Micro Replication

Tobias Distler, Michael Eischer, and Laura Lawniczak
Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
Email: {distler,eischer,lawniczak}@cs.fau.de

***Abstract*—State-machine replication protocols represent the foundation of many fault-tolerant services. Unfortunately, their inherent complexity makes existing implementations notoriously difficult to debug and test. To address this problem, we propose a novel design approach, *micro replication*, whose main goal is to reduce bugs and enable replication protocols with improved debuggability properties. At its core, our concept consists of a set of principles that, if followed during protocol design, later significantly facilitate crucial tasks such as bug-source isolation, state-information retrieval, as well as root-cause identification. To achieve this, micro replication organizes a protocol as a composition of specialized modules ("micro replicas") that each encapsulate a particular protocol phase or mechanism, and therefore are easier to test and monitor than traditional monolithic replicas. Besides discussing the underlying ideas of our approach, to show its feasibility we also present and evaluate MIRADOR, the first micro-replicated Byzantine fault-tolerant protocol.**

***Index Terms*—Replication, debuggability, fault tolerance.**

I. Introduction

State-machine replication [1] is an essential building block of a variety of today's distributed systems, including coordination services [2]–[4], blockchains [5]–[7], firewalls [8]–[10], and file storage [11], [12]. Unfortunately, keeping a replicated service correct and available in the presence of server and network failures in itself is a difficult problem, causing replication protocols to be inherently complex. This is already true for crash fault tolerance (CFT), but even more so for Byzantine fault tolerance (BFT), and it makes debugging of such replication protocols a notoriously difficult task [13].

While debugging itself is a far-reaching area of research, in this paper we focus on three specific challenges that usually sooner or later arise when analyzing why a replication-protocol implementation does not behave according to specification: (1) As one of the first steps, it is typically crucial to be able to quickly limit the search space by isolating the area in which the bug is located. However, with traditional protocols implementing the entire protocol logic inside a set of monolithic replicas [11], [14], [15], this step in general is not straightforward since failures often affect an entire replica. (2) For a comprehensive analysis it is essential to retrieve information about the protocol's current state in order to precisely determine anomalous behavior. Log files can provide valuable insights in this regard, especially in the context of offline debugging, but they do not always accurately reflect a replica's actual runtime state. Dynamically inspecting replica state commonly requires additional instrumentation and is usually not integrated into replication libraries [11], [15]–[17]. (3) Having gained a deeper understanding of the problem, a third important aspect of debugging is to pinpoint the root cause of a bug. Even when knowing the general area in which to look, this step in practice is often times complicated by the fact that existing replica implementations are multi-threaded and thus represent a complex combination of heterogeneous mechanisms (e.g., consensus, checkpointing, leader election).

To address these challenges we propose *micro replication*, a novel approach that aims at improving the debuggability of state-machine replication protocols, ideally avoiding bugs in the first place. In a nutshell, the main idea behind micro replication is to design a protocol as a collection of clusters that each are responsible for a different protocol task and for this purpose comprise tiny modules, the micro replicas. As a key benefit of such a system design, the blast radius [18] of a failure in many cases is limited to the affected micro replica, thereby reducing the search space. In contrast to replicas in traditional protocols, micro replicas communicate by issuing queries to each other. This pull-based interaction makes it straightforward to retrieve up-to-date state information for specific protocol steps by building debuggers that use the exact same queries and interfaces as regular replicas. Furthermore, root-cause identification is simplified by the fact that all micro replicas are single-threaded and share a common architecture.

Please note that in this paper we do not address the problem of how to run effective message-replay or fault-injection campaigns. Being able to reproduce a failure and to reveal previously undiscovered issues are both crucial parts in the development process, which is why several tools [19]–[27] exist that can be used in such campaigns. Our work, on the other hand, approaches debuggability from the protocol perspective, focusing on how to design a replication protocol in such a way that with the help of these kinds of tools it becomes easier to isolate and identify the source of a bug.

In particular, this paper makes the following contributions: (1) It introduces micro replication as an approach to design replication protocols with improved debuggability properties. (2) It presents MIRADOR, the first state-machine replication protocol that incorporates the micro-replication principles. (3) It evaluates MIRADOR in comparison to BFT-SMaRt [16].

II. Background and System Model

State-machine replication protocols offer clients access to a service that for robustness is distributed across multiple replicas, each hosting an instance of the application state [1]. To ensure consistency, replicas run an agreement protocol that assigns unique sequence numbers to the commands issued by clients. The specifics of the agreement protocol vary between systems. A common approach is to model the agreement process as a series of views, and for each view elect a leader replica to make proposals that are then accepted by the other replicas [11], [28]. Having reached consensus, replicas use the sequence numbers to all execute the commands in the same order. To support the dynamic replacement of replicas and allow trailing replicas to catch up, most protocols enable replicas to checkpoint their copy of the application state and transfer such a snapshot to another replica on demand.

Nodes (i.e., clients and replicas) interact with each other by sending messages over an unreliable, asynchronous network. Transient network failures may temporarily prevent nodes from communicating with each other, but if a node repeatedly sends a message, it will eventually arrive at the receiver. Some protocols require messages to be authenticated in order to allow a receiver to unequivocally identify a message's sender.

For these circumstances, CFT protocols are designed to tolerate a configurable maximum number of crashed replicas and an unbounded number of client crashes [14], [15]. BFT protocols, on the other hand, are resilient against arbitrary behavior of faulty/malicious replicas or clients [11], [29].

III. Problem Statement

In this section, we discuss three key challenges that typically arise during the debugging of replicated systems and analyze them in the context of state-of-the-art replication protocols. For comparison, we also provide an intuition of how we tackle each of these challenges with our micro-replication approach.

Challenge #1: Limiting the Search Space

One of the first major steps after observing faulty system behavior is to locate and isolate the source of the problem. In complex software architectures such as state-machine replication protocols, this narrowing down of the search space is crucial since a thorough analysis of the entire replicated system is usually time-consuming, sometimes even unfeasible.

State of the Art. Unfortunately, existing state-machine replication protocols [11], [14], [15] make it often difficult to pinpoint the location of a bug due to their designs commonly being based on a small set of versatile replicas [30]. Specifically, to participate in a traditional protocol each replica typically needs to handle a variety of tasks such as communicating with clients, electing a leader for the agreement process, proposing sequence numbers for commands, committing commands, executing commands, and checkpointing the application state; note that this list is not exhaustive. As illustrated in Figure 1, this plethora of responsibilities in general results in systems with complex interaction patterns among a small number of

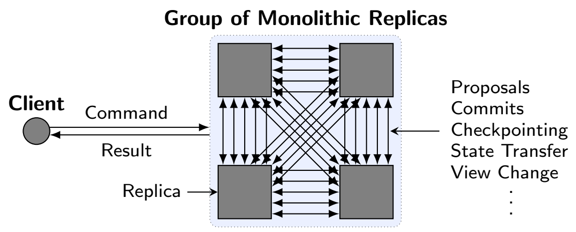


Figure 1. Traditional approach to state-machine replication

monolithic replicas in which especially the associated interference between different protocol parts is a common source of bugs (see Section VI-A). Not surprisingly, in such architectures it is often times not straightforward to precisely identify the particular protocol phase or mechanism that causes an issue. This problem is complicated by the fact that in monolithic replica implementations different protocol parts commonly share the same data structures for efficiency reasons [11], [16], thereby increasing the probability that a failure observed in one part is actually the result of a bug located elsewhere.

Our Approach. To avoid the problems associated with the traditional monolithic-replica designs, we separate a replication protocol into tiny single-purpose modules called *micro replicas*. Since each type of micro replica represents only a particular task of the overall protocol, and with micro replicas running in isolation of each other, interference-related bugs are eliminated and determining the source of other bugs generally becomes easier than in state-of-the-art protocol architectures. Of course, even with our approach it is still possible for a bug in one phase to manifest in another (e.g., a faulty leader output may be observed by other replicas), however the modularization by definition rules out many forms of hard-to-detect propagation issues due to micro replicas not sharing any data structures. Thanks to the inherent isolation between micro replicas, compared with traditional protocols our approach is able to significantly reduce the blast radius of a failure, and thereby often decisively limits the debugging search space.

Challenge #2: Facilitating Information Retrieval

Knowing the general area in which to look for a bug in many cases is not sufficient to exactly locate it, let alone to fix it. For these steps, it is typically necessary to obtain and analyze information about protocol state related to the problem, including for example the values of a replica's internal data structures or the contents of transmitted messages.

State of the Art. A common way to retrieve such knowledge in existing replicated systems is the use of log files that are kept by each replica and represent a transcript of important events, state changes, and network interactions. Despite overall being a powerful means for debugging, in the context of replicated systems log files have two important limitations: (1) A replica's log file does not necessarily represent the current state of the replica. This is due to the fact that the update of a state variable and the addition of a corresponding entry to the log in most implementations are two independent procedures. For example, if as result of a programming error the logged information does not represent the actual state change, or if a variable is updated without the change being logged at all, then the resulting log file no longer matches the replica state. Similar observations can be made with regard to discrepancies between the content of a message that is sent to another replica and the message content that appears in the log. (2) Although primarily effective as a means to analyze a problem after a failure has occurred, logs are not ideal when it comes to debugging a system while it is still running. Thus, the retrieval of online information typically requires additional debugging interfaces and thus in general is not supported in existing replication protocol implementations [11], [15]–[17].

Our Approach. Micro replicas interact by directly querying each other for their current states. In contrast to the push-based patterns applied in traditional replication protocols, in which replicas propagate state changes by unilaterally distributing messages, the pull-based interaction in our approach makes it straightforward to retrieve up-to-date replica-state information at runtime. All an external debugger[1] needs to do is to send the same queries as regular replicas. Without the need for additional interfaces, this makes it possible for a debugger to limit the examination to individual micro replicas of interest and perform it in a way that does not affect the micro-replicated system outside of the debugging process. As a result, the offline analysis based on log files (which of course remains invaluable) can be complemented with an online component.

