Resource-efficient Byzantine Fault Tolerance

Tobias Distler, Christian Cachin, and Rüdiger Kapitza

Abstract—One of the main reasons why Byzantine fault-tolerant (BFT) systems are currently not widely used lies in their high resource consumption: $3f + 1$ replicas are required to tolerate only f faults. Recent works have been able to reduce the minimum number of replicas to $2f + 1$ by relying on trusted subsystems that prevent a faulty replica from making conflicting statements to other replicas without being detected. Nevertheless, having been designed with the focus on fault handling, during normal-case operation these systems still use more resources than actually necessary to make progress in the absence of faults.

This paper presents *Resource-efficient Byzantine Fault Tolerance* (REBFT), an approach that minimizes the resource usage of a BFT system during normal-case operation by keeping f replicas in a passive mode. In contrast to active replicas, passive replicas neither participate in the agreement protocol nor execute client requests; instead, they are brought up to speed by verified state updates provided by active replicas. In case of suspected or detected faults, passive replicas are activated in a consistent manner. To underline the flexibility of our approach, we apply REBFT to two existing BFT systems: PBFT and MinBFT.

Index Terms—Byzantine fault tolerance, state machine replication, distributed systems.

1 Introduction

OUTSOURCING infrastructure and applications to third-party data centers is convenient for both users and providers as availability increases while provisioning costs decrease. On the other hand, this trend makes data-center customers more and more dependent on the well-functioning of such network-based services, which becomes evident when they fail or deliver wrong results.

Today, the fault tolerance techniques applied in practice are almost solely dedicated to handling crash-stop failures, for example, by employing replication. Apart from that, only specific techniques are used to selectively address the most common or most severe non-crash faults, for example, by using checksums to detect bit flips. As a consequence, a wide spectrum of threats remains largely unaddressed, including software bugs, spurious hardware errors, viruses, and intrusions. Handling such arbitrary faults in a generic fashion requires Byzantine fault tolerance (BFT).

In the past, BFT systems have mainly been considered of theoretical interest. However, numerous research efforts in recent years have contributed to bringing their performance [1]–[8], scalability [9]–[12], implementation costs [13], [14], and resilience [15]–[18] to practical levels. Unfortunately, providing these properties is not enough to make Byzantine fault tolerance truly economical: Compared to crash-tolerant solutions, BFT systems incur a higher computational cost due to their cryptographic operations. They also rely on more independence between the nodes that comprise the system because there should not exist common vulnerabilities affecting all of them simultaneously.

With regard to minimizing the resource overhead of Byzantine fault tolerance, the focus of research up to now has mostly been on reducing the number of replicas in a system. Traditional BFT systems such as PBFT [1] require $3f + 1$ replicas to tolerate up to f faults. By separating request ordering (i.e., the *agreement stage*) from request processing (i.e., the *execution stage*), the number of execution replicas can be reduced to $2f+1$ [9]. Nevertheless, $3f + 1$ replicas still need to take part during the agreement on requests. To further decrease the number of replicas, systems with a hybrid fault model, such as MinBFT [19], have been proposed, consisting of *untrusted* parts that may fail arbitrarily and *trusted* parts that are assumed to only fail by crashing [19]–[25]. Applying this approach, BFT systems can be built that comprise a total of $2f + 1$ replicas, thereby matching the size of crash-tolerant systems [26], [27].

Although reducing the provisioning costs for BFT, the state-of-the-art systems mentioned above still occupy, during most of the time, more resources than strictly necessary. Due to their adoption of the active-replication paradigm, all non-faulty replicas always participate equally in all system operations. As a consequence, in the absence of faults, more replicas order and execute requests than are actually required to make progress under benign conditions.

In this paper, we present *Resource-efficient Byzantine Fault Tolerance* (REBFT), an approach that allows BFT systems to reduce their resource footprint in the absence of faults without losing the ability to ensure liveness

- *This work was partially supported by the German Research Council (DFG) under grant no. DI 2097/1-2 and KA 3171/1-2.*

- *Tobias Distler is with the Department of Computer Science 4, Distributed Systems and Operating Systems, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany. E-mail: distler@cs.fau.de*
- *Christian Cachin is with IBM Research – Zurich, Switzerland. E-mail: cca@zurich.ibm.com*
- *Rüdiger Kapitza is with the Institute of Operating Systems and Computer Networks, TU Braunschweig, Germany. E-mail: kapitza@ibr.cs.tu-bs.de*

Manuscript received 21 May 2014; revised 19 Oct. 2015; accepted 20 Oct. 2015.
Recommended for acceptance by T. Gonzalez.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier 10.1109/TC.2015.2495213

in the presence of faults. For this purpose, we rely on two different modes of operation: one for the normal case and one for fault handling. In the normal-case operation mode, a REBFT system saves resources by keeping only a subset of the replicas active; at this point, the system is able to make progress as long as all replicas behave according to specification. If the system suspects or detects that this is not the case, it switches to a fault-handling mode, activates the remaining replicas, and tolerates the fault. Aiming at use cases in which Byzantine faults at the replica level are transient and relatively rare [7], [8], [14], [28], [29], we expect a system relying on REBFT to spend most of the time in normal-case operation mode.

Applying REBFT does not require a BFT system to be completely redesigned from scratch. Instead, by means of PBFT [1] and MinBFT [19], we show that a resource-saving operation mode for the normal case can be introduced by adding a sub protocol. The same applies to the switching mechanism that allows the system to fall back to its original BFT protocol in order to tolerate faults.

The particular contributions of this paper are:

- To present REBFT, a general approach to dynamically minimize the resource footprint of an agreement-based BFT system by making use of *passive replication*. In contrast to traditional BFT protocols, only a subset of replicas in REBFT participate in the agreement and execution of client requests during normal-case operation (Section 3). REBFT is a revised, generalized, and newly evaluated version of CheapBFT [30].
- To apply the REBFT approach to two state-of-the-art BFT systems, PBFT (Section 4) and MinBFT (Section 5), thereby showing that REBFT can be combined with different fault models.
- To evaluate the REBFT variants of PBFT and MinBFT using microbenchmarks (Section 7) and using a coordination service (Section 8).

In addition, Section 2 provides background on resource usage in existing BFT protocols, Section 6 discusses important design implications of REBFT, Section 9 presents related work, and Section 10 concludes.

2 BACKGROUND

In this section, by example of a PBFT protocol instance, we provide background on how resources are used in state-of-the-art BFT systems. Similar observations can be made for MinBFT and other protocols closely related to PBFT [6], [11], [16], [21], [24].

As shown in Fig. 1, PBFT replicas handle requests by executing a sequence of protocol phases that are named after the messages sent in them. Having received a request from the client, the primary replica proposes the request to the backup replicas in a PREPREPARE message. Next, backup replicas distribute PREPARE messages to be able to confirm that they have seen the same proposal from the primary. If this is the case, replicas commit to executing the request in this protocol instance by sending COMMIT messages. Finally, replicas process the request and return the corresponding reply to the client.

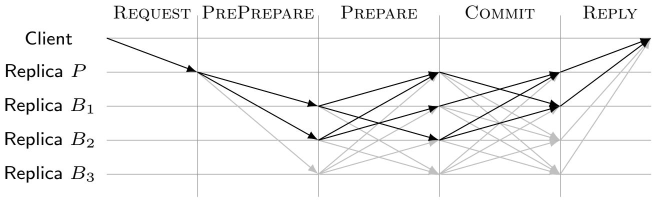


Fig. 1. Overview of the message flow in a PBFT protocol instance that is able to tolerate one fault: A primary replica P proposes a client request, which is then accepted by a group of backup replicas B_1–B_3. Of all the messages sent, only a subset (—) contributes to the stability of the result at the client; in contrast, the majority of messages (—) has no effect on the result.

In order to limit the damage that a faulty replica (i.e., a replica that does not behave according to specification) may cause, a non-faulty replica in PBFT only advances to the COMMIT phase after having obtained $2f$ PREPAREs that match the primary's proposal, and only processes a request after having collected $2f+1$ matching COMMITs. For the same purpose, a client waits for $f+1$ matching replies before accepting a result as stable: In the presence of at most f faulty replicas, $f+1$ matching messages provided by different replicas prove the message correct; in addition, a quorum of $2f+1$ messages also proves that at least $f+1$ non-faulty replicas have reached the state represented by the message. The latter is crucial to ensure safety across view changes by means of which a faulty primary can be replaced by reassigning the primary role to another replica [1].

