro. 2008. FOREVER: Fault/intrusiOn REmoVal through Evolution & Recovery. In *Proceedings of the Middleware 2008 Conference Companion (Middleware '08 Poster Session)*. 99–101.

[20] Alysson Neves Bessani, Paulo Sousa, Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. 2008. The CRUTIAL Way of Critical Infrastructure Protection. *IEEE Security & Privacy* 6, 6 (2008), 44–51.

[21] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. 2003. Address Obfuscation: An Efficient Approach to Combat a Board Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium (USENIX Security '03)*. 105–120.

[22] Alexandra Boldyreva. 2003. Threshold Signatures, Multisignatures and Blind Signatures based on the Gap-Diffie-Hellman-Group Signature Scheme. In *Proceedings of the 6th International Workshop on Public Key Cryptography (PKC '03)*. 31–46.

[23] Ethan Buchman. 2016. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains.Thesis. University of Guelph.

[24] Christian Cachin, Simon Schubert, and Marko Vukolić. 2017. Non-Determinism in Byzantine Fault-Tolerant Replication. In *Proceedings of the 20th International Conference on Principles of Distributed Systems (OPODIS '16)*. Article 24, 16 pages.

[25] Christian Cachin and Marko Vukolic. 2017. Blockchain Consensus Protocols in the Wild (Keynote Talk). In *Proceedings of the 31st International Symposium on Distributed Computing (DISC '17)*. 1–16.

[26] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*. 173–186.

[27] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems* 20, 4 (2002), 398–461.

[28] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. 2003. BASE: Using Abstraction to Improve Fault Tolerance. *ACM Transactions on Computer Systems* 21, 3 (2003), 236–269.

[29] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos Made Live: An Engineering Perspective. In *Proceedings of the 26th Symposium on Principles of Distributed Computing (PODC '07)*. 398–407.

[30] Monica Chew and Dawn Song. 2002. *Mitigating Buffer Overflows by Operating System Randomization.* Technical Report CMU-CS-02-197. Carnegie Mellon University.

[31] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested Append-only Memory: Making Adversaries Stick to their Word. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP '07)*. 189–204.

[32] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. 2009. UpRight Cluster Services. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP '09)*. 277–290.

[33] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI '09)*. 153–168.

[34] Miguel Correia, Nuno Ferreira Neves, Lau Cheuk Lung, and Paulo Veríssimo. 2007. Worm-IT – A Wormhole-based Intrusion-tolerant Group Communication System. *Journal of Systems and Software* 80, 2 (2007), 178–197.

[35] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. 2013. BFT-TO: Intrusion Tolerance with Less Replicas. *Comput. J.* 56, 6 (2013), 693–715.

[36] Miguel Correia, Giuliana Santos Veronese, Nuno Ferreira Neves, and Paulo Veríssimo. 2011. Byzantine Consensus in Asynchronous Message-Passing Systems: A Survey. *International Journal of Critical Computer-Based Systems* 2, 2 (2011), 141–161.

[37] Domenico Cotroneo, Roberto Natella, Roberto Pietrantuono, and Stefano Russo. 2014. A Survey of Software Aging and Rejuvenation Studies. *ACM Journal on Emerging Technologies in Computing Systems* 10, 1 (2014), 1–34.

[38] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. 2006. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. 177–190.

[39] Christian Deyerl and Tobias Distler. 2019. In Search of a Scalable Raft-based Replication Architecture. In *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '19)*. 1–7.

[40] Tobias Distler, Christian Cachin, and Rüdiger Kapitza. 2016. Resource-efficient Byzantine Fault Tolerance. *IEEE Trans. Comput.* 65, 9 (2016), 2807–2819.

[41] Tobias Distler and Rüdiger Kapitza. 2011. Increasing Performance in Byzantine Fault-Tolerant Systems with On-Demand Replica Consistency. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys '11)*. 91–105.

[42] Tobias Distler, Rüdiger Kapitza, Ivan Popov, Hans P. Reiser, and Wolfgang Schröder-Preikschat. 2011. SPARE: Replicas on Hold. In *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS '11)*. 407–420.