Challenge #3: Simplifying Root-Cause Identification

Determining a problem's actual cause within a failing protocol part often requires deeper insights into the implementation and the possibility to repeatedly trigger the failure.

State of the Art. Today's replicated systems commonly represent heterogeneous compositions of tailored software components in which the replication logic is distributed across multiple threads [3], [15], [16]. While highly efficient, with respect to debuggability such an internal structure has drawbacks, especially when it comes to the time it takes to examine third-party code and the reproducibility of concurrency bugs.

Our Approach. Independent of the specific tasks they are responsible for, all micro replicas implement the same work flow and internal replica architecture. The execution of a micro replica is strictly single-threaded, thereby avoiding many concurrency-related issues in the first place. Altogether, the standardized structure and behavior of micro replicas not only enables systematic debugging strategies but also makes it easier to write unit tests for each individual micro replica.

IV. Micro Replication

To address the challenges discussed in Section III we propose *micro replication*, a new paradigm for the specification and implementation of state-machine replication protocols that primarily aims at improving their debuggability. Given this focus, efficiency and high performance are only secondary concerns, although based on our experiments (see Section VI) we are confident that it is possible to develop micro-replicated protocols which are also competitive in these categories.

As illustrated in Figure 2, to circumvent the drawbacks associated with monolithic replicas, our approach designs a system as a composition of specialized services that each represent an atomic task required for state-machine replication (e.g., communicating with clients, starting the agreement process, determining whether a checkpoint is stable). For fault tolerance, each of the services is provided by a dedicated cluster of micro replicas that all concentrate on the same task.

[1]In this paper, we use "debugger" as generic term for an external (software and/or hardware) tool that may be used to assist in the debugging process.

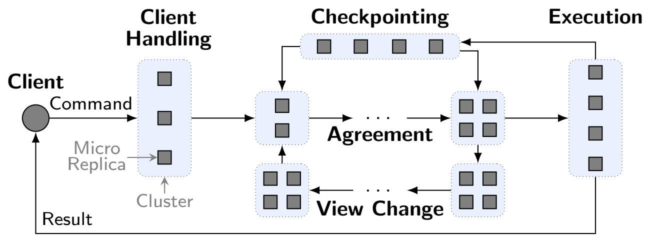


Figure 2. Composition of micro-replica clusters

A. Principles

Micro replication is guided by three principles whose main purpose is to make the inherent complexity of state-machine replication manageable by designing protocols in a standardized manner. The three micro-replication principles are:

Limited Replica Functionality. To minimize the blast radius of failures, each micro replica should only provide the functionality required for the task the replica is responsible for. In similar form, this idea can be found in traditional modularization concepts such as service-oriented architectures [31] or microservices [32], however to the extent we envision it, the concept so far has not been applied to structure state-machine replication protocols. In particular, we aim at a degree of modularization that is significantly more fine-grained than, for example, the proposer/acceptor/learner scheme used in many existing replication protocols such as Paxos [33].

With each micro replica implementing a single task and being directly accessible over the network, it is straightforward to first test and later monitor specific protocol steps in isolation. Furthermore, in addition to facilitating debugging, the separation of individual protocol steps into dedicated modules is also already beneficial during the design process of a protocol. Specifically, it forces protocol designers to identify and externalize the dependencies between different isolated protocol mechanisms, and thereby makes it easier to reason about the correctness of a protocol while developing it.

Loosely Coupled Clusters. For resilience, micro replication handles each protocol step on a cluster of micro replicas that are all dedicated to the same task. Overall, the clusters are loosely coupled and may comprise different numbers of micro replicas. Technically, it is even possible to build micro-replicated protocols in which different clusters offer different fault-tolerance guarantees (e.g., protecting crucial protocol steps against Byzantine faults while tolerating crashes in others), however exploring such opportunities is future work.

As a general rule, micro replicas only communicate with replicas of clusters that are responsible for tasks they depend on. That is, it is sufficient for each replica to know its immediate neighbors handling the previous, (in some cases) the same, as well as the next protocol step. In contrast to replicas in many traditional protocols, in particular from the BFT domain [8], [11], [34], [35], micro replicas do not directly forward or include messages that they themselves have received from others. Instead, when providing input for the next protocol step, a micro replica always does this by creating and sending a new message with the relevant values. This has the key advantage of acting as an additional barrier against the propagation of faults, thereby further reducing the blast radius.

To withstand server failures, replicas of the same cluster must not be hosted by the same machine. Apart from that, there are no restrictions on replica placement. For example, all replicas of a system may be distributed across a large number of servers to spread out the load, or they can run on a small set of machines to save resources. Micro replicas assigned to the same machine may be executed in separate processes, dedicated threads, or share the same thread, depending on the degree of fault isolation targeted for a particular use case.

Standardized Replica Work Flow. To further improve debuggability, as third principle micro replication demands replicas to all follow the same work flow. Specifically, a replica must (1) periodically query its counterparts of an upstream cluster for opinions on the outcome of the previous protocol step, (2) process the collected opinions according to a set of task-specific rules, thereby determining and storing the result of its own protocol step, and (3) provide this result to downstream replicas on request. That is, unlike replicas in traditional protocols [11], [15], [28], which must actively take care of distributing their outputs in each protocol phase, micro replicas fetch their inputs and store the corresponding outputs locally, to be delivered later when another replica asks for them.

Specifying replica interactions in a standardized and pull-based manner has several advantages. First, as further detailed in Section IV-B, the standardized approach enables micro replicas of all protocol stages to share a common basic architecture, which significantly simplifies orientation during the search for bugs as well as the implementation of testing, debugging, and monitoring tools. Furthermore, the pull-based communication makes it straightforward to precisely select a group of target replicas to examine, and to dynamically retrieve relevant information about their states without requiring additional instrumentation or debug interfaces. While the system is running and without further replica-code modifications, an external debugger for example can directly investigate the states of specific replicas by sending the same regular requests as the replicas' downstream neighbors. Once the debugging process is complete, the debugger discontinues its queries, resulting in no further runtime overhead outside of debugging sessions.

B. Micro Replicas

Independent of their individual responsibilities, to ease orientation and improve implementability all micro replicas share the same internal structure sketched in Figure 3.

Basic Architecture. Representing a certain step in a replication protocol, a micro replica's task is to obtain a set of opinions on the outcome of the previous protocol step (`inputs`) and based on them reach a decision on the outcome of its own protocol step (`outputs`). At all times, a micro replica keeps input and output information for a limited window of sequence numbers (see Lines 1–2). Each window has a lower bound (`min`), an upper bound (`max`), as well as a position attribute `pos` marking the sequence number of the lowest empty slot. Micro replicas always fill their windows from bottom to top, without leaving sequence-number gaps. The rationale behind this design decision is to reduce complexity by eliminating corner cases such as out-of-order commits which tend to make protocol implementations more difficult [15].

```
                         Micro-Replica State
 1  WINDOW<INPUTVALUE>[] inputs;                      // Array of arrays
 2  WINDOW<OUTPUTVALUE> outputs;                      // Array
                   Periodic Queries at Upstream Replicas
 3  Periodically:
 4    For each upstream replica u:
 5      if(inputs[u] is full) continue;
 6      Query u for RANGE r := [inputs[u].pos, inputs[u].max];
                    Responses from Upstream Replicas
 7  On receiving SEQUENCE<INPUTVALUE> s from upstream replica u:
 8    // Check and store input
 9    if(s cannot be appended to inputs[u]) return;
10    inputs[u].append(s);
11
12    // Determine output and invalidate the corresponding input slots
13    for(SEQNR s in [outputs.pos, outputs.max]:
14      OUTPUTVALUE decision := Process values inputs[*][s];
15      if(decision == nil) break;
16      outputs[s] := decision;
17      For each upstream replica x: inputs[x][s] := ♣;
                   Requests from Downstream Replicas
18  On receiving RANGE r from downstream replica d:
19    Send SEQUENCE<OUTPUTVALUE> s := outputs.seq(r) to d;
                                Events
20  On learning garbage-collection threshold g:
21    Move lower window bounds in inputs and outputs to g;
```

Figure 3. Basic internal architecture of a micro replica

To acquire new inputs, a micro replica periodically sends requests to its upstream replicas (L. 3–6). In this context, the replica only queries another replica `u` for the range of sequence numbers that are still undecided and for which replica `u` has not yet provided inputs. If there are no such window slots, there is no need to send a request. Upstream replicas respond to queries with a sequence of values (L. 7–17), of which the replica appends those that fit into the window and do not conflict with the no-gap rule. If a response does not include such values, for example due to having been delayed and carrying old state, a replica simply ignores the message. Whenever storing new input values, a replica checks whether it is now able to decide on the outcome of the next pending sequence-number slot (L. 13–17). The specifics of this step depend on the replica's overall task and usually involve the collection of a predefined number of matching input values from different upstream replicas. If a decision is reached, the replica *accepts* the value and inserts it to its output window. Furthermore, it invalidates all corresponding input slots of the affected sequence number by inserting a special ♣ value to exclude the slot from upcoming periodic input queries.

Once values are part of the output window, downstream replicas can fetch them via range queries (L. 18–19). If its output window is full, a replica waits until receiving permission to shift its windows to higher sequence numbers (L. 20–21), relying on some form of garbage-collection mechanism, which typically involves the creation of checkpoints. Such procedure is omitted in Figure 3 but described in detail in Section V-B.