Given these rules, PBFT is still able to make progress if up to f of its $3f+1$ replicas are faulty and either send incorrect messages or none at all. However, as all replicas actively participate in system operations, PBFT consumes significantly more resources during normal-case operation than necessary. As illustrated in Fig. 1, in the absence of faults, a majority of the messages sent does not directly influence the result at the client. Nevertheless, these messages contribute to the resource consumption of PBFT: Apart from increasing the amount of data transmitted over the network, the redundant messages also cause additional CPU overhead, as all messages in PBFT are protected by cryptography.

In conclusion, because systems like PBFT are designed for fault tolerance, at all times, they also continuously consume the resources for tolerating f faulty replicas, even if all replicas behave according to specification. As a result, there is potential to reduce the resource footprint of a BFT system in the absence of faults. In the next section, we present an approach to achieve this by implementing a clear separation between normal-case operation and fault handling.

3 The ReBFT Approach

This section introduces the ReBFT approach and its resource-saving operation mode. In Sections 4 and 5, we show how to apply ReBFT to two existing BFT systems.

3.1 Normal-case Operation

We call a ReBFT system instance a *cell* in the following. As illustrated by the examples in subsequent sections, not all systems applying ReBFT use the same number of replicas in a cell. However, they all have in common that during normal-case operation replicas assume different roles in order to minimize the resource footprint of the system: Of all the replicas in the cell, f remain *passive*, which means that they do not participate in the ordering of requests in the agreement stage and that they also do not process any requests in the execution stage. All other replicas are *active* and fully participate in both stages. Initial replica roles are assigned at system start, but we do not make any assumptions on how they are selected, as long as all replicas know the primary and are aware of which replicas are active and which replicas are passive. ReBFT ensures that non-faulty replicas always maintain a consistent view of active and passive replicas.

As passive replicas do not process requests, their states would become quickly outdated if no additional measures were taken. Therefore, in order to bring passive replicas up to speed, active replicas provide them with *state updates*; an update contains all state modifications that an active replica has performed as a result of the corresponding request. To ensure safety, a passive replica only applies an update after having obtained $f+1$ matching updates from different active replicas.

3.2 Protocol Switch

A BFT system with f passive replicas can still make progress as long as all active replicas behave according to specification and messages are delivered on time. However, in case of a fault or timeout, the agreement on requests stalls, for example, when not enough backup replicas confirm the proposal of the primary (see Section 2). ReBFT addresses such situations by initiating a *protocol switch* that allows a system to activate passive replicas and to switch to a more resilient default BFT protocol (e.g., PBFT or MinBFT). The *transition protocol* that implements the switch ensures that all replicas may resume afterwards in a consistent state.

Protocol switches in ReBFT are triggered by clients indicating that the results to their requests have not become stable within a certain period of time. In order to reach a consistent view, the active replicas then provide their local views on the progress of the agreement process in the form of *local commit histories*, which are subsequently combined to a *global commit history*. The information distributed via a global commit history not only allows the active replicas to consolidate their views among each other, but also enables the passive replicas to safely join the agreement stage.

ReBFT transition protocols may vary between different systems (see Sections 4.2 and 5.3). However, all protocols guarantee the following two properties, concerning the agreement stage ($\mathrm{P_A}$) and the execution stage ($\mathrm{P_E}$):

$\mathbf{P_A}$ *At the end of a protocol switch, any non-faulty replica knows all requests that have been committed by at least one non-faulty active replica.*

$\mathbf{P_E}$ *At the end of a protocol switch, any non-faulty passive replica knows all agreed requests that have not been processed on all non-faulty active replicas and/or whose corresponding state updates have not been applied on all non-faulty passive replicas.*

$\mathrm{P_A}$ ensures that all non-faulty (active and passive) replicas know the requests that have been committed and therefore might have been executed by a replica. As a result, non-faulty replicas will not accept different requests for the same protocol instances after the restart of the corresponding agreement stage. $\mathrm{P_E}$ serves the same purpose for requests that have successfully completed the agreement stage but not the execution stage, for example, because their state updates have not been applied to all passive replicas. The property allows non-faulty passive replicas to learn the particular requests to accept and execute for the protocol instances affected.

3.3 Fault Handling and Return to Normal Operation

Having completed the transition, ReBFT replicas only run the default BFT protocol for a number of instances x before returning to normal-case operation. The value of x is attached by the primary to the first request proposed in fault-handling mode. Backup replicas only accept the proposal if the x announced is within a predefined range, otherwise they request a view change to replace the primary. After the x-th instance in fault-handling mode has been committed, replicas automatically switch back to normal-case operation. This step does not require additional interaction, as replicas have agreed on x.

If the problem that had caused the switch to fault-handling mode still exists after the system has returned to normal-case operation, the transition protocol will be triggered again. In order to prevent a system from continuously switching back and forth in the presence of prolonged periods of faults or network problems, ReBFT increases x exponentially with every switch [14]. The value of x can be set back, either automatically after the system managed to remain in normal-case mode for a certain number of instances, or manually by an administrator, for example, after the repair of a replica.

4 Resource-efficient PBFT

This section presents RePBFT, an instance of ReBFT that relies on PBFT [1] to ensure progress in the presence of faults and consequently requires $3f+1$ replicas. However, to save resources, under benign conditions only $2f+1$ of the replicas actively participate in providing the service while f replicas remain passive. We now detail RePBFT's normal-case protocol and then present the switch to PBFT in case of suspected or detected faults.

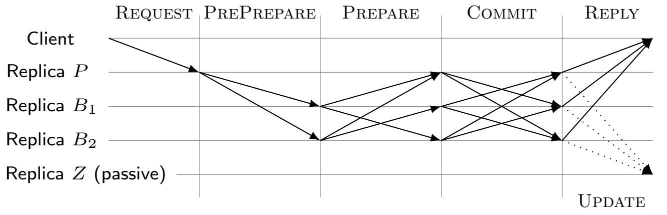


Fig. 2. Message flow of requests, agreement messages, and replies (—) as well as state updates (····) in RePBFT for a cell that is able to tolerate one Byzantine fault.

4.1 Normal-case Operation

In the absence of faults, RePBFT executes the protocol shown in Fig. 2. Active replicas assume different roles (i.e., primary or backup) and go through three protocol phases (i.e., PrePrepare, Prepare, and Commit) to agree on a request. The other f replicas remain passive and do not actively participate in the agreement protocol.

All messages exchanged between nodes in RePBFT are authenticated. For this purpose, the sender X of a message m either computes a digital signature (denoted as $\langle m \rangle_{\sigma_X}$) or an authenticator (denoted as $\langle m \rangle_{\alpha_X}$); an authenticator [1] is a vector containing a MAC for each replica computed using a session key shared between the sender and the replica. The particular form of authentication used depends on the type of a message.

4.1.1 *Agreement Stage*

Having received a request o from a client, the primary P is responsible for initiating the agreement process by assigning a sequence number s to the request and then sending a $\langle \textsc{PrePrepare}, P, o, s, p \rangle_{\alpha_P}$ message to all backups; p is the id of the current *protocol generation* and used by replicas to identify (and consequently drop) old messages. A backup B accepts a PrePrepare from the primary if it has not yet accepted another PrePrepare binding a different request o' to the same sequence number s. Having accepted a PrePrepare, the backup multicasts a $\langle \textsc{Prepare}, B, o, s, p \rangle_{\alpha_B}$ message to all active replicas informing them about the event. To complete the Prepare phase, replicas participating in the protocol must obtain Prepares from all backups and those messages must match the primary's PrePrepare. In case of success, an active replica A *prepares* the request by creating a $\langle \textsc{Commit}, A, o, s, p \rangle_{\alpha_A}$ message and sending it to all other active replicas. Once a replica has received matching Commits from all active replicas, the replica *commits* request o and forwards it to the execution stage.

In contrast to PBFT, to successfully complete a protocol instance in RePBFT during normal-case operation, all active replicas must have provided a Commit message for the corresponding request. This has two important consequences: First, if a request is committed on some active replica, it must (at least) have been prepared on all active replicas; as further discussed in Section 4.2, this property is crucial for ensuring safety during a protocol switch. Second, a protocol instance only makes progress as long as all active replicas behave according to specification and all protocol messages reach their intended recipients. If this is not given, RePBFT switches to PBFT to ensure liveness (see Section 4.2).