[43] Tobias Distler, Rüdiger Kapitza, and Hans P. Reiser. 2010. State Transfer for Hypervisor-Based Proactive Recovery of Heterogeneous Replicated Services. In *Proceedings of the 5th "Sicherheit, Schutz und Zuverlässigkeit" Conference (SICHERHEIT '10)*. 61–72.

[44] Sisi Duan, Michael K. Reiter, and Haibin Zhang. 2018. BEAT: Asynchronous BFT Made Practical. In *Proceedings of the 25th Conference on Computer and Communications Security (CCS '18)*. 2028–2041.

[45] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (1988), 288–323.

[46] Michael Eischer, Markus Büttner, and Tobias Distler. 2019. Deterministic Fuzzy Checkpoints. In *Proceedings of the 38th International Symposium on Reliable Distributed Systems (SRDS '19)*. 153–162.

[47] Michael Eischer and Tobias Distler. 2018. Latency-Aware Leader Selection for Geo-Replicated Byzantine Fault-Tolerant Systems. In *Proceedings of the 1st Workshop on Byzantine Consensus and Resilient Blockchains (BCRB '18)*. 140–145.

[48] Michael Eischer and Tobias Distler. 2019. Scalable Byzantine Fault-tolerant State-Machine Replication on Heterogeneous Servers. *Computing* 101, 2 (2019), 97–118.

[49] Ian Aragon Escobar, Eduardo Alchieri, Fernando Luís Dotti, and Fernando Pedone. 2019. Boosting Concurrency in Parallel State Machine Replication. In *Proceedings of the 20th International Middleware Conference (Middleware '19)*. 228–240.

[50] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (1985), 374–382.

[51] Stephanie Forrest, Anil Somayaji, and David H. Ackley. 1997. Building Diverse Computer Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS '97)*. 67–72.

[52] Roy Friedman and Robbert Van Renesse. 1997. Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols. In *Proceedings of the 6th International Symposium on High Performance Distributed Computing (HPDC '97)*. 233–242.

[53] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. 2014. Analysis of Operating System Diversity for Intrusion Tolerance. *Software—Practice & Experience* 44, 6 (2014), 735–770.

[54] Miguel Garcia, Alysson Bessani, and Nuno Neves. 2019. Lazarus: Automatic Management of Diversity in BFT Systems. In *Proceedings of the 20th International Middleware Conference (Middleware '19)*. 241–254.

[55] Miguel Garcia, Alysson Neves Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. 2011. OS Diversity for Intrusion Tolerance: Myth or Reality?. In *Proceedings of the 41st International Conference on Dependable Systems and Networks (DSN '11)*. 383–394.

[56] Miguel Garcia, Nuno Neves, and Alysson Bessani. 2016. SieveQ: A Layered BFT Protection System for Critical Services. *IEEE Transactions on Dependable and Secure Computing* 15, 3 (2016), 511–525.

[57] Ilir Gashi, Peter Popov, Vladimir Stankovic, and Lorenzo Strigini. 2004. On Designing Dependable Services with Diverse Off-the-Shelf SQL Servers. In *Architecting Dependable Systems II*. Springer, 191–214.

[58] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *Proceedings of the 49th International Conference on Dependable Systems and Networks (DSN '19)*. 568–580.

[59] Gerhard Habiger, Franz J. Hauck, Johannes Köstler, and Hans P. Reiser. 2018. Resource-Efficient State-Machine Replication with Multithreading and Vertical Scaling. In *Proceedings of the 14th European Dependable Computing Conference (EDCC '18)*. 87–94.

[60] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. 2006. The Case for Byzantine Fault Detection. In *Proceedings of the 2nd Workshop on Hot Topics in System Dependability (HotDep '06)*. Article 5, 6 pages.

[61] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. 2007. PeerReview: Practical Accountability for Distributed Systems. In *Proceedings of 21st Symposium on Operating Systems Principles (SOSP '07)*. 175–188.

[62] William G. J. Halfond, Jeremy Viegas, and Alessandro Orso. 2006. A Classification of SQL Injection Attacks and Countermeasures. In *Proceedings of the International Symposium on Secure Software Engineering (ISSSE '06)*. 13–15.