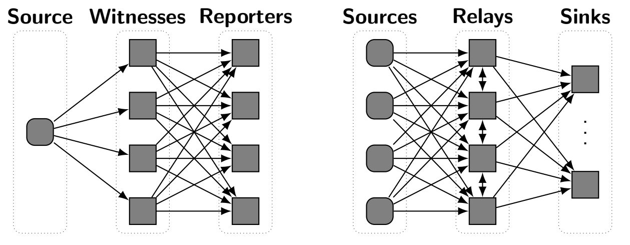
Variants. Starting from the basic architecture described above, micro replicas can be tailored to fit the needs of the particular protocol step they represent. In some cases, this means to add further input and/or output windows, while in others it is sufficient for a replica to manage a single, sequence-number-independent output value (see Section V-B). Apart from that, a micro replica may be specified to switch between different modes that influence the replica's objectives and behavior. While in normal mode, an agreement micro replica for example aims at handling the leader's proposals, whereas in view-change mode the replica should focus on switching to the new leader before resuming the normal mode again.

Accelerators. The requirement to periodically query its upstream replicas for input values does not prohibit a micro replica from additionally engaging in other forms of interaction. In particular, this makes it possible to optimize protocol latency by introducing push-based mechanisms we refer to as *accelerators*. Using an accelerator, whenever a replica accepts a new output value, it proactively forwards this value to its downstream replicas by sending the response the replica would have provided if it had received a corresponding query. The downstream replica stores the value if it fits into the input window, otherwise the replica ignores the message. In contrast to the regular pull-based interaction, accelerators are a best-effort mechanism that attempts to transmit each value only once and without guaranteeing eventual delivery at the receiving replica. Hence, accelerators allow micro replicas to poll their upstream neighbors less frequently, but they cannot (and should not) replace pull-based queries as the main mechanism ensuring the liveness of a micro-replicated protocol.

C. Architectural Building Blocks

Analyzing existing protocols with respect to a possible micro-replication-compliant decomposition, we identified several architectural patterns that each span multiple micro-replica clusters and can be reused across different protocols or even at different locations within the same protocol. Like the standardized internal replica architecture presented in Section IV-B, such multi-cluster building blocks have the benefit of reducing complexity, simplifying the debugging process, and making it easier to implement tooling support. To illustrate the basic concept, in the following we provide details on two important architectural patterns used in MIRADOR, our first micro-replicated protocol which is further described in Section V. Both patterns were designed for application in a Byzantine fault-tolerant context where up to f replicas per cluster may fail in an arbitrary way. The first pattern targets safety, whereas the second pattern primarily addresses liveness.

Reliable Distribution Pattern. This architectural pattern ensures that information provided by a single, potentially unreliable source (e.g., a leader replica) is distributed across a sufficiently large number of correct replicas in order to not get lost. Consequently, it serves the same purpose as reliable broadcast mechanisms in traditional replication protocols (cf. PBFT's pre-prepare and prepare phase [11]).



(a) Reliable distribution pattern (b) Relay pattern

Figure 4. Examples of multi-cluster building blocks

As shown in Figure 4a, the pattern includes two clusters comprising $3f + 1$ micro replicas each. Members of the first cluster ("witnesses") are responsible for querying the source for a value for each window slot. If the source behaves correctly, all witnesses will (1) eventually receive a value for each slot and (2) all observe the same values for the same slots. In contrast, a faulty source may provide different replicas with different values or no values at all, thereby possibly causing inconsistent opinions on the values put out by the source. This problem is addressed by the second cluster ("reporters"). Reporters periodically collect the opinions of witnesses and accept a value as soon as $2f+1$ different witnesses have reported identical values for the same sequence number. This rule ensures that two correct reporters cannot accept different values, as their quorums of witnesses intersect in at least one correct witness, which supplies all reporters with the same value.

If the source is correct, the reliable distribution pattern ensures that all correct reporters will eventually accept the same value even if up to f witnesses are arbitrarily faulty. On the other hand, a faulty source in combination with faulty witnesses may cause some correct reporters to obtain $2f + 1$ matching opinions and reach a decision for a window slot, while other correct reporters in the cluster might not. If such behavior is not acceptable for a protocol part, the reliable distribution pattern can be combined with the relay pattern.

Relay Pattern. As shown in Figure 4b, this pattern targets scenarios with multiple sources of which up to f may be faulty and distribute arbitrary values. The other sources are correct and either put out the correct values or (if they do not know the correct values) do not respond to queries. Besides, there are multiple micro replicas ("sinks") which rely on the source values as inputs. Given these circumstances, the purpose of the pattern is to ensure that (1) a correct sink only accepts a value provided by a correct source and (2) if a correct sink accepts a value, then all correct sinks eventually accept the same value.

To meet this requirement, the relay pattern relies on a cluster of $3f + 1$ replicas ("relays") that seek to learn a new value via two different ways. On the one hand, they periodically query the sources for input and accept a value after having obtained $2f + 1$ matching opinions, thereby guaranteeing the value to be correct. On the other hand, in parallel they ask other relays for their decisions and accept a value as soon as it is supported by $f + 1$ relays, which means that at least one correct relay has previously obtained and accepted the same value directly from the sources. Correct sinks accept a value after having learned it from $2f+1$ relays. Apart from ensuring the correctness of the value, this also guarantees the existence of at least $f+1$ correct relays that have already accepted the value and are consequently able to assist other relays in learning it. As a result, there will eventually be $2f+1$ correct relays enabling each correct sink to obtain the correct value.

Out of itself, the relay pattern does not guarantee that correct sinks are actually able to reach a decision. Whether this is in fact possible depends on the behavior of the sources. The relay pattern only ensures that once a correct sink accepts a value, eventually all other correct sinks will do the same.

V. MIRADOR

This section presents MIRADOR, the first state-machine replication protocol that follows the micro replication principles. MIRADOR tolerates up to f Byzantine faults in each cluster and relies on cluster sizes of at most $3f+1$ micro replicas. We opted for a BFT protocol because (1) recent years have brought a variety of industrial use cases such as permissioned blockchains [5]–[7] or SCADA systems [36]–[38] and (2) the increased complexity of BFT protocols makes good debuggability an even higher concern than in CFT protocols.

Designing MIRADOR, our goal was to build on established BFT concepts to be able to study how they integrate with micro replication. For this reason, for consensus MIRADOR for example relies on the traditional three-phase algorithm introduced by PBFT [11]. Leveraging this and other existing ideas enabled us to focus on the main contribution of this section, which is the decomposition of the entire state-machine replication protocol into micro-replica clusters, thereby reaching a degree of modularization that is much more fine-grained than in any state-of-the-art protocol. In total, MIRADOR consists of 14 clusters (including the client; 6 of the clusters are on the critical path), which is why for clarity in the following we focus our discussion on the most important parts. For the MIRADOR specification and source code please refer to [39].

A. Command Handling

As illustrated in Figure 5, MIRADOR is organized into three main stages that each consist of micro-replica clusters with different responsibilities. The *front-end stage* acts as contact cluster for clients and receives incoming commands. Next, the *agreement stage* then establishes a stable total order on these commands by assigning unique sequence numbers to them. Finally, the *execution stage* processes the sequence of agreed commands and produces the corresponding results. In the following, we discuss each of the stages in more detail.

Front-End Stage. To invoke an operation o in MIRADOR, a client c creates a command $m = \langle c, t, o \rangle$ using a command sequence number t that is generated from a client-specific and monotonically increasing counter. To submit a command to the replication protocol, the client sends it to a cluster of $2f+1$ *front-end* micro replicas. For this step, MIRADOR offers two different methods: a pull-based mechanism that enables front-end replicas to query clients for new commands, and an accelerator-based mechanism that allows clients to push new

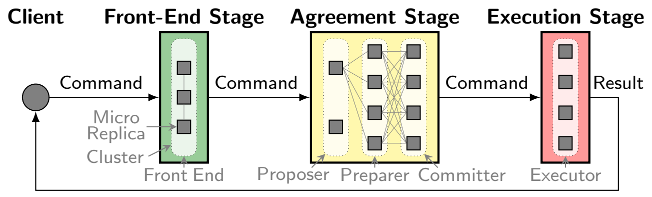


Figure 5. MIRADOR clusters involved in handling commands

commands once they become available. The latter is useful to support push-based legacy clients and has the additional advantage of freeing front-end micro replicas from the need to query (a potentially large number of) clients at high frequency.

Front ends check the validity of each incoming command by verifying that the command is well-formed and that the client has the necessary privileges to invoke the corresponding operation. If all of these checks are successful, a front-end replica accepts the command and stores it as one of its own output values. For this purpose, a front end maintains multiple windows, one for each client, in which the incoming commands are kept in increasing order of command sequence numbers. Through these windows, the commands are made available to the rest of the system (e.g., the agreement stage) on request.

Besides communicating with clients, micro replicas of the front-end cluster also periodically query each other for new commands. This is to ensure that if a valid command for a number t reaches at least one correct front end, then all correct front ends will eventually learn of a command being available for t, even if the client crashes in the meantime or refuses to upload its command to all correct front-end replicas. If a faulty client submits diverging commands for the same t, correct front ends may end up storing different versions, but only one of them will be considered by the agreement stage.

Agreement Stage. MIRADOR's agreement stage defines a stable total order on commands by relying on the three-step algorithm that previously served as basis for a multitude of BFT protocols [8], [11], [34]. Translated into a micro-replicated protocol element, it implements the reliable distribution pattern from Section IV-C and operates as follows. The agreement process is initiated by a dedicated leader micro replica of the *proposer* cluster; the specific replica to act as reigning proposer is determined by the current view (see Section V-C). To obtain new commands, the proposer repeatedly queries all front ends. Whenever the proposer learns of a new and valid command, it appends the command to its local output window, thereby assigning a unique agreement sequence number.

From there, micro replicas of the subsequent *preparer* cluster fetch the commands and verify the proposer's decisions by checking that the proposed commands (1) are valid (according to the same criteria as used by front-end and proposer replicas) and (2) have not already been proposed before. Only if both of these requirements are satisfied, a correct preparer accepts a proposal and stores the command in its own output window.