The agreement stage may increase throughput by running protocol instances for different client requests in parallel. The number of concurrent instances W in which a replica participates in is limited to prevent a faulty primary from exhausting the space of sequence numbers. In particular, an active replica only sends its own agreement messages for sequence numbers between a low water mark s_{low} and a high water mark $s_{high} = s_{low} + W$. In Section 4.1.3, we discuss how to advance the window defined by the water marks based on checkpoints.

4.1.2 *Execution Stage*

Active replicas execute the committed client requests in the order of their sequence numbers. Having processed a request, an active replica A sends a reply r to the client and multicasts an $\langle \textsc{Update}, A, s, u, r \rangle_{\alpha_A}$ message to all passive replicas in the cell (see Fig. 2); s is the agreement sequence number of the corresponding request and u is a state update reflecting the request's modifications to the service state. Having obtained at least $f + 1$ matching Updates from different active replicas, a passive replica has verified the update to be correct. In this case, the replica stores the reply r included in the update; this enables the passive replica to provide the client with the correct r during fault handling (see Section 4.2.1). In addition, the replica brings its state up to date by applying the update to its local application instance.

4.1.3 *Checkpoints and Garbage Collection*

In the protocol presented so far, active replicas can never be sure that their state updates have actually brought the passive replicas up to date. For example, if the active replicas in a cell were separated from the passive replicas due to a network partition, updates would not reach their intended recipients, leading the passive replicas to fall behind without the active replicas noticing. RePBFT addresses this problem by using periodic checkpoints.

A checkpoint in RePBFT is reached each time a replica has processed a client request (active replicas) or applied a state update (passive replicas) whose agreement sequence number s is a multiple of a global constant K (e.g., 100). Having reached a checkpoint, a replica R multicasts a $\langle \textsc{Checkpoint}, R, s \rangle_{\sigma_R}$ message to all other replicas. In contrast to other BFT systems [1], [4], [13], [19], [21], [24], a checkpoint in RePBFT does not require the creation of an application snapshot, since it serves primarily as a notification indicating the progress a replica has made at the execution stage.

Checkpoints in RePBFT become stable as soon as a replica manages to obtain Checkpoint notifications for the same sequence number s from all replicas in the cell. At this point, an active replica advances the window for protocol instances in which the replica participates

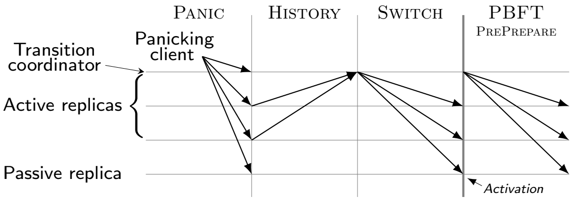


Fig. 3. Overview of the message flow during a protocol switch in a RePBFT cell that is able to tolerate one fault.

by setting the start of the window to s. Furthermore, an active replica discards client requests and stored messages that relate to sequence numbers equal to s or lower. This is safe because a stable checkpoint is a proof that all replicas in the cell have at least advanced to sequence number $s+1$ and consequently will never require information about prior protocol instances.

4.2 Protocol Switch

We next present the mechanism that allows RePBFT to perform a protocol switch to PBFT (see Fig. 3). During such a switch, replicas provide a commit history (see Section 3.2) containing information about the progress of pending protocol instances. Based on the local histories of the replicas, a designated *transition coordinator* creates and distributes a global commit history to ensure that replicas start PBFT in a consistent manner.

4.2.1 Initiating a Protocol Switch

RePBFT replicas rely on clients to inform them about suspected or detected faults. If a client C is not able to obtain a verified result to a request o within a certain period of time, the client multicasts a $\langle \text{Panic}, C, o \rangle_{\alpha_C}$ message to all replicas. In general, there could be different reasons for a result not becoming stable. Among other things, this includes scenarios in which active replicas are faulty, do not properly participate in the agreement protocol, and/or fail to return a correct reply, leaving the client with too few matching replies.

Having received a Panic, non-faulty replicas try to prevent RePBFT from unnecessarily abandoning normal-case operation. For example, a replica ignores a Panic for an old request if the same client has already issued a new request. In addition, when a replica receives a Panic for a request it has not seen before, it first relays the request to the primary and only triggers a protocol switch if it receives another Panic for the same request. As an additional protection against faulty clients, replicas wait some time between processing Panics of the same client. If the request indicated by a Panic is already covered by the latest stable checkpoint, a replica only retransmits the corresponding reply. No protocol switch is necessary in such case, as the stability of the checkpoint proves that all replicas in the cell have obtained the correct reply to the request (see Section 4.1.3), either by processing the request themselves (active replicas) or by learning it in an update (passive replicas, see Section 4.1.2). In consequence, all non-faulty replicas will send the correct reply as a reaction to the Panic, eventually allowing the client to make progress.

If none of the above scenarios apply, a replica considers a protocol switch to be necessary and thus forwards the Panic to all other replicas (omitted in Fig. 3) to ensure that they also receive the message. Furthermore, the replica runs the transition protocol presented below, which is responsible for safely switching to PBFT.

4.2.2 Creating a Local Commit History

While running the transition protocol, a non-faulty active replica stops participating in the agreement on requests. As a result, the agreement stage of the system no longer makes progress (see Section 4.1.1), and this allows the replicas to reach a consistent state. For this purpose, at the beginning of the transition protocol, each non-faulty active replica creates its local commit history. Section 4.2.3 explains how the transition coordinator assembles local histories to a global commit history.

The local commit history of a non-faulty RePBFT replica contains a set of matching Checkpoints (see Section 4.1.3) proving the validity of the latest stable checkpoint. In addition, in order to enable other replicas to learn about the requests committed (see Section 3.2), a non-faulty replica inserts information about all requests that have been prepared after the checkpoint into its local history. If a request has been prepared locally, the replica has sent a Commit, which might have led to the request being committed and executed by a non-faulty active replica (see Section 4.1.1). By adding the corresponding PrePrepare as well as $2f$ matching Prepares for such a request to its local history, a non-faulty replica proves that no other request could have been accepted for the particular sequence number.

Once its local commit history h for protocol p is complete, an active replica A sends a $\langle \text{History}, A, h, p \rangle_{\sigma_A}$ message to the transition coordinator. The role of the transition coordinator is statically assigned to one of the active replicas, for example, based on replica ids.

4.2.3 Creating a Global Commit History

When the transition coordinator receives a History message from an active replica, it only considers the local commit history contained therein when the signature on the message is valid. Based on its own local commit history and the local histories submitted by f other active replicas, the transition coordinator creates a global commit history as follows (see also Section 4.3.1). The construction ensures that the global history contains all requests that have been committed on at least one non-faulty active replica, independent of whether or not the local histories of these replicas have actually been used for the global history.

As shown in Fig. 4, a global commit history contains different *slots*, one for each agreement sequence number between the latest stable checkpoint (i.e., `#200`) and the newest agreement-protocol instance (i.e., `#205`) included

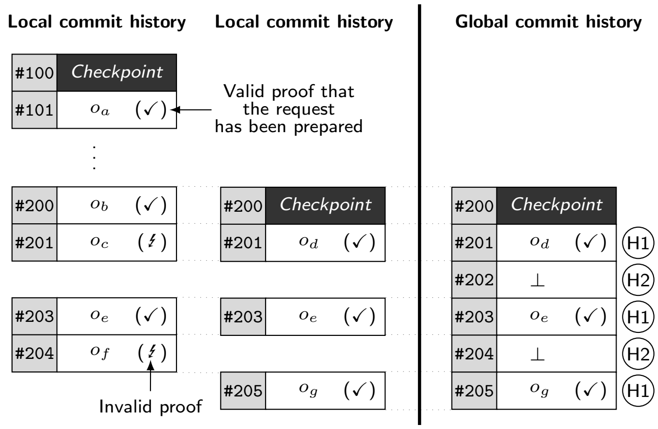


Fig. 4. Creation of a global commit history.

in the local commit histories. Since all active replicas in RePBFT have to participate in the agreement on requests in order for the system to make progress in the absence of faults, in the worst case, the number of slots in the global commit history matches the size of the agreement-protocol window (see Section 4.1.1).