[63] Franz J. Hauck, Gerhard Habiger, and Jörg Domaschka. 2016. UDS: A Novel and Flexible Scheduling Algorithm for Deterministic Multithreading. In *Proceedings of the 35th Symposium on Reliable Distributed Systems (SRDS '16)*. 177–186.

[64] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492.

[65] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Profile-Guided Automated Software Diversity. In *Proceedings of the 11th International Symposium on Code Generation and Optimization (CGO '13)*. 1–11.

[66] Yennun Huang, Chandra Kintala, Nick Kolettis, and N. Dudley Fulton. 1995. Software Rejuvenation: Analysis, Module and Applications. In *Proceedings of 25th International Symposium on Fault-Tolerant Computing (FTCS-25)*. 381–390.

[67] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: Resource-efficient Byzantine Fault Tolerance. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys '12)*. 295–308.

[68] Rüdiger Kapitza, Thomas Zeman, Franz J. Hauck, and Hans P. Reiser. 2007. Parallel State Transfer in Object Replication Systems. In *Proceedings of the 7th International Conference on Distributed Applications and Interoperable Systems (DAIS '07)*. 167–180.

[69] Manos Kapritsos and Flavio P. Junqueira. 2010. Scalable Agreement: Toward Ordering as a Service. In *Proceedings of the 6th Workshop on Hot Topics in System Dependability (HotDep '10)*. 7–12.

[70] Manos Kapritsos, Yang Wang, Vivien Quéma, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. 2012. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI '12)*. 237–250.

[71] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. 2003. Countering Code-Injection Attacks with Instruction-Set Randomization. In *Proceedings of the 10th Conference on Computer and Communications Security (CCS '03)*. 272–280.

[72] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2009. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems* 27, 4, Article 7 (2009), 39 pages.

[73] Ramakrishna Kotla and Mike Dahlin. 2004. High Throughput Byzantine Fault Tolerance. In *Proceedings of the 34th International Conference on Dependable Systems and Networks (DSN '04)*. 575–584.

[74] Petr Kuznetsov and Rodrigo Rodrigues. 2009. BFTW3: Why? When? Where? Workshop on the Theory and Practice of Byzantine Fault Tolerance. *SIGACT News* 40, 4 (2009), 82–86.

[75] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2010. Reconfiguring a State Machine. *SIGACT News* 41, 1 (2010), 63–73.

[76] Butler W. Lampson. 1983. Hints for Computer System Design. In *Proceedings of the 9th Symposium on Operating Systems Principles (SOSP '83)*. 33–48.

[77] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. 2009. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (OSDI '09)*. 1–14.

[78] Bijun Li, Nico Weichbrodt, Johannes Behl, Pierre-Louis Aublin, Tobias Distler, and Rüdiger Kapitza. 2018. Troxy: Transparent Access to Byzantine Fault-Tolerant Systems. In *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN '18)*. 59–70.

[79] Bijun Li, Wenbo Xu, Muhammad Zeeshan Abid, Tobias Distler, and Rüdiger Kapitza. 2016. SAREK: Optimistic Parallel Ordering in Byzantine Fault Tolerance. In *Proceedings of the 12th European Dependable Computing Conference (EDCC '16)*. 77–88.

[80] Jinyuan Li and David Maziéres. 2007. Beyond One-third Faulty Replicas in Byzantine Fault Tolerant Systems. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI '07)*. 131–144.

[81] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. 2016. XFT: Practical Fault Tolerance beyond Crashes. In *Proceedings of the 12th Conference on Operating Systems Design and Implementation (OSDI '16)*. 485–500.

[82] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*. 369–384.

[83] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. 2012. Multi-Ring Paxos. In *Proceedings of the 42nd International Conference on Dependable Systems and Networks (DSN '12)*. 1–12.

[84] Michael A. Marsh and Fred B. Schneider. 2004. CODEX: A Robust and Secure Secret Distribution System. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (2004), 34–47.

[85] Jean-Philippe Martin and Lorenzo Alvisi. 2004. A Framework for Dynamic Byzantine Storage. In *Proceedings of the 34th International Conference on Dependable Systems and Networks (DSN '04)*. 325–334.

[86] Jean-Philippe Martin and Lorenzo Alvisi. 2006. Fast Byzantine Consensus. *IEEE Transactions on Dependable and Secure Computing* 3, 3 (2006), 202–215.