On request, preparers send their outputs to the next cluster: the *committers*. Each committer compares the obtained preparer outputs for each agreement sequence number and accepts

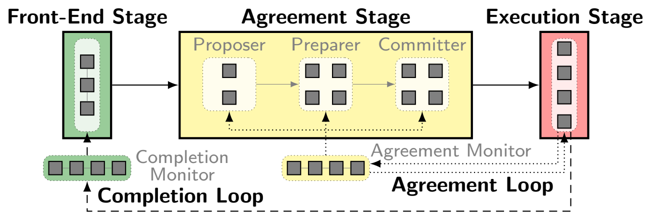


Figure 6. MIRADOR flow-control loops

a command after it has received $2f+1$ matching opinions for the same view from different preparer replicas. This ensures that within a view correct committers will never accept diverging proposals for the same agreement sequence number.

Execution Stage. MIRADOR's execution stage consists of a cluster of *executors* that are responsible for processing committed commands in the order of their associated agreement sequence numbers. To learn of the outcome of the agreement process, an executor repeatedly queries the committer replicas for their outputs and considers a command committed once at least $2f+1$ different committer replicas in a view reported the same value for a particular agreement sequence number.

If a client has a pending command, it periodically queries the executors for the result and accepts the result once $f+1$ different executors returned the same value; alternatively, for legacy clients MIRADOR also offers a mechanism to push results from executors to clients. In contrast to existing state-machine replication protocols [11], [35], clients in MIRADOR do not need to protect their commands with timeouts and in particular are not required to rebroadcast pending commands when the timeout expires in order to assist the server side in triggering a view change. Instead, MIRADOR's front-end stage ensures that once a command has reached a correct front end, the command will eventually be handled by the agreement stage and processed on at least $f+1$ correct executor replicas.

B. Flow Control and Checkpointing

To improve efficiency, MIRADOR does not wait for a command to be executed before agreeing on the next, but instead enables replicas to handle multiple commands concurrently. For this purpose, all micro replicas of Section V-A possess fixed-size windows that define the range of (command or agreement) sequence numbers a replica needs to maintain. As a crucial side effect, the bounded windows prevent replicas from becoming overloaded. In the following, we explain MIRADOR's flow-control mechanism for shifting these windows.

Basic Approach. An upstream micro replica maintains its output values so that downstream clusters can use them as inputs for their respective tasks. As a consequence, micro replicas must keep their outputs as long as this information is needed by others. With the main clusters in MIRADOR being organized as a chain, this means that outputs can be garbage collected once the last affected cluster in the chain approves the deletion. To safely and reliably relay such permission MIRADOR comprises multiple control loops (see Figure 6) that all implement the relay pattern presented in Section IV-C.

The sources in this pattern are represented by the members of the cluster giving the approval to garbage collect values up to a certain sequence number. The roles of relays are assigned to a dedicated cluster of *monitor* micro replicas that periodically collect the latest opinions on the garbage-collection threshold from their sources as well as each other. With the threshold dynamically changing, a validation by direct comparison of the provided opinions is not feasible. Each monitor therefore determines its accepted threshold by selecting the maximum of two computed sequence numbers g_s (the $2f+1$ highest threshold obtained from sources) and g_m (the $f+1$ highest threshold announced by monitors). Determining the garbage-collection threshold this way guarantees that at least $f+1$ correct source replicas approve the deletion.

The sinks of a control loop are all upstream replicas that are interested in getting the permission to shift their windows. They compute the garbage-collection threshold by selecting the $2f+1$ highest sequence number obtained from different monitors. Thanks to the relay pattern, this ensures that if one correct upstream micro replica observes a certain threshold and acts on it, eventually all correct upstream micro replicas will act accordingly, resulting in the replicas to stay in sync.

Specific Control Loops. As shown in Figure 6, MIRADOR performs flow control using two control loops. The *completion control loop* ensures that micro replicas in the front-end stage only discard a command after it has been processed by a sufficient number of executors. As front ends manage their outputs in independent client-specific windows, the garbage-collection threshold in the completion control loop is represented as a vector $\vec{g}$ containing a separate threshold g_c for each client c. Each of these thresholds reflects the highest processed command sequence number of the respective client and is individually handled according to the rules described above.

In the agreement stage, the *agreement control loop* is responsible for keeping consensus-related information available until the agreement process for a sequence number is complete. For this purpose, executors announce a garbage-collection threshold in the form of a single sequence number, which after verification by a cluster of agreement monitors is used as lower window bound by all agreement-stage replicas.

Checkpoints. With monitor micro replicas accepting garbage-collection thresholds once $2f+1$ of the $3f+1$ downstream replicas have approved them, it is possible that upstream replicas discard values before they were fetched by all correct downstream replicas. Consequently, before increasing its announced threshold a downstream replica must be sure that there are means for other correct replicas to skip sequence numbers in case they have fallen behind, for example when rejoining the system after a temporary network partition.

Depending on the specific characteristics of a cluster, MIRADOR addresses this problem in two different ways: (1) For front ends whose tasks for a command sequence number t can be correctly performed without knowledge of the effects of previous sequence numbers $t' < t$, no further action is necessary. Trailing replicas can simply increase their window bounds after learning a higher garbage-collection threshold. (2) In stages handling tasks for which such dependencies exist (e.g., the execution stage), micro replicas regularly checkpoint all relevant state and on request forward a checkpoint to a trailing replica; the replica accepts and applies the checkpoint after verifying it based on $f+1$ matching opinions.

To minimize overhead, micro replicas in MIRADOR do not create checkpoints for every sequence number but in predefined intervals. When a checkpoint is due, a replica creates a snapshot of its state and stores the checkpoint locally. When asked by a monitor for the garbage-collection threshold, a replica reports the sequence number of the latest checkpoint. With monitors accepting the $2f+1$ highest value they obtained from different replicas (see above), a correct monitor only supports a garbage-collection threshold if at least $f+1$ correct replicas have created a checkpoint for the sequence number. Due to the fact that correct replicas produce matching checkpoints for the same sequence number, this is sufficient to ensure that (if necessary) trailing replicas will later be able to fetch the checkpoint and verify its correctness.

A replica knows that it is trailing once it learns of the existence of a garbage-collection threshold g that is higher than the sequence number for which the replica is currently seeking inputs. At this point, a trailing replica stops querying upstream replicas for normal-case inputs and instead focuses on obtaining a checkpoint for sequence number g and verifying it based on $f+1$ matching opinions. If in the meantime the replica obtains knowledge of a higher threshold $g' > g$, the replica restarts the entire process for sequence number g'.

C. View Change

As a leader-based protocol, MIRADOR comprises means to reassign the proposer role to a different replica. Similar to traditional monolithic BFT protocols [11], [40], the main task of such a view-change mechanism in MIRADOR is to ensure that already agreed commands keep their assigned agreement sequence numbers across views. Specifically, if at least one correct executor has observed a command as committed, then the command is guaranteed to be the agreed value for the respective sequence number in the current view and all higher views. Following the principles of micro replication, MIRADOR's view-change mechanism is distributed across a set of micro-replica clusters with dedicated responsibilities.

Announcing a New View. As illustrated in Figure 7, MIRADOR's view-change process is initiated by a cluster of $2f+1$ *controller* micro replicas whose primary task is to determine and announce the system's current view. During normal-case operation, each controller for this purpose continuously checks whether all commands that newly arrive at the front-end cluster are indeed accepted (and thus processed) by the executor cluster within a certain period of time. As long as this is the case, the current view is operational and controllers take no further action. However, if a controller observes a prolonged discrepancy between submitted and agreed commands, the controller suspects the current proposer to be faulty and reacts by announcing a higher view number.

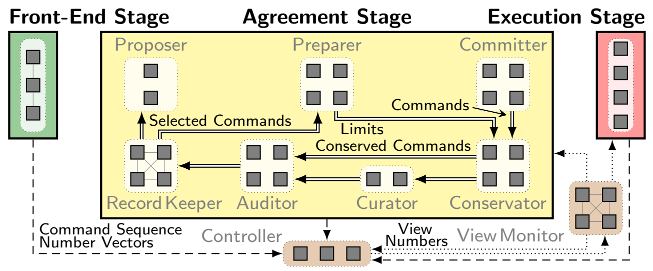


Figure 7. MIRADOR clusters involved in view change

For efficiency, controllers do not gather full commands, but instead base their decisions on command sequence numbers. More precisely, a controller repeatedly queries front-end replicas for a vector $\vec{t}_{expected}$ containing the highest known command sequence number of each client. In parallel, it also obtains a corresponding vector $\vec{t}_{actual}$ from executors, which indicates the progress that the agreement actually made. If a command advertised in $\vec{t}_{expected}$ is not reflected in $\vec{t}_{actual}$ within a timeout T_p, a controller advocates for a higher view.

To announce the new view, MIRADOR uses a third control loop that matches the structure of the other control loops presented in Section V-B. Here, a cluster of *view monitors* constantly watches the controllers and distributes the current view number to all replicas in the agreement and execution stage as well as the controller cluster. If a replica this way learns of a higher view, it immediately ceases participation in its old view. For proposers and preparers, this for example means to temporarily suspend the agreement of new commands.

Picking Up the Pieces. Whenever the view control loop announces a new view, a cluster of *conservators* starts to collect knowledge about the state that the old view has left the agreement stage in. Specifically, conservators accumulate two pieces of information: (1) Using input from the preparer cluster, they determine an upper bound for the agreement sequence numbers that potentially have been decided. (2) Conservators query committers for the commands they have previously accepted (cf. prepared requests in PBFT [11]), as these commands may have been processed by a correct executor. Both preparers and committers only respond to such conservator queries if they currently are in the view that a conservator asks for. This ensures that they do no longer contribute to previous views and also have not already participated in higher views.