After the transition coordinator has determined which slots to add to the global commit history, it chooses a value for each slot according to the following rules:

H1 A request o is chosen as the slot value if any of the $f+1$ valid local commit histories contain a valid proof (in the form of PrePrepares and Prepares) that o has been prepared (e.g., slot #201).

H2 Otherwise, if H1 does not apply, the slot value is set to a special *null request* $\perp$ that corresponds to a no-op at execution time. This rule either takes effect if none of the valid local histories contains a proof for the slot (as for slot #202) or if all proofs available for the slot are invalid (as for slot #204). In both cases, no request could have been committed in the associated protocol instance: As at least one of the $f+1$ valid local histories must have been provided by a non-faulty active replica, this replica would have included a valid proof if such a request existed.

When the global commit history h_{global} for a protocol p is complete, the transition coordinator TC multicasts a $\langle \text{Switch}, TC, h_{global}, \mathcal{H}_{local}, p\rangle_{\sigma_{TC}}$ message to all replicas in the cell; $\mathcal{H}_{local}$ is the set of $f+1$ valid local commit histories that served as basis for the global history.

4.2.4 Processing a Global Commit History

Having obtained a global commit history, both active and passive replicas verify its correctness by reproducing the steps performed by the transition coordinator to create the history. In order to be able to do this properly, the local commit histories included must have been authenticated using digital signatures (see Section 4.2.2). Otherwise, that is, if authenticators would be used, a non-faulty replica might not necessarily concur with the verdict of the transition coordinator about the validity of local commit histories.

Once a replica has verified the global commit history successfully, it uses the history to complete the switch to PBFT and initializes this protocol with a view in which the transition coordinator serves as primary. In particular, a replica creates a new PBFT instance for each slot contained in the global commit history: If the slot value is a regular request, the replica must ensure that this request will be the result of the PBFT instance; that is, if the replica is the PBFT primary, it must send a PrePrepare message for this request, and if the replica is a PBFT backup, it is only allowed to send a matching Prepare message in case the primary has proposed this request. In contrast, a slot in the global commit history containing a null request does not impose such restrictions on the corresponding PBFT instance, as a null request indicates that no regular request with this particular agreement sequence number could have previously been committed.

Note that repeating the agreement on client requests that have not been covered by the latest stable checkpoint does not lead to client requests being executed more than once. Based on the agreement sequence numbers assigned to requests, which remain the same for requests that have been committed on at least one replica (see Section 4.2.3), the execution stage is able to identify and consequently ignore such requests.

Having processed the global commit history, the protocol switch on a replica is complete. At this point, a former passive replica is considered activate as from then on it is no longer brought up to speed by state updates, but instead executes requests itself. This allows clients to retry requests that so far have not been successful.

4.2.5 Handling Faults during the Protocol Switch

In case the default transition coordinator is faulty, it might fail to deliver a valid global commit history. To address such and related problems, replicas protect the protocol switch with a timeout that expires if a replica is not able to obtain a valid global commit history within a certain period of time. The timeout should be long enough to make it unlikely that a non-faulty transition coordinator is wrongfully accused of being faulty.

When the timeout expires, an active replica A changes its protocol id from p to p' and sends its local commit history h in a $\langle \text{History}, A, h, p'\rangle_{\sigma_A}$ message to the transition coordinator of p', which is different from the coordinator of p, but selected based on the same algorithm (see Section 4.2.2). In addition, the replica sets up a new timeout with twice the length of the previous timeout duration. If this timeout also expires, the procedure is retried (possibly multiple times) until the switch completes successfully thanks to one of the at least $f+1$ non-faulty active replicas serving as transition coordinator.

4.3 Safety and Liveness

Below, we discuss how a global commit history enables a safe protocol switch in RePBFT, and why such a switch will eventually be performed by all non-faulty replicas.

4.3.1 Ensuring a Safe Protocol Switch

In order for passive replicas to safely join the agreement stage during a protocol switch, it is crucial for them to learn about all requests that have been committed since the latest stable checkpoint. REPBFT's transition protocol satisfies this requirement as follows: First, the global commit history is guaranteed to be based on at least one local commit history that has been provided by a non-faulty active replica. Second, the local history of each non-faulty active replica contains all committed requests that are not covered by the latest stable checkpoint.

As discussed in Section 4.2.3, a transition coordinator in REPBFT requires $f+1$ valid local commit histories from different active replicas to create a global commit history. With the normal-case protocol only making progress if all active replicas participate accordingly, it is guaranteed that each subset of $f+1$ local histories contains at least one history provided by a non-faulty active replica A that has seen the latest agreement state. Applying rule H1 to such a local history allows the transition protocol to provide property $\mathrm{P_A}$ (see Section 3.2): If a request o has been committed on one or more non-faulty active replicas, the request must at least have been prepared on some replica A; in consequence, replica A inserts a valid proof for o into its local commit history, which will eventually be also included in the global commit history due to rule H1. The only situation when a replica A may fail to supply a proof for o occurs when the request is covered by the latest stable checkpoint. However, in this case, all non-faulty replicas in the cell have either confirmed that they have executed the request (if they are active) or have applied the corresponding state update (if they are passive); otherwise the checkpoint would not have become stable.

For the same reason, a global commit history also allows REPBFT to ensure property $\mathrm{P_E}$ (see Section 3.2): If there are one or more non-faulty replicas that have not yet confirmed the execution of agreed requests (or their state updates) by sending a checkpoint, the global commit history contains valid proofs for these requests.

4.3.2 Ensuring System Progress

If an active replica fails to participate in the agreement on requests due to a fault while REPBFT is in normal-case mode, the agreement process stops (see Section 4.1.1). In contrast, faulty passive replicas only indirectly prevent the system from making progress: If at least one passive replica does not confirm to have reached a checkpoint, the checkpoint will not become stable. As a result, the agreement on requests will eventually stop because active replicas are no longer able to advance the window limiting the number of concurrent protocol instances (see Section 4.1.3). Either way, a stopped agreement process prevents the system from executing additional requests, which consequently forces clients to demand a protocol switch due to lack of replies (see Section 4.2.1). As non-faulty replicas forward the clients' PANICs, eventually all non-faulty replicas will initiate the transition protocol.

Having triggered the protocol switch locally, a transition coordinator requires $f+1$ valid local commit histories from different active replicas to create a global commit history. As at most f of the $2f+1$ active replicas in the cell are assumed to fail, it is guaranteed that eventually a transition coordinator has $f+1$ or more valid local commit histories available. Furthermore, by adjusting timeouts, REPBFT ensures that the role of transition coordinator can be assigned to different replicas in case acting transition coordinators fail to provide a valid global commit history (see Section 4.2.5).

4.4 Optimizations

Many optimizations proposed for other BFT systems can also be applied to REPBFT. In particular, this includes batching of requests and replacing requests in PREPAREs and COMMITs by their hash values. Moreover, the latter optimization can also be applied to client replies and state updates, that is, one replica sends the complete information whereas the others only provide a hash.

5 RESOURCE-EFFICIENT MINBFT

In this section, we present REMINBFT, an application of the REBFT approach to the MinBFT [19] protocol. MinBFT and REMINBFT rely on a trusted certification service to authenticate messages exchanged between replicas. This lets them operate with one protocol phase fewer than PBFT and REPBFT, respectively, and a total of $2f+1$ replicas. However, only $f+1$ replicas in REMINBFT remain active during normal-case operation. We next introduce the trusted message certification service and then describe the normal-case operation and protocol switch of REMINBFT. Since REMINBFT and REPBFT are very similar, we focus on aspects that are specific to REMINBFT in this section, which have not already been presented in the context of REPBFT before.

5.1 Message Certification Service

In order to be able to reduce the number of required replicas in a cell from $3f+1$ to $2f+1$, a faulty replica must be prevented from successfully performing *equivocation* [21]; that is, a replica must not be able to send the same logical protocol message with different contents to multiple recipients. As MinBFT, REMINBFT addresses this problem by relying on a trusted message certification service (MCS). We assume that each replica has a local MCS instance at its disposal that only fails by crashing; our REMINBFT prototype (see Section 7.1), for example, uses a special-purpose hardware component.