[87] Michael G. Merideth. 2008. *Tradeoffs in Byzantine-Fault-Tolerant State-Machine-Replication Protocol Design.* Technical Report CMU-ISR-08-110. Carnegie Mellon University.

[88] Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology (CRYPTO '87)*. 369–378.

[89] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 23rd Conference on Computer and Communications Security (CCS '16)*. 31–42.

[90] Henrique Moniz, Nuno Ferreria Neves, Miguel Correia, and Paulo Veríssimo. 2008. RITAS: Services for Randomized Intrusion Tolerance. *IEEE Transactions on Dependable and Secure Computing* 8, 1 (2008), 122–136.

[91] Louise E. Moser, Peter M. Melliar-Smith, Priya Narasimhan, Lauren A. Tewksbury, and Vana Kalogeraki. 2000. Eternal: Fault Tolerance and Live Upgrades for Distributed Object Systems. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX '00)*. 184–196.

[92] André Nogueira, Miguel Garcia, Alysson Bessani, and Nuno Neves. 2018. On the Challenges of Building a BFT SCADA. In *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN '18)*. 163–170.

[93] National Institute of Standards and Technology. 2020. National Vulnerability Database. http://nvd.nist.gov/.

[94] Rafail Ostrovsky and Moti Yung. 1991. How to Withstand Mobile Virus Attacks. In *Proceedings of the 10th Symposium on Principles of Distributed Computing (PODC '91)*. 51–59.

[95] Marco Platania, Daniel Obenshain, Thomas Tantillo, Yair Amir, and Neeraj Suri. 2016. On Choosing Server- or Client-Side Solutions for BFT. *Comput. Surveys* 48, 4, Article 61 (2016), 30 pages.

[96] Marco Platania, Daniel Obenshain, Thomas Tantillo, Ricky Sharma, and Yair Amir. 2014. Towards a Practical Survivable Intrusion Tolerant Replication System. In *Proceedings of the 33rd International Symposium on Reliable Distributed Systems (SRDS '14)*. 242–252.

[97] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. 2015. Visigoth Fault Tolerance. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys '15)*. Article 8, 14 pages.

[98] Hans P. Reiser and Rüdiger Kapitza. 2007. Hypervisor-Based Efficient Proactive Recovery. In *Proceedings of the 26th Symposium on Reliable Distributed Systems (SRDS '07)*. 83–92.

[99] Ronald L Rivest, Adi Shamir, and Leonard Adleman. 1978. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM* 21, 2 (1978), 120–126.

[100] Rodrigo Rodrigues and Barbara Liskov. 2004. Byzantine Fault Tolerance in Long-Lived Systems. In *Proceedings of the 2nd Bertinoro Workshop on Future Directions in Distributed Computing (FuDiCo '04)*. Article 10, 3 pages.

[101] Rodrigo Rodrigues, Barbara Liskov, Kathryn Chen, Moses Liskov, and David Schultz. 2012. Automatic Reconfiguration for Large-Scale Reliable Storage Systems. *IEEE Transactions on Dependable and Secure Computing* 9, 2 (2012), 146–158.

[102] Tom Roeder and Fred B. Schneider. 2010. Proactive Obfuscation. *ACM Transactions on Computer Systems* 28, 2, Article 4 (2010), 54 pages.

[103] Signe Rüsch, Kai Bleeke, and Rüdiger Kapitza. 2019. Bloxy: Providing Transparent and Generic BFT-Based Ordering Services for Blockchains. In *Proceedings of the 38th International Symposium on Reliable Distributed Systems (SRDS '19)*. 305–314.

[104] John M. Rushby. 1981. Design and Verification of Secure Systems. In *Proceedings of the 8th Symposium on Operating Systems Principles (SOSP '81)*. 12–21.

[105] Rainer Schiekofer, Johannes Behl, and Tobias Distler. 2017. Agora: A Dependable High-Performance Coordination Service for Multi-Cores. In *Proceedings of the 47th International Conference on Dependable Systems and Networks (DSN '17)*. 333–344.

[106] Fred B. Schneider. 1990. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *Comput. Surveys* 22, 4 (1990), 299–319.