Prior to obtaining potentially committed commands, conservators first need to learn the range of agreement slots that are relevant for the view change. The lower bound of this range is defined by the garbage-collection threshold of the agreement loop (see Section V-B), to which all view-change clusters are connected. For the upper bound, conservators ask preparers for a tuple $\langle \psi, \omega \rangle$ in which ψ denotes the highest agreement sequence number for which a preparer has previously accepted a command in any lower view, and ω represents the upper end of the preparer's output window. Based on the responses of at least $2f+1$ preparers, a conservator selects the upper bound Ψ to be the $2f+1$ lowest reported ψ and accepts this bound once Ψ does not exceed the $f+1$ highest reported ω. Since $2f+1$ matching preparer outputs are necessary for a committer to accept a command, slots higher than Ψ cannot have been committed in any lower view and thus do not have to be considered. Bounding Ψ with the $f+1$ highest reported ω ensures that faulty preparers cannot trick correct conservators into accepting arbitrarily high sequence numbers as upper bounds.

Having determined the relevant range, conservators then query replicas of the committer cluster for the associated commands. For each agreement sequence number, a committer x reports a tuple $z_x = \langle m, v \rangle_x$ containing the command m and the number of the latest view v in which the command was accepted; empty slots are encoded with a dummy value $\clubsuit$. Having verified the validity of the included commands, conservators for each agreement slot s combine the obtained tuples into a vector $\vec{z_s}$ representing a summary of the committer replicas' opinions on the agreement sequence number.

Entering the New View. In a next step, the conserved tuples are handled by a series of *curator*, *auditor*, and *record keeper* clusters that together implement a three-step BFT consensus responsible for determining the command-to-agreement-sequence-number mapping with which to start the new view. For this purpose, the current curator (i.e., a micro replica acting as view-change leader) queries the conservator cluster for $\vec{z}$ vectors until for each relevant agreement sequence number it obtains a set $\mathcal{Z}$ of $2f+1$ tuples z_i (from different committers) in which each z_i has been reported by at least $2f+1$ conservators. This ensures that, after downloading $\mathcal{Z}$ from the curator, a correct auditor will be able to find at least $f+1$ correct conservator replicas confirming the correctness of each z_i in $\mathcal{Z}$. For each agreement sequence number, the view-change result (i.e., the value to be selected for the new view) is the command of the tuple with the highest view v in $\mathcal{Z}$.

Record keepers query auditors and each other for view-change results and accept a command once it is backed up by at least $2f+1$ auditors or $f+1$ record keepers. Once these commands are obtained by preparers and the new proposer, the agreement stage resumes normal-case operation.

Coordinating the View Change. MIRADOR cleanly distinguishes between (1) assigning a new command to an agreement sequence number during normal-case operation and (2) selecting the value for an agreement sequence number during view change. Following the micro-replication principles, these tasks consequently are separated into different clusters (i.e., proposer and curator). Since both tasks require a dedicated leader replica, there needs to be a possibility to reassign the leader roles of proposers and curators independently. MIRADOR solves this problem by defining the view number $v = e_p || e_c$ as a concatenation of a proposer epoch e_p and a curator epoch e_c, and enabling controllers to increase either in case of suspected faulty behavior. In order to achieve this in a coordinated fashion, controllers switch between different modes. During normal-case operation, they monitor the progress of front ends and executors (as discussed above) and if necessary abandon the current proposer by setting their local view number to $v' = (e_p+1)||0$. In a similar way, if conserved commands do not arrive at the record-keeper cluster in a timely manner, controllers can replace a curator by announcing a view $v'' = e_p||(e_c+1)$. Computing the next view number this way ensures that the sequence of view numbers is strictly monotonically increasing and therefore establishes a total order on the commands reported by committers for lower views, as it is the case in traditional protocols [11], [40].

VI. EVALUATION

This section presents a case study that evaluates our Java-based MIRADOR prototype in comparison with the widely used replication library BFT-SMaRt [16]. In addition to the parts detailed below, for the case study we also used TLA+ [41] to model check selected parts of MIRADOR, verifying certain elementary behavior for both the preparer and the committer. Furthermore, we designed an external dashboard that collects and presents the current state of micro replicas. Using the same messages and interfaces as regular micro replicas do to retrieve protocol information from their upstream neighbors, the dashboard only queries the micro replicas that are currently of interest. Thus, when the dashboard is inactive, the system is able to run at full native speed.

A. Debuggability

As first part of our case study, we want to answer the question whether micro replication indeed offers benefits when it comes to debugging replication-protocol implementations. To obtain meaningful results, we analyze a set of 14 real-world bugs that were recently reported for BFT-SMaRt, and most of which have been fixed in the meantime [42]. As summarized in Figure 8, we conclude that two of the bugs (#10 and #13) in themselves are already straightforward to diagnose due to resulting in exceptions that directly point to the root cause. With regard to the remaining 12 bugs, micro replication provides advantages by either avoiding them in the first place or simplifying the search for them, as discussed in the following.

Bug Avoidance. Interestingly, our study shows that the majority of the bugs discovered in BFT-SMaRt cannot occur in a micro-replicated implementation, a fact that could be viewed as ideal form of debuggability since it obviates the need for any debugging procedures. 5 of the bugs (labeled MRA in Figure 8) are ruled out by the micro-replica architecture, which

#	Bug description	Type
1	Faulty clients may overload replicas by sending huge requests	MRA
2	Faulty clients may trigger a view change via forwarded messages	CSR
3	Replicas erroneously accept negative request sequence numbers	MRA
4	Timers of pending requests not discarded at client-session restart	CSR
5	Replicas possibly ignore new requests after client-session restart	ETL
6	Faulty clients may exploit synchronization bug to cause deadlock	MRA
7	Timing issue at request reception may lead to unnecessary view change	CSR
8	Deadlock if different consensus steps are processed in a particular order	CSR
9	Race condition when waiting for another thread to deserialize message	MRA
10	Faulty clients may trigger an exception by sending unsigned requests	EQU
11	Synchronization bug in delivery thread may result in deadlock	MRA
12	Faulty clients may prevent replicas from responding to correct clients	CSR
13	Faulty replicas may cause a correct replica's receiver thread to crash	EQU
14	Out of memory error due to buffering messages from higher views	ETL

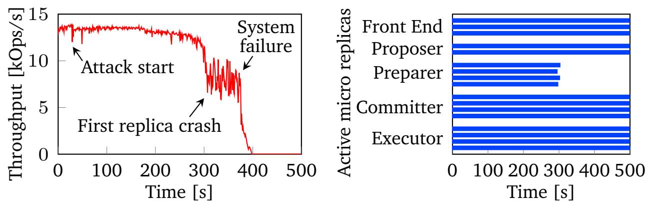
Figure 8. Categorization of recent bugs in BFT-SMaRt [42] with regard to their debuggability in the context of micro replication: equally difficult to debug (EQU), cannot occur due to micro-replica architecture (MRA) or clean separation of responsibilities into modules (CSR), easier to locate (ETL).

stores inputs and outputs in preallocated data structures (#1 and #3) and executes each micro replica in a single thread (#6, #9, #11). An additional 5 bugs (labeled CSR) are avoided by the micro-replication principles demanding a clean separation of responsibilities. As our analysis shows, this aspect is crucial due to a significant number of problems in BFT-SMaRt being caused by interference of different protocol mechanisms, with the reception of client requests affecting the initiation of view changes (#2, #4, #7), a monolithic replica handling multiple consensus steps (#8), and incoming faulty requests possibly disrupting the distribution of replies to clients (#12). In contrast, in MIRADOR the mentioned protocol parts are delegated to separate clusters, with front ends receiving commands, controllers triggering view changes, and executors sending out results, thereby eliminating the potential for interference.

Bug Search. The remaining two bugs are not per se prevented by our approach, however micro replication makes it significantly easier to locate them. For Bug #5, whose observable effect is that a restarted client does not receive any results to its requests, a debugging tool such as our dashboard would show that new commands do not reach later protocol stages, thereby correctly indicating the front-end cluster to be the culprit.

Bug #14, one of the most critical bugs in the analyzed set, represents a memory leak that can be exploited by a malicious leader to deliberately crash any correct replica. It is caused by the fact that when receiving agreement messages for higher views, BFT-SMaRt replicas buffer these messages for later use. Since there is no upper limit on the capacity of this buffer, a faulty leader for example may force other replicas out of memory by sending large proposals for views with arbitrarily high numbers. As shown in Figure 9a, we were able to reproduce such a scenario in an experiment in which the attack starts at 30 s and the overall system fails at about 400 s.

Unfortunately, debugging this kind of issues is inherently difficult because the observable symptoms not necessarily point in the direction of the cause. Specifically, even though the bug is associated with the reception of proposals, we saw out-of-memory errors in various locations, including code parts that interact with clients. Not surprisingly, when running out of memory, the accompanying exception in traditional monolithic replicas may be thrown in any protocol part that tries to allocate memory. In contrast, micro replication makes it possible to reduce the blast radius of memory leaks to small areas (e.g., by hosting each micro replica in a separate virtual machine). This way, when we introduce the same bug into MIRADOR, it always results in preparer failures (see Figure 9b), thereby making it easier to identify the source of this bug.