The local MCS instance of a replica is responsible for creating and verifying message certificates $\langle \text{CERTIFICATE}, MCS, c, proof \rangle$; MCS is the id of the instance that created the certificate, c is a counter value, and $proof$ is a cryptographically protected proof linking the certificate to the corresponding message. Message certificates provide the following property: If c_1 is the

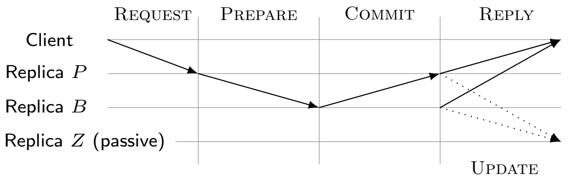


Fig. 5. Message flow of requests, agreement messages, and replies (—) as well as state updates (····) in ReMinBFT for a cell that is able to tolerate one fault.

counter value included in a certificate created by a MCS instance and c_2 is the counter value included in the *next* certificate created by the same instance, then $c_2 = c_1 + 1$ [23]. Having received a certified message, a replica only accepts it if the embedded counter matches the expected value and the proof is valid.

Requiring replicas to provide message certificates forces a faulty replica trying to perform equivocation to create multiple certificates if it wants to send messages with the same identity but different contents. Non-faulty replicas are able to protect themselves against such an attempt by following a simple rule: A non-faulty replica must process the messages received from another replica in the order determined by the counter values in their respective certificates; if the sequence of messages contains a gap, the replica must wait until the corresponding message becomes available. This way, non-faulty replicas either process all messages sent by a faulty replica in the same order (and consequently make consistent decisions) or, in case the faulty replica refuses to send any message at all, they stop processing the faulty replica's messages due to detecting gaps. Either way, a faulty replica is not able to lead non-faulty replicas into performing inconsistent actions.

As further discussed in Section 5.2, some messages in ReMinBFT are not authenticated using the message certification service and therefore do not carry a certified counter value. For such messages, there are no restrictions on the order in which they are processed.

5.2 Normal-case Operation

During normal-case operation, ReMinBFT executes the protocol shown in Fig. 5. Of the $2f + 1$ replicas in the cell, only $f + 1$ actively participate in system operations.

5.2.1 Agreement Stage

Having received a request o, the primary P proposes it by sending a $\langle \text{Prepare}, P, o, s, p \rangle_{cert_P}$ message to all active backups; s is the agreement sequence number that P has assigned to the request, p is the current protocol id, and $cert_P$ is a certificate created by the MCS (see Section 5.1) covering the entire Prepare message. When a backup B accepts the proposal of the primary, it notifies all active replicas in the cell by multicasting a $\langle \text{Commit}, B, o, s, p, cert_P \rangle_{cert_B}$ message, which is also authenticated by the MCS. Having obtained a Prepare message and f matching Commit messages, the agreement process of a request is complete; in such case, an active replica forwards the request to the execution stage.

As RePBFT (see Section 4.1.1), ReMinBFT uses a window-based mechanism to limit the number of concurrent protocol instances. However, ReMinBFT poses an additional requirement: A replica may only send a Prepare/Commit message for an instance if it has already sent a Prepare/Commit message for all preceding instances. This rule still allows different protocol instances to overlap, however, it forces a replica to process them in the order of their sequence numbers, which enables the replica to account for a continuous sequence of certified messages during a protocol switch (see Section 5.3). If a faulty replica fails to meet this requirement, all non-faulty replicas stop to process messages from the faulty replica as soon as they detect the gap. As in RePBFT, a protocol switch will be performed in such case, allowing the system to make progress.

5.2.2 Execution Stage

Similar to RePBFT (see Section 4.1.2), active ReMinBFT replicas bring passive replicas up to speed via state updates. A passive replica only applies an update after having received matching versions from all active replicas. Updates in ReMinBFT are authenticated as in RePBFT; they do not contain MCS certificates, as sending an incorrect update has the same effect as sending no update at all: the update never becomes stable.

5.2.3 Checkpoints and Garbage Collection

ReMinBFT relies on checkpoints to perform garbage collection, as RePBFT (see Section 4.1.3). For the same reason as state updates (see Section 5.2.2), Checkpoints are not certified by the MCS. In contrast to RePBFT, Checkpoints of active replicas in ReMinBFT also contain a set of counter values: For each active replica, this set contains the counter value assigned to the agreement message (i.e., the Prepare in case of the primary or the Commit in case of a backup) the replica has sent in the protocol instance of the request that triggered the checkpoint. In Section 5.3, we discuss how these counter values allow passive replicas to be activated in the course of a protocol switch. In ReMinBFT, a checkpoint only becomes stable if the counter values included in the Checkpoint notifications of active replicas also match.

5.3 Protocol Switch

Similar to RePBFT, ReMinBFT's normal-case protocol only makes progress under benign conditions, requiring a mechanism to safely switch to a more resilient protocol, in this case MinBFT. Note the following key difference between RePBFT and ReMinBFT: Whereas the transition coordinator is guaranteed to receive messages from at least $f + 1$ non-faulty active replicas in RePBFT, it must rely on information from only a *single* non-faulty active replica in REMINBFT. As a consequence, the protocol switch cannot be based on information provided by multiple replicas. Below, we discuss how this problem is addressed by REMINBFT's transition protocol.

5.3.1 Initiating a Protocol Switch

REMINBFT uses the same mechanism as REPBFT to initiate a protocol switch after a client panic (see Section 4.2.1): If an active replica decides to switch, the replica stops participating in the agreement on requests and informs the other replicas. In addition, an active replica also forwards the latest stable checkpoint notification to all passive replicas, followed by all certified messages it has sent since the creation of the checkpoint; as discussed in Section 5.3.3, this step later allows passive replicas to complete the switch to MinBFT.

5.3.2 Creating a Commit History

Having initiated a protocol switch, only a single active replica in REMINBFT creates and distributes a commit history: the transition coordinator. In contrast to REPBFT (see Section 4.2.4), the (global) commit history of the transition coordinator in REMINBFT is only based on local knowledge (i.e., there is no need for a HISTORY phase). This is necessary, as in the worst case, f active replicas may fail and the transition coordinator is the only remaining non-faulty replica that has participated in the agreement on client requests.

In order to create a commit history, the transition coordinator performs the following steps: First, it adds the latest stable checkpoint to the history. Next, it includes all certified messages that it has sent since the point in time reflected by latest stable checkpoint. When the commit history h for a protocol p is complete, the transition coordinator TC multicasts a $\langle \text{SWITCH}, TC, h, p\rangle_{cert_{TC}}$ message to all replicas in the cell; $cert_{TC}$ is the MCS certificate for the SWITCH.

A valid commit history provides a continuous sequence with regard to the counter values of certified messages: The sequence starts with the transition coordinator's counter value in the stable checkpoint certificate (see Section 5.2.3) and ends with the counter value of the MCS certificate in the SWITCH message. As a non-faulty transition coordinator includes all certified messages in between, the sequence of counter values is complete; a history that contains gaps is invalid and will not be processed by non-faulty replicas. As a result, a valid commit history necessarily contains information about all protocol instances in which the transition coordinator has participated. This allows other replicas to reach a consistent state by processing the history.

5.3.3 Processing a Commit History

When a non-faulty (active or passive) replica receives a valid commit history from the transition coordinator, the replica initializes MinBFT with a view in which the transition coordinator serves as primary. For each request in the commit history, the replica creates a new MinBFT instance and only accepts the request included in the message as the outcome of this particular instance.

In order for passive replicas in REMINBFT to complete a protocol switch, additional measures need to be taken: While the system runs the normal-case protocol, passive replicas do not receive any certified messages from active replicas. As a result, without intervention, passive replicas would not process the first certified message they receive, which is the commit history, as from their point of view the history does not contain the next counter value expected from the transition coordinator. To address this problem, passive replicas rely on the verified counter-value information included in stable checkpoints (see Section 5.2) to update their expectations on counter values. This way, when active replicas provide the latest stable checkpoint notifications as well as subsequent certified messages at the start of the protocol switch (see Section 5.3.1), passive replicas are able to join the agreement-stage communication.

5.4 Safety and Liveness

In the following. we address safety and liveness aspects in REMINBFT. As in previous sections, we focus our discussion on the main differences to REPBFT.

5.4.1 Ensuring a Safe Protocol Switch

Although being created based on local information of a single active replica (i.e., the transition coordinator), a commit history in REMINBFT provides similar properties as the global commit history in REPBFT (see Section 4.3.1). If a client request has been committed on at least one non-faulty active replica in the cell, the request is guaranteed to be included in a valid commit history of the transition coordinator. This follows because the transition coordinator has either provided a valid PREPARE (if it has been the primary) or a valid COMMIT (if it has been a backup) for the request. In consequence, the only way for the transition coordinator to create a valid commit history is to add this message to its history (see Section 5.3).