[107] Siddhartha Sen, Wyatt Lloyd, and Michael J. Freedman. 2010. Prophecy: Using History for High-Throughput Fault Tolerance. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI '10)*. 345–360.

[108] Victor Shoup. 2000. Practical Threshold Signatures. In *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques (EUROCRYPT '00)*. 207–220.

[109] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. 2008. BFT Protocols Under Fire. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI '08)*. 189–204.

[110] João Sousa and Alysson Bessani. 2015. Separating the WHEAT from the Chaff: An Empirical Design for Geo-Replicated State Machines. In *Proceedings of the 34th International Symposium on Reliable Distributed Systems (SRDS '15)*. 146–155.

[111] João Sousa, Alysson Bessani, and Marko Vukolić. 2018. A Byzantine Fault-tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform. In *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN '18)*. 51–58.

[112] João Sousa and Alysson Bessani. 2012. From Byzantine Consensus to BFT State Machine Replication: A Latency-Optimal Transformation. In *Proceedings of the 9th European Dependable Computing Conference (EDCC '12)*. 37–48.

[113] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. 2010. Highly Available Intrusion-Tolerant Services with Proactive-Reactive Recovery. *IEEE Transactions on Parallel and Distributed Systems* 21, 4 (2010), 452–465.

[114] Paulo Sousa, Alysson Neves Bessani, and Rafael R. Obelheiro. 2008. The FOREVER Service for Fault/Intrusion Removal. In *Proceedings of the 2nd Workshop on Recent Advances on Intrusion-Tolerant Systems (WRAITS '08)*. 1–6.

[115] Paulo Sousa, Nuno Ferreira Neves, and Paulo Veríssimo. 2006. Proactive Resilience through Architectural Hybridization. In *Proceedings of the 21st Symposium on Applied Computing (SAC '06)*. 686–690.

[116] Paulo Sousa, Nuno Ferreira Neves, and Paulo Veríssimo. 2007. Hidden Problems of Asynchronous Proactive Recovery. In *Proceedings of the 3rd Workshop on Hot Topics in System Dependability (HotDep '07)*. 5–9.

[117] Philip Thambidurai and You-Keun Park. 1988. Interactive Consistency with Multiple Failure Modes. In *Proceedings of the 7th Symposium on Reliable Distributed Systems (SRDS '88)*. 93–100.

[118] Gene Tsudik. 1992. Message Authentication with One-Way Hash Functions. *ACM SIGCOMM Computer Communication Review* 22, 5 (1992), 29–38.

[119] Robbert Van Renesse, Chi Ho, and Nicolas Schiper. 2012. Byzantine Chain Replication. In *Proceedings of the 16th International Conference On Principles Of Distributed Systems (OPODIS '12)*. 345–359.

[120] Paulo E. Veríssimo. 2006. Travelling through Wormholes: A New Look at Distributed Systems Models. *SIGACT News* 37, 1 (2006), 66–81.

[121] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2009. Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary. In *Proceedings of the 28th International Symposium on Reliable Distributed Systems (SRDS '09)*. 135–144.

[122] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2010. EBAWA: Efficient Byzantine Agreement for Wide-Area Networks. In *Proceedings of the 12th Symposium on High-Assurance Systems Engineering (HASE '10)*. 10–19.

[123] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. 2013. Efficient Byzantine Fault-Tolerance. *IEEE Trans. Comput.* 62, 1 (2013), 16–30.

[124] Marko Vukolić. 2015. The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication. In *Proceedings of the International Workshop on Open Problems in Network Security (iNetSec '15)*. 112–125.

[125] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. 2011. ZZ and the Art of Practical BFT Execution. In *Proceedings of the 6th European Conference on Computer Conference (EuroSys '11)*. 123–138.

[126] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. 2003. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP '03)*. 253–267.

[127] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. 2002. COCA: A Secure Distributed Online Certification Authority. *ACM Transactions on Computer Systems* 20, 4 (2002), 329–368.

[128] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. 2005. APSS: Proactive Secret Sharing in Asynchronous Systems. *ACM Transactions on Information and System Security* 8, 3 (2005), 259–286.

Received March 2020; revised September 2020; accepted November 2020