(a) Replica crashes in BFT-SMaRt (b) Preparer crashes in MIRADOR

Figure 9. Impact of a memory leak in both systems

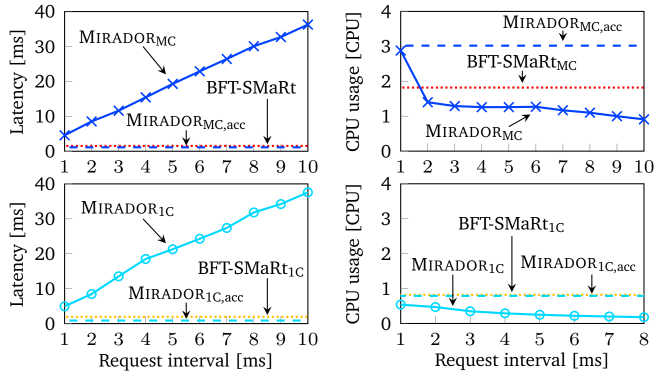


Figure 10. Request-interval impact on latency and CPU usage

B. Performance

For the next part of our study, we conduct a performance evaluation comparing the following four settings: (1) **MIRADOR$_{MC}$** refers to our prototype when configured to run each micro replica in its own Java virtual machine (JVM). (2) **BFT-SMaRt$_{MC}$** relies on the library's default BFT configuration, which applies the BFT protocol proposed by Cachin in [43] for agreement [6]. (3) **MIRADOR$_{1C}$** is a variant of MIRADOR in which for resource efficiency micro replicas of different clusters are combined in the same JVM and executed on a single core. (4) **BFT-SMaRt$_{1C}$** refers to a setting where we apply the Linux tool `taskset` to limit BFT-SMaRt to a single core per server, thereby providing a baseline for MIRADOR$_{1C}$. For our experiments we use a set of five servers (8 cores, 3.60 GHz, 16 GB RAM, Ubuntu 20.04.4) of which four are hosting replicas and one is executing the participating clients.

Protocol Latency. In our first experiment, we analyze the impact of the interval with which a MIRADOR replica queries its upstream counterparts for inputs. Since our focus is on end-to-end protocol latency, for this purpose we use a microbenchmark in which 30 clients exchange empty commands and results with the replicated service. As clients run in a closed loop, the per-client throughput at latency L is $1/L$.

The results in Figure 10 show that for a request interval of 1 ms, both MIRADOR$_{MC}$ and MIRADOR$_{1C}$ provide response times of about 5 ms. With the protocol's normal-case path consisting of 6 steps (see Figure 5), this observation indicates that outputs do not always have to wait an entire request interval to be collected by downstream replicas. For larger intervals, the latency experienced by clients increases linearly with the duration between two queries. Although the two protocol implementations in this scenario advance at about the same speed, MIRADOR$_{MC}$ due to the use of isolated JVMs does this at a higher (but still reasonable) resource consumption, which represents the costs for enabling improved debuggability.

Responding on average after less than 2 ms, the two BFT-SMaRt variants in the experiment offer lower latency than MIRADOR$_{MC}$ and MIRADOR$_{1C}$. Apart from BFT-SMaRt's replication protocol requiring only 5 phases of message exchange, this is caused by the fact that replicas in BFT-SMaRt immediately push their outputs at the end of each protocol step. As discussed in Section IV-B, although the pull-based

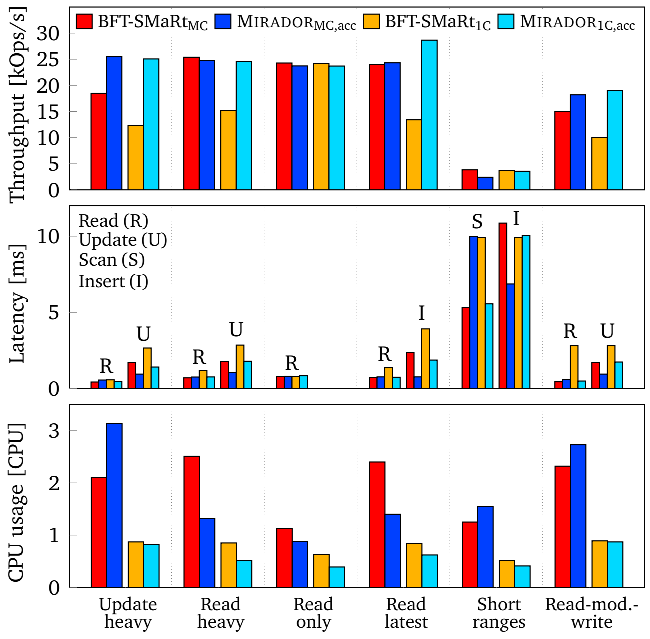


Figure 11. Measurement results for the YCSB benchmarks

communication is an essential part of micro replication, its principles do not prevent micro replicas from using additional forms of interaction as optimization. To illustrate how this flexibility may be exploited in practice, we implement two further variants MIRADOR$_{MC,acc}$ and MIRADOR$_{1C,acc}$ using an accelerator-based mechanism (see Section IV-B) in which replicas push their outputs to a preferred quorum [12] of downstream replicas. As the results in Figure 10 show, without impeding debuggability, these measures enable MIRADOR$_{MC,acc}$ and MIRADOR$_{1C,acc}$ to provide latencies as low as BFT-SMaRt.

YCSB. Next, we rely on YCSB [44] to evaluate micro replication with realistic application workloads. At the server side, YCSB provides a database of 1 KB records which each contain multiple fields of 100 B values. The six benchmarks test a variety of workloads using different combinations of inserts, reads, updates, and scans. To reduce read and scan latency, both BFT-SMaRt and MIRADOR leverage PBFT's read-only optimization [11]. In MIRADOR this means that clients directly submit their reads to executors, thereby bypassing 4 of the 6 regular-path steps, and for all operations accept a result after obtaining $2f + 1$ matching replies. Figure 11 shows that the read-only optimization is also effective in MIRADOR. Overall, the measurements confirm that when using accelerators a micro-replicated protocol is able to offer throughputs and latencies that match the performance of traditional protocols.

Unreliable Network. Our final experiment examines the impact of unstable network behavior with a phase in which 10% of protocol messages are randomly dropped. As shown in Figure 12, once the issues start, BFT-SMaRt enters a series of repeated view changes due to followers missing some of the (old and new) leaders' proposals. In contrast, thanks to micro replicas periodically submitting pull-based queries, MIRADOR is able to continuously make progress without view change.

VII. Discussion

In the following, we discuss implications, advantages, and limitations of our design choices made for micro replication.

Modularization. Micro replication breaks a protocol down into its atoms, thereby introducing development and maintenance overhead compared with traditional designs. Taking into account that some of today's production infrastructures comprise hundreds or even thousands of loosely coupled microservices [45], [46], we think it is not unreasonable to believe that the additional costs are manageable in practice. One of Twitter's main data-processing frameworks, for example, was specifically built to place nodes in separate JVMs for better debuggability [47]. Besides, as shown by MIRADOR$_{1C}$, at the expense of weaker replica isolation the deployment overhead can be reduced by combining clusters into a single process.

While fined-grained modularization decisively reduces the blast radius for certain bugs (see Bug #14 in Section VI-A), there are issues for which micro replication does not offer additional benefits over traditional approaches. Specifically, this is true for bugs that take effect multiple stages away from their source, for example if during view change the leader propagates a command with the wrong view number [48] or a correct replica loses state during a restart [49]. In both cases, the impact may only become visible after the agreement stage. Furthermore, micro replication cannot serve as protection against protocol designs that violate fundamental principles, for example by committing a command in too few steps [50].

Pull-based Communication. Compared with traditional push-based approaches, our design decision to use pull-based micro-replica interaction introduces additional costs in terms of latency (i.e., two communication steps instead of one) and messages (i.e., request + reply instead of a single message). However, as confirmed by our evaluation results, the impact on performance can be effectively mitigated by adding accelerators. As discussed in Section IV-B, accelerators are a best-effort mechanism and thus may only serve as addition to pull-based queries, not as a replacement. Nevertheless, during periods of benign network conditions accelerators free replicas from the need to send queries at high frequency, thereby minimizing the runtime overhead of pull-based communication.

Organizing replica interaction in a pull-based manner offers the key benefit that external debuggers can use the same queries as regular downstream replicas to inspect an upstream replica's current state (see Section IV-A). Notice that this does not guarantee that a debugger will always be able to observe a replica's complete history. Specifically, it can miss outputs that have been created and already garbage-collected between two debugger queries. Our experience and the analysis of

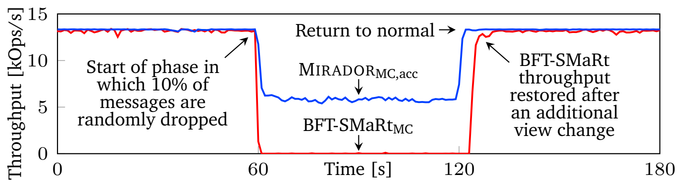


Figure 12. Impact of an unreliable network

BFT-SMaRt in Section VI-A show that this limitation usually does not pose a major problem in practice as many bugs in protocol implementations (including the MIRADOR prototype) result in the entire system getting stuck. In such cases, knowledge about the current replica state is key and there is no functioning garbage collection to discard the relevant evidence. Nevertheless, since this of course does not cover all debugging scenarios, as discussed in Section III, we regard pull-based debugging only as additional option to the use of log files.

Single-Threaded Micro-Replica Architecture. Besides avoiding race conditions (see Bugs #6, #9, #11 in Section VI-A), relying on single-threaded execution within micro replicas can also offer benefits with regard to performance. With most steps in a replication protocol only involving a few comparably inexpensive actions (e.g., checking message fields, updating in-memory data structures), as pointed out by Kończak et al. [17] there are usually no advantages in implementing the protocol logic as a multi-threaded component. Even worse, distributing tasks across multiple threads using staged event-driven designs [51] typically results in significant synchronization overhead that prevents a replica from fully exploiting the available CPU and network resources [4], [52]–[54].