Furthermore, a valid commit history in REMINBFT is also guaranteed to contain all requests that have been executed by at least one non-faulty active replica but whose corresponding state updates may not become stable at some non-faulty passive replicas. With confirmation from all replicas in the cell being required for a checkpoint to become stable, as in REPBFT, such requests cannot be covered by the latest stable checkpoint. Therefore, as the request in question must have been committed on at least one non-faulty replica, it is guaranteed to be included in a valid commit history.

5.4.2 Ensuring System Progress

Relying on the same mechanism as REPBFT to initiate a protocol switch, all non-faulty replicas in REMINBFT will eventually start the transition to MinBFT if the normal-case protocol no longer makes progress (see Section 4.3.2). During the switch, a faulty transition coordinator may require the coordinator role to be assigned to a different replica. However, as at most f of the $f+1$ active replicas may be faulty, there is at least one non-faulty active replica that provides a valid commit history while serving as transition coordinator.

6 DISCUSSION

In this section, we analyze the impact of faulty nodes in REBFT and discuss different implementation aspects.

6.1 Impact of Faulty Clients and Replicas

REBFT's normal-case protocol only makes progress under benign conditions. As a result, a single faulty node may cause a switch to fault-handling mode. For example, an active replica stalls progress if it fails to participate in the agreement on requests. The same is eventually true for a faulty passive replica that omits to distribute a checkpoint notification, consequently preventing the checkpoint from becoming stable. Finally, a faulty client can trigger an unnecessary protocol switch, for example, by sending a PANIC for a request for which it has already obtained a stable result. In comparison, apart from network problems, in PBFT and MinBFT in general only a faulty primary can cause the agreement process to temporarily stall by failing to propose requests.

Although a single faulty client or replica can trigger fault handling in REBFT, it cannot cause the system to switch back and forth at high frequency due to REBFT exponentially increasing the number of protocol instances x for which a system stays in fault-handling mode (see Section 3.3). In particular, a faulty primary cannot unilaterally reset x to a low value as the non-faulty replicas in the system do not accept invalid proposals for x. Also, a faulty backup is not able to prevent a proposal by a non-faulty primary from being accepted.

6.2 Assignment of Replica Roles

REBFT does not specify how the roles of active and passive replicas are assigned, only that the algorithm used for this purpose must be globally known. In the simplest case, the roles are assigned statically and never change over the lifetime of a system. However, replica roles may also be assigned dynamically, possibly selecting a different set of replicas to remain active whenever the system switches back to normal-case operation.

Apart from distinguishing active and passive replicas, also the selection of the transition coordinator is important for REBFT. In principle there are two different options: To appoint one of the active backups as transition coordinator or to select the current primary. While protocols like PBFT and MinBFT apply the former during a view change, the latter offers a key benefit in REBFT: If the primary is stable across a mode switch, an adversary cannot exploit the mechanism to force the system into taking away the primary role from a non-faulty replica. Apart from that, as any faulty node can cause a transition to fault-handling mode in REBFT, there is no inherent reason to change the primary.

6.3 Support for Proactive Recovery

Some BFT systems provide long-term resilience against faults by proactively rejuvenating replicas [1], [15], [22]. This method may readily be combined with REBFT. In particular, passive replicas can be rejuvenated during normal-case operation. On the other hand, the recovery of active replicas cannot occur during normal-case operation. A recovering active replica would need to obtain state information from sufficiently many non-faulty replicas, but there are not enough of them in REBFT's normal mode. Instead, a proactive REBFT system would periodically switch to the default protocol, which allows proactive recovery of all replicas.

7 EVALUATION

Below, we evaluate REPBFT and REMINBFT during both normal-case operation as well as protocol switches.

7.1 Environment

We conduct our experiments using a replica cluster of 8-core servers (2.3 GHz, 8 GB RAM) and a client cluster of 12-core machines (2.4 GHz, 24 GB RAM); all servers are connected with switched Gigabit Ethernet. In order to focus on the differences between the four BFT protocols, all prototypes share as much code as possible. Requiring a trusted message certification service, our MinBFT and REMINBFT prototypes rely on the FPGA-based CASH subsystem [30], which calculates a SHA-256 hash. PBFT and REPBFT use authenticators based on SHA-256. In all cases, the systems evaluated are dimensioned to be resilient against one Byzantine fault. As a result, the cells of PBFT and REPBFT comprise four replicas whereas the cells of MinBFT and REMINBFT comprise three replicas.

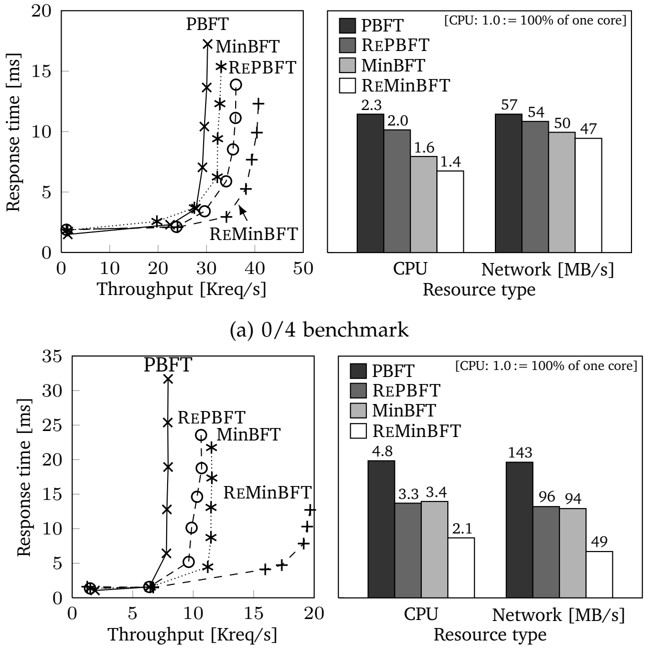
7.2 Normal-case Operation

In the following, we run two benchmarks that are commonly used [1], [4], [6], [9], [14], [19], [21], [24] to evaluate BFT protocols: a *0/4 benchmark*, in which clients repeatedly send requests with empty payloads to the service and receive replies with four-kilobyte payloads, and a *4/0 benchmark*, in which request payloads are of size four kilobytes and the payloads of replies are empty.

In the experiments below, the execution of a request in REPBFT and REMINBFT leads to an empty state update being sent to the passive replica. We evaluate the impact of different state-update sizes in Section 7.2.3.

7.2.1 0/4 Benchmark

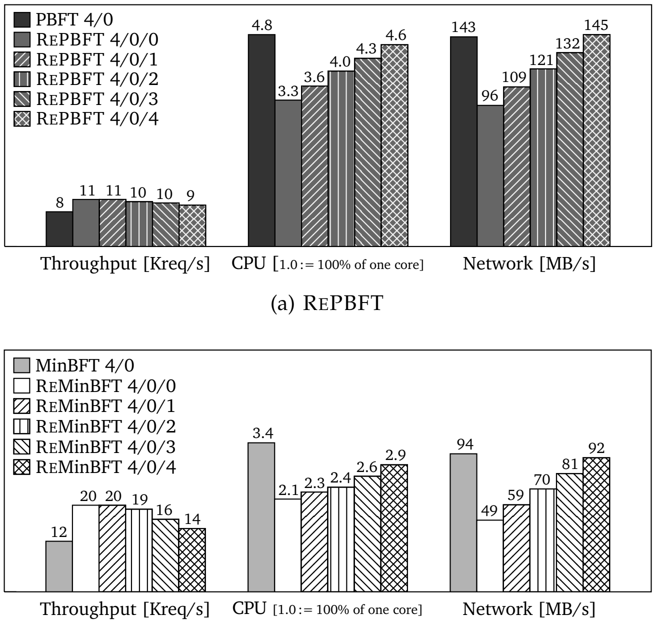
Fig. 6a shows the results for the 0/4 benchmark. The numbers for CPU and network usage are an aggregation of the resource consumption of all replicas at maximum throughput. For better comparison, the numbers are normalized to a throughput of 10,000 requests per second.



(a) 0/4 benchmark

(b) 4/0 benchmark

Fig. 6. Performance and resource-usage results for PBFT, REPBFT, MinBFT, and REMINBFT.



(a) REPBFT

(b) REMINBFT

Fig. 7. Throughput and resource-usage results for different state-update sizes in REPBFT and REMINBFT.

Performance: With clients issuing requests with empty payloads, a major factor influencing performance in this benchmark is the replicas' task to send replies. Thus, the fact that MinBFT and REMINBFT comprise one protocol phase less than PBFT and REPBFT only results in minor differences in maximum throughput (i.e., 9% for MinBFT over PBFT and 12% for REMINBFT over REPBFT). However, comparing the systems that are directly related to each other, our experiments show that by requiring fewer messages than their counterparts, REPBFT and REMINBFT allow throughput increases of 19% and 23% over PBFT and MinBFT, respectively.

The benefits of needing to authenticate fewer messages is illustrated by a comparison between MinBFT and REMINBFT. For this experiment, the limiting factor in MinBFT is the access to the FPGA providing the message certification service. As a result, MinBFT's maximum throughput is even lower than the maximum throughput of REPBFT. In contrast, REMINBFT performs better than MinBFT due to only creating/verifying certificates for two messages per protocol instance instead of three.

Resource Usage: The use of reply hashes (see Section 4.4) reduces the amount of data to be sent over the network for all four systems, as only one (active) replica has to send a full reply. Nevertheless, the need to send full replies after all, combined with the fact that for this benchmark replies are much larger than agreement messages, results in REPBFT/REMINBFT replicas transmitting moderate 5% less data over the network than PBFT/MinBFT replicas. With regard to CPU usage, the savings achieved by REPBFT and REMINBFT are higher: 11% and 15%, respectively. For both REPBFT and REMINBFT, the contribution of the passive replica to the overall system resource footprint is small: about 3% of total CPU usage and about 0.5% of network traffic.

7.2.2 4/0 Benchmark

While replies are the decisive factor in the 0/4 benchmark, the 4/0 benchmark is dominated by requests. Fig. 6b shows the results for this use case.

Performance: For the 4/0 benchmark, the maximum throughput of a system depends on the number of backup replicas participating in the protocol. Having received the request from the client, the primary in PBFT proposes it to three backups, thereby saturating its network connection at less than 8,000 requests per second. In contrast, the primary in REPBFT and MinBFT distributes each request to only two backups, allowing these systems to achieve higher throughputs of about 10,700 and 11,500 requests per second; the difference in maximum throughput between both systems illustrates the overhead of REPBFT's more complex agreement protocol. Finally, with the REMINBFT primary forwarding requests to a single active backup, REMINBFT is able to realize a maximum throughput of about 19,700 requests per second, an increase of 71% compared with MinBFT.

Resource Usage: The CPU and network usage results for the 4/0 benchmark show that the introduction of a passive replica also has a significant impact on resource consumption. Compared with PBFT, REPBFT uses 31% less CPU and sends 33% less data over the network. For REMINBFT, the savings over MinBFT are 38% (CPU) and 48% (network), respectively. In both systems, passive replicas are responsible for less than 1% of the overall CPU usage and about 0.1% of the network traffic.

7.2.3 Impact of State Updates

We investigate the impact of state-update sizes in RePBFT and ReMinBFT by extending the 4/0 benchmark to a *4/0/z benchmark*, with *z* indicating the payload size of updates. In all the experiments, only one active replica sends the full state update to the passive replica, while the other active replicas in the cell provide hashes. Drawing from the insight gained in Section 7.2.2 that the network connection of the primary replica is a bottleneck for the 4/0 benchmark, we configure the primary in RePBFT and ReMinBFT to always send update hashes.

Fig. 7 presents the maximum throughput achieved for different update sizes as well as the impact on resource usage. The results show that increasing the size of updates from zero to one kilobyte has no observable effect on the overall throughput of RePBFT and ReMinBFT; nevertheless, small non-empty updates come with additional overhead in terms of CPU and network usage.

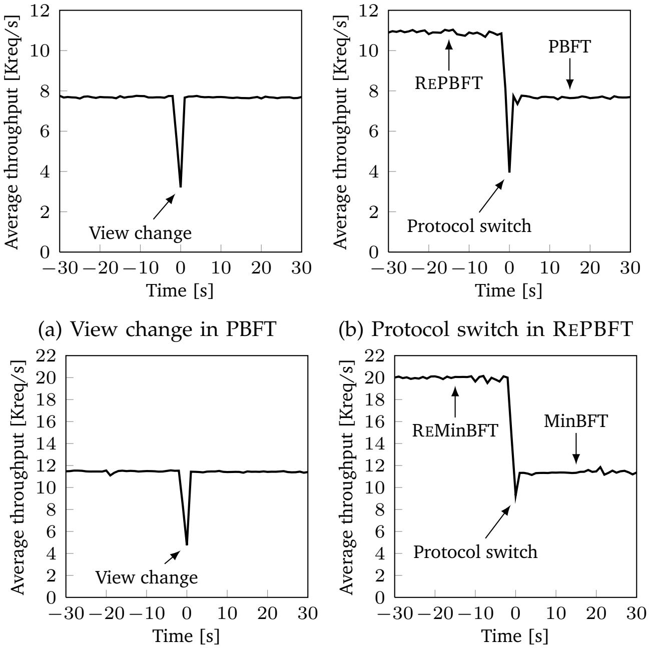
When update sizes increase, RePBFT and ReMinBFT consume more resources and eventually reach similar (network) or slightly lower (CPU) levels than PBFT and MinBFT. However, even for updates of four kilobytes, RePBFT and ReMinBFT achieve a 19% and 25% higher throughput than PBFT and MinBFT, respectively. This is due to the optimization that allows the primary to save network resources by providing update hashes.

7.3 Fault Handling

Having been designed to save resources under benign conditions, the normal-case protocols of RePBFT and ReMinBFT do not ensure progress in the presence of faults, requiring the systems to switch to the resilient PBFT and MinBFT protocol, respectively. In the following, we evaluate the performance impact of such a protocol switch and compare it to the performance impact of a view change in PBFT and MinBFT. For this purpose, we conduct a 4/0 benchmark experiment in which we manipulate the primary to stop proposing requests one protocol instance short of a new checkpoint. Thus, for a checkpoint interval of 100, the commit histories of RePBFT and ReMinBFT comprise 99 slots.

Fig. 8b shows the overall system throughput prior, during, and after a protocol switch in RePBFT. As a result of the transition protocol being executed, the throughput briefly drops to about 4,000 requests per second before stabilizing at the normal-case level for PBFT. Comparing a protocol switch in RePBFT to a view change in PBFT (see Fig. 8a), we can conclude that both cause a similar performance overhead: In both cases, the maximum latency experienced by a client in the experiments was less than 850 milliseconds.

In contrast, a protocol switch in ReMinBFT is more efficient than a view change in MinBFT, as illustrated by Figs. 8c and 8d. While changing the primary in MinBFT (similar to the protocol switch in RePBFT) requires two rounds of replica communication after having been initiated, in ReMinBFT only the transition coordinator's commit history has to be distributed for the switch (see Section 5.3). Therefore, ReMinBFT clients whose requests were affected by the reconfiguration procedure had to wait less than 700 milliseconds for their replies to become stable in our experiments.



(a) View change in PBFT (b) Protocol switch in RePBFT

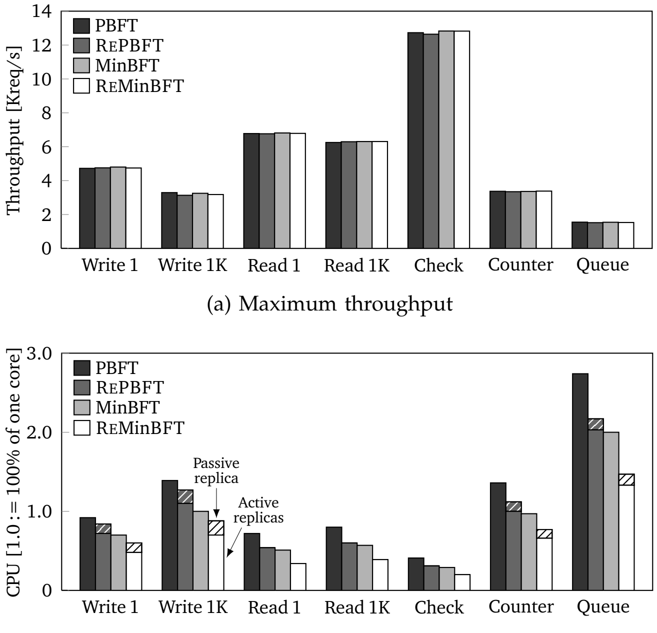
(c) View change in MinBFT (d) Protocol switch in ReMinBFT

Fig. 8. Impact of a replica fault in PBFT, RePBFT, MinBFT, and ReMinBFT for the 4/0 benchmark.

8 Case Study: Coordination Service

The evaluation in Section 7 has confirmed passive replication to be an effective means to save resources in a BFT system for use cases in which performance and resource consumption are dominated by the agreement protocol. To investigate the other end of the spectrum, scenarios dominated by the application, we have developed ReCS, a resource-efficient coordination service that relies on a relational database for persistence. ReCS exposes a hierarchical namespace (i.e., a tree) to clients and implements the ZooKeeper API [27]: After a client has created a tree node, all clients can check whether the node exists, read the node's data, or assign new data to the node. In addition, ReCS offers server-side implementations of two abstractions typically built on top of ZooKeeper: a shared counter and a distributed queue.

Internally, ReCS manages the tree in a relational database (currently MySQL 5.1.73). Different operations require different numbers of database accesses, ranging from one access for a check to three accesses for removing the head element from the queue. For the RePBFT and ReMinBFT versions of ReCS, updates consist of the SQL statements of accesses modifying the database (e.g., `UPDATE`s). In contrast, queries (i.e., `SELECT`s) do not have to be included in state updates as they do not contribute to bringing passive replicas up to speed.



(a) Maximum throughput

(b) CPU usage

Fig. 9. Throughput and resource-usage results for different coordination-service operations in different systems.

We evaluate RECS in the environment presented in Section 7.1. Figure 9 shows results for the different operations. CPU-usage numbers are normalized to a throughput of 1,000 requests per second. We omit the results for network usage as they present a similar picture as the experiments of Section 7. Due to the fact that this use case is dominated by the processing overhead of the application, all four systems evaluated achieve similar throughputs. Differences in throughput between operations can be mainly attributed to differences in the number of database accesses required per operation.

The results in Figure 9b show that with regard to the CPU overhead at passive replicas, the coordination-service operations can be divided into two categories: On the one hand, there are operations that involve modifying accesses to the service-application state, which must also be performed at passive replicas. As a result, passive replicas contribute up to 20% to the overall CPU usage. On the other hand, for operations that only read the database, resource consumption of passive replicas is negligible. In practice, coordination-service workloads comprise mostly read-only operations [27].

9 Related Work

Reducing the associated overhead is a key step to make BFT systems applicable to real-world use cases. Most optimized BFT systems introduced so far have focused on improving time and communication delays, however, and still need $3f+1$ replicas that run both the agreement and execution stage [4], [14]. Note that this is the same as in the pioneering work of Castro and Liskov [1]. The high resource demand of BFT was first addressed by Yin et al. [9] with their separation of agreement and execution that enables a system to comprise only $2f+1$ execution replicas. In a next step, systems were subdivided in trusted and untrusted components for preventing equivocation; based on a trusted subsystem, these protocols only need a total of $2f+1$ replicas [20]–[22]. The trusted subsystems may become as large as a complete virtual machine and its virtualization layer [20], [22], or may be as small as a trusted counter [19], [24].

The concept of building an agreement-based BFT system as a collection of different sub protocols has been proposed by Guerraoui et al. [14]. In their approach, they optimistically employ a very efficient but less robust agreement protocol to increase performance under benign conditions, and resort to a more resilient algorithm if needed. REBFT builds on this work and applies the concept to change the number of replicas actively involved in system operations, rather than only changing the protocol, with the goal of saving resources.

SPARE [25] and ZZ [29] are two BFT systems that exploit virtualization to minimize the number of execution replicas processing client requests during normal-case operation. However, in both systems, at all times all non-faulty replicas actively participate in the agreement on requests. In contrast, REBFT does not require a virtualized environment; furthermore, our approach reduces the number of active replicas at both the agreement stage as well as the execution stage by relying on passive replicas that only witness progress.

REBFT adopts the well-known paradigm of *passive* replication formulated for systems with crash failures, where a primary executes requests and backups only apply state updates. Protocols that involve witnesses for voting go back to work in the fail-stop model [31]. In this regard, REBFT is conceptually related to the Cheap Paxos protocol [26], in which $f+1$ main processors perform agreement and can invoke the services of up to f auxiliary processors. In case of processor crashes, the auxiliary processors take part in the agreement protocol and support the reconfiguration of the main processor set. In the Byzantine fault-tolerant Shuttle [32] protocol, witnesses do not execute requests but, unlike passive replicas in REBFT, participate in the agreement stage. Shuttle requires a centralized service for reconfiguration in case of faults. In contrast, replicas in REBFT directly initiate a switch to a more resilient protocol after having been informed by clients. As a result, REBFT is able to omit the resource overhead associated with running an additional configuration service.

10 Conclusion

This paper presented REBFT, an approach to minimize the resource consumption of BFT systems by relying on a normal-case operation mode in which only a subset of replicas actively participate in the ordering and execution of requests. Introducing such a mode does not require a system to be redesigned from scratch. Instead, a resource-saving protocol for the normal case can be derived from existing Byzantine fault-tolerant protocols, as illustrated by the examples of PBFT and MinBFT.

References

[1] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Trans. on Computer Systems*, vol. 20, no. 4, pp. 398–461, 2002.

[2] R. Kotla and M. Dahlin, "High throughput Byzantine fault tolerance," in *Proc. of the 34th Intl. Conf. on Dependable Systems and Networks (DSN '04)*, 2004, pp. 575–584.

[3] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ replication: A hybrid quorum protocol for Byzantine fault tolerance," in *Proc. of the 7th Symp. on Operating Systems Design and Implementation (OSDI '06)*, 2006, pp. 177–190.

[4] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine fault tolerance," in *Proc. of the 21st Symp. on Operating Systems Principles (SOSP '07)*, 2007, pp. 45–58.

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

[6] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin one's wheels? Byzantine fault tolerance with a spinning primary," in *Proc. of the 28th Symp. on Reliable Distributed Systems (SRDS '09)*, 2009, pp. 135–144.

[7] J. Hendricks, S. Sinnamohideen, G. R. Ganger, and M. K. Reiter, "Zzyzx: Scalable fault tolerance through Byzantine locking," in *Proc. of the 40th Intl. Conf. on Dependable Systems and Networks (DSN '10)*, 2010, pp. 363–372.

[8] T. Distler and R. Kapitza, "Increasing performance in Byzantine fault-tolerant systems with on-demand replica consistency," in *Proc. of the 6th Europ. Conf. on Computer Systems (EuroSys '11)*, 2011, pp. 91–105.

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

[10] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable Byzantine fault-tolerant services," in *Proc. of the 20th Symp. on Operating Systems Principles (SOSP '05)*, 2005, pp. 59–74.

[11] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Steward: Scaling Byzantine fault-tolerant replication to wide area networks," *Trans. on Dependable and Secure Computing*, vol. 7, no. 1, pp. 80–93, 2010.

[12] M. Kapritsos, Y. Wang, V. Quéma, A. Clement, L. Alvisi, and M. Dahlin, "All about Eve: Execute-verify replication for multicore servers," in *Proc. of the 10th Symp. on Operating Systems Design and Implementation (OSDI '12)*, 2012, pp. 237–250.

[13] M. Castro, R. Rodrigues, and B. Liskov, "BASE: Using abstraction to improve fault tolerance," *ACM Trans. on Computer Systems*, vol. 21, no. 3, pp. 236–269, 2003.

[14] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 BFT protocols," in *Proc. of the 5th Europ. Conf. on Computer Systems (EuroSys '10)*, 2010, pp. 363–376.

[15] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Veríssimo, "Resilient intrusion tolerance through proactive and reactive recovery," in *Proc. of the 13th Pacific Rim Intl. Symp. on Dependable Computing (PRDC '07)*, 2007, pp. 373–380.

[16] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making Byzantine fault tolerant systems tolerate Byzantine faults," in *Proc. of the 6th Symp. on Networked Systems Design and Implementation (NSDI '09)*, 2009, pp. 153–168.

[17] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis, "Zeno: Eventually consistent Byzantine-fault tolerance," in *Proc. of the 6th Symp. on Networked