Notice that the absence of parallelism at the replica level does not rule out the use of parallelism in other parts of the replicated system. For example, micro replication does not prohibit multi-threading at the application level (i.e., after an executor delivered a command to its local service-application instance), meaning that it can be integrated with approaches that exploit concurrency during command execution [55], [56]. Furthermore, to prevent single-threaded replicas from becoming a bottleneck (e.g., due to high message-authentication overhead), a micro-replicated protocol may distribute the agreement load across multiple partitions [53], [57], [58] by instantiating the agreement-stage clusters (i.e., proposers, preparers, committers etc.) several times and assigning each partition a portion of commands. This creates parallelism at the protocol level (which can also be used to improve resilience [59]) without violating the single-threaded replicas principle.

Generalizability. We believe that many of the designs and mechanisms developed for CFT and BFT protocols [30], [60] can either be directly incorporated or adapted to fit the micro-replication principles. For example, exploiting the close structural similarity between Kirsch et al.'s Paxos variant [28] and PBFT, deriving a CFT version of MIRADOR is straightforward. Furthermore, the use of trusted subsystems may allow micro-replicated protocols to save resources [29], [61]–[67]. However, some concepts such as Raft's strong leader [15] are not as easily expressible using micro replication. In general, this applies to all approaches that exploit the fact that the same monolithic replica can participate in multiple subsequent protocol steps. Although a source for bugs (see Bug #8 in Section VI-A), if done correctly, this property can foster efficiency. PBFT, for example, ensures that a new leader presides over the view change preceding its own view and therefore can leverage that a correct leader will properly handle both tasks; in contrast, in MIRADOR curators and proposers may fail independently (see Section V-C). Notice that this does not rule out designs in which individual replicas are assigned special roles (e.g., to achieve linear communication complexity [7], [40] or to coordinate leaderless consensus [68]–[70]), as long as this difference in characteristics is taken into account.

VIII. RELATED WORK

Since debugging distributed and multi-threaded systems is notoriously difficult, a myriad of approaches and frameworks exist that have been designed to provide assistance in this process. This includes record and replay support for individual virtual machines/processes [25], [71], [72] or even entire distributed systems [20], [26], [73], [74], hardware-assisted replay [75]–[77], parallelized debuggers [20], [78], visualization of protocol execution [79], automated tracing [80]–[82], automated damage assessment [22], and effective log analysis [83]–[85]. When used for replicated systems, such tools can benefit from protocols being designed based on the micro-replication principles. First and foremost, the fact that all protocol-phase transitions are externalized and involve (pull-based) network communication allows tracers to gain easy access to up-to-date state information. Also, the standardized architecture and interfaces of micro replicas greatly simplify the provision of tooling support for the entire system. Finally, the reduced blast radius of failures enables debuggers to concentrate on specific areas, thereby speeding up the bug search.

We are not the first to leverage modularization in replication protocols. For Paxos, Lamport [33] introduced the roles of proposers, acceptors, and learners to distinguish essential algorithm steps. Whittaker et al. [86] extended this idea in a decoupled architecture focused on eliminating scalability bottlenecks. Boichat et al. [87] deconstructed Paxos into modules for leader election and consensus, thereby providing the basis for new protocol variants. For BFT, Aublin et al. [88] presented an approach to build complex protocols as a composition of multiple tailored instances of the same abstraction. Yin et al. [8] proposed an architecture that separates agreement from execution, to which Clement et al. [12] later added a client-handling stage. Recently, Messadi et al. [89] split a protocol into three compartments (i.e., preparation, confirmation, execution). Compared with these approaches, micro replication differs in two important aspects: (1) It aims for a much finer granularity by isolating each protocol step in a separate module. (2) It not only focuses on specific tasks, but instead targets all mechanisms that are essential in practice. This includes protocol parts such as flow control and view change, which are often the ones most difficult to debug.

IX. CONCLUSION

Micro replication enables the design of replication protocols that simplify debugging-related tasks such as bug-source isolation, state-information retrieval, and root-cause identification. Our evaluation shows that the improvement in debuggability can be achieved without impeding performance.

Acknowledgments: Partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 446811880, 391790956.

REFERENCES

[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[2] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, 2006, pp. 335–350.

[3] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems," in *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC '10)*, 2010, pp. 145–158.

[4] R. Schiekofer, J. Behl, and T. Distler, "Agora: A dependable high-performance coordination service for multi-cores," in *Proceedings of the 47th International Conference on Dependable Systems and Networks (DSN '17)*, 2017, pp. 333–344.

[5] M. Vukolić, "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)*, 2015, pp. 112–125.

[6] J. Sousa, A. Bessani, and M. Vukolić, "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)*, 2018, pp. 51–58.

[7] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, "SBFT: A scalable and decentralized trust infrastructure," in *Proceedings of the 49th International Conference on Dependable Systems and Networks (DSN '19)*, 2019, pp. 568–580.

[8] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for Byzantine fault tolerant services," in *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP '03)*, 2003, pp. 253–267.

[9] A. N. Bessani, P. Sousa, M. Correia, N. F. Neves, and P. Veríssimo, "The CRUTIAL way of critical infrastructure protection," *IEEE Security & Privacy*, vol. 6, no. 6, pp. 44–51, 2008.

[10] M. Garcia, N. Neves, and A. Bessani, "SieveQ: A layered BFT protection system for critical services," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 3, pp. 511–525, 2016.

[11] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, 1999, pp. 173–186.

[12] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "UpRight cluster services," in *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP '09)*, 2009, pp. 277–290.

[13] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, and Z. Zhang, "D^3S: Debugging deployed distributed systems," in *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI '08)*, 2008.

[14] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[15] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*, 2014, pp. 305–320.

[16] A. Bessani, J. Sousa, and E. E. P. Alchieri, "State machine replication for the masses with BFT-SMaRt," in *Proceedings of the 44th International Conference on Dependable Systems and Networks (DSN '14)*, 2014, pp. 355–362.

[17] J. Kończak, N. F. de Sousa Santos, T. Żurkowski, P. T. Wojciechowski, and A. Schiper, "JPaxos: State machine replication based on the Paxos protocol," EPFL, Tech. Rep. 167765, 2011.

[18] M. Brooker, T. Chen, and F. Ping, "Millions of tiny databases," in *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI '20)*, 2020, pp. 463–478.

[19] D. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay debugging for distributed applications," in *Proceedings of the 2006 USENIX Annual Technical Conference (USENIX ATC '06)*, 2006.

[20] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, "Friday: Global comprehension for distributed replay," in *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI '07)*, 2007, pp. 285–298.

[21] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, "BFT protocols under fire," in *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI '08)*, 2008, pp. 189–204.

[22] R. Banabic, G. Candea, and R. Guerraoui, "Automated vulnerability discovery in distributed systems," in *Proceedings of the 41st International Conference on Dependable Systems and Networks Workshops (DSN-W '11)*, 2011, pp. 188–193.

[23] H. Lee, J. Seibert, E. Hoque, C. Killian, and C. Nita-Rotaru, "Turret: A platform for automated attack finding in unmodified distributed system implementations," in *Proceedings of the 34th International Conference on Distributed Computing Systems (ICDCS '14)*, 2014, pp. 660–669.

[24] D. Gupta, L. Perronne, and S. Bouchenak, "BFT-Bench: Towards a practical evaluation of robustness and effectiveness of BFT protocols," in *Proceedings of the 16th International Conference on Distributed Applications and Interoperable Systems (DAIS '16)*, 2016, pp. 115–128.

[25] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability," in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*, 2017, pp. 337–389.

[26] D. Aumayr, S. Marr, C. Béra, E. G. Boix, and H. Mössenböck, "Efficient and deterministic record & replay for actor languages," in *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang '18)*, 2018.

[27] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, "Twins: BFT systems made robust," in *Proceedings of the 25th International Conference On Principles Of Distributed Systems (OPODIS '21)*, 2021, pp. 7:1–7:29.

[28] J. Kirsch and Y. Amir, "Paxos for system builders: An overview," in *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS '08)*, 2008, pp. 14–18.

[29] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "CheapBFT: Resource-efficient Byzantine fault tolerance," in *Proceedings of the 7th European Conference on Computer Systems (EuroSys '12)*, 2012, pp. 295–308.

[30] T. Distler, "Byzantine fault-tolerant state-machine replication from a systems perspective," *ACM Computing Surveys*, vol. 54, no. 1, pp. 24:1–24:38, 2021.

[31] T. Erl, *Service-Oriented Architecture*. Pearson India, 2005, vol. 8.

[32] S. Newman, *Building Microservices: Designing Fine-grained Systems*. O'Reilly Media, 2015.

[33] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column)*, vol. 32, pp. 51–58, 2001.

[34] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Prime: Byzantine replication under attack," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 4, pp. 564–577, 2011.

[35] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine fault tolerance," *ACM Transactions on Computer Systems*, vol. 27, no. 4, pp. 7:1–7:39, 2009.

[36] A. Nogueira, M. Garcia, A. Bessani, and N. Neves, "On the challenges of building a BFT SCADA," in *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN '18)*, 2018, pp. 163–170.

[37] A. Babay, T. Tantillo, T. Aron, M. Platania, and Y. Amir, "Network-attack-resilient intrusion-tolerant SCADA for the power grid," in *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN '18)*, 2018, pp. 255–266.

[38] A. Babay, J. Schultz, T. Tantillo, S. Beckley, E. Jordan, K. Ruddell, K. Jordan, and Y. Amir, "Deploying intrusion-tolerant SCADA for the power grid," in *Proceedings of the 49th International Conference on Dependable Systems and Networks (DSN '19)*, 2019, pp. 328–335.

[39] MIRADOR, "Specification and implementation," https://gitlab.cs.fau.de/i4/micro-replication/mirador, 2023.

[40] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: BFT consensus with linearity and responsiveness," in *Proceedings of the 38th Symposium on Principles of Distributed Computing (PODC '19)*, 2019, pp. 347–356.

[41] L. Lamport, "The TLA+ home page," https://lamport.azurewebsites.net/tla/tla.html, 2022.

[42] BFT-SMaRt GitHub Repository, Commits on April 15, 2022, `17d0bcd6ef1f3d288f6f6b614c97d22bf718eecc` to `ae06f9292509d00a2387f9dfff16be203a37be40`, https://github.com/bft-smart/library/commits/master.

[43] C. Cachin, "Yet another visit to Paxos," IBM Research Zurich, Tech. Rep. RZ3754, 2009.

[44] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st Symposium on Cloud Computing (SoCC '10)*, 2010, pp. 143–154.

[45] Y. Gan and C. Delimitrou, “The architectural implications of cloud microservices,” *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 155–158, 2018.

[46] I. Papapanagiotou and V. Chella, “NDBench: Benchmarking microservices at scale,” *CoRR*, vol. abs/1807.10792, 2018.

[47] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellog g, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, “Twitter Heron: Stream processing at scale,” in *Proceedings of the 41st International Conference on Management of Data (SIGMOD ’15)*, 2015, pp. 239–250.

[48] X. Liu, W. Lin, A. Pan, and Z. Zhang, “WiDS checker: Combating bugs in distributed systems,” in *Proceedings of 4th Symposium on Networked Systems Design and Implementation (NSDI ’07)*, 2007.

[49] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak, “CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems,” in *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI ’09)*, 2009.

[50] N. Shrestha and M. Kumar, “Revisiting ezBFT: A decentralized Byzantine fault tolerant protocol with speculation,” *CoRR*, vol. abs/ 1909.03990, 2019.

[51] M. Welsh, D. Culler, and E. Brewer, “SEDA: An architecture for well-conditioned, scalable Internet services,” in *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP ’01)*, 2001, pp. 230–243.

[52] N. Santos and A. Schiper, “Achieving high-throughput state machine replication in multi-core systems,” in *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS ’13)*, 2013, pp. 266–275.

[53] J. Behl, T. Distler, and R. Kapitza, “Consensus-oriented parallelization: How to earn your first million,” in *Proceedings of the 16th Middleware Conference (Middleware ’15)*, 2015, pp. 173–184.

[54] C. Deyerl and T. Distler, “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)*, 2019, pp. 1–7.

[55] R. Kotla and M. Dahlin, “High throughput Byzantine fault tolerance,” in *Proceedings of the 34th International Conference on Dependable Systems and Networks (DSN ’04)*, 2004, pp. 575–584.

[56] M. Kapritsos, Y. Wang, V. Quéma, A. Clement, L. Alvisi, and M. Dahlin, “All about Eve: Execute-verify replication for multi-core servers,” in *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI ’12)*, 2012, pp. 237–250.

[57] P. J. Marandi, M. Primi, and F. Pedone, “Multi-ring Paxos,” in *Proceedings of the 42nd International Conference on Dependable Systems and Networks (DSN ’12)*, 2012, pp. 1–12.

[58] M. Eischer and T. Distler, “Scalable Byzantine fault-tolerant state-machine replication on heterogeneous servers,” *Computing*, vol. 101, no. 2, pp. 97–118, 2019.

[59] P.-L. Aublin, S. B. Mokhtar, and V. Quéma, “RBFT: Redundant Byzantine fault tolerance,” in *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS ’13)*, 2013, pp. 297–306.

[60] Z. Wang, C. Zhao, S. Mu, H. Chen, and J. Li, “On the parallels between Paxos and Raft, and how to port optimizations,” in *Proceedings of the 38th Symposium on Principles of Distributed Computing (PODC ’19)*, 2019, pp. 445–454.

[61] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, “Attested append-only memory: Making adversaries stick to their word,” in *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP ’07)*, 2007, pp. 189–204.

[62] H. P. Reiser and R. Kapitza, “Hypervisor-based efficient proactive recovery,” in *Proceedings of the 26th Symposium on Reliable Distributed Systems (SRDS ’07)*, 2007, pp. 83–92.

[63] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, “EBAWA: Efficient Byzantine agreement for wide-area networks,” in *Proceedings of the 12th Symposium on High-Assurance Systems Engineering (HASE ’10)*, 2010, pp. 10–19.

[64] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo, “Efficient Byzantine fault-tolerance,” *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2013.

[65] J. Behl, T. Distler, and R. Kapitza, “Hybrids on steroids: SGX-based high performance BFT,” in *Proceedings of the 12th European Conference on Computer Systems (EuroSys ’17)*, 2017, pp. 222–237.

[66] B. Li, N. Weichbrodt, J. Behl, P.-L. Aublin, T. Distler, and R. Kapitza, “Troxy: Transparent access to Byzantine fault-tolerant systems,” in *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN ’18)*, 2018, pp. 59–70.

[67] S. Gupta, S. Rahnama, S. Pandey, N. Crooks, and M. Sadoghi, “Dissecting BFT consensus: In trusted components we trust!” *CoRR*, vol. abs/2202.01354, 2022.

[68] I. Moraru, D. G. Andersen, and M. Kaminsky, “There is more consensus in egalitarian parliaments,” in *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP ’13)*, 2013, pp. 358–372.

[69] T. Crain, V. Gramoli, M. Larrea, and M. Raynal, “DBFT: Efficient leaderless Byzantine consensus and its application to blockchains,” in *Proceedings of the 17th International Symposium on Network Computing and Applications (NCA ’18)*, 2018, pp. 1–8.

[70] M. Eischer and T. Distler, “Egalitarian Byzantine fault tolerance,” in *Proceedings of the 26th Pacific Rim International Symposium on Dependable Computing (PRDC ’21)*, 2021, pp. 77–86.

[71] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, “Execution replay of multiprocessor virtual machines,” in *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE ’08)*, 2008, p. 121–130.

[72] A. Burtsev, D. Johnson, M. Hibler, E. Eide, and J. Regehr, “Abstractions for practical virtual machine replay,” in *Proceedings of the 12th International Conference on Virtual Execution Environments (VEE ’16)*, 2016, p. 93–106.

[73] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang, “Rex: Replication at the speed of multi-core,” in *Proceedings of the 9th European Conference on Computer Systems (EuroSys ’14)*, 2014.

[74] W. Wang, Z. Hao, and L. Cui, “ClusterRR: A record and replay framework for virtual machine cluster,” in *Proceedings of the 18th International Conference on Virtual Execution Environments (VEE ’22)*, 2022, p. 31–44.

[75] A. J. Mashtizadeh, T. Garfinkel, D. Terei, D. Mazieres, and M. Rosenblum, “Towards practical default-on multi-core record/replay,” in *Proceedings of the 22th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’17)*, 2017, p. 693–708.

[76] H. Liu, S. Silvestro, W. Wang, C. Tian, and T. Liu, “Ireplayer: Insitu and identical record-and-replay for multithreaded applications,” in *Proceedings of the 39th Conference on Programming Language Design and Implementation (PLDI ’18)*, 2018, p. 344–358.

[77] G. Zuo, J. Ma, A. Quinn, P. Bhatotia, P. Fonseca, and B. Kasikci, “Execution reconstruction: Harnessing failure reoccurrences for failure reproduction,” in *Proceedings of the 42nd Conference on Programming Language Design and Implementation (PLDI ’21)*, 2021, p. 1155–1170.

[78] A. Quinn, J. Flinn, and M. Cafarella, “Sledgehammer: Cluster-fueled debugging,” in *Proceedings of the 13th Conference on Operating Systems Design and Implementation (OSDI ’18)*, 2018, p. 545–560.

[79] N. Rakotondravony and H. P. Reiser, “Visualizing BFT SMR distributed systems - example of BFT-SMaRt,” in *Proceedings of the 48th International Conference on Dependable Systems and Networks Workshops (DSN-W ’18)*, 2018, pp. 152–157.

[80] A. Chanda, K. Elmeleegy, A. L. Cox, and W. Zwaenepoel, “Causeway: Operating system support for controlling and analyzing the execution of distributed programs,” in *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS ’05)*, 2005, p. 18.

[81] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, “Be conservative: Enhancing failure diagnosis with proactive logging,” in *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI ’12)*, 2012, p. 293–306.

[82] J. Mace, R. Roelke, and R. Fonseca, “Pivot tracing: Dynamic causal monitoring for distributed systems,” in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP ’15)*, 2015, p. 378–393.

[83] A. Burtsev, N. Mishrikoti, E. Eide, and R. Ricci, “Weir: A streaming language for performance analysis,” *ACM SIGOPS Operating Systems Review*, vol. 48, no. 1, p. 65–70, 2014.

[84] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm, “Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle,” in *Proceedings of the 12th Conference on Operating Systems Design and Implementation (OSDI ’16)*, 2016, pp. 603–618.

[85] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswa nath, L. Jiao, and C. Fetzer, “Sieve: Actionable insights from monitored metrics in distributed systems,” in *Proceedings of the 18th Middleware Conference (Middleware ’17)*, 2017, p. 14–27.

[86] M. Whittaker, A. Ailijiang, A. Charapko, M. Demirbas, N. Giridharan, J. M. Hellerstein, H. Howard, I. Stoica, and A. Szekeres, “Scaling

replicated state machines with compartmentalization," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2203–2215, 2021.

[87] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui, "Deconstructing Paxos," *ACM Sigact News*, vol. 34, no. 1, pp. 47–67, 2003.

[88] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 BFT protocols," *ACM Transactions on Computer Systems*, vol. 32, no. 4, pp. 12:1–12:45, 2015.

[89] I. Messadi, M. H. Becker, K. Bleeke, L. Jehl, S. B. Mokhtar, and R. Kapitza, "SplitBFT: Improving Byzantine fault tolerance safety using trusted compartments," in *Proceedings of the 23rd Middleware Conference (Middleware '22)*, 2022, pp. 56–68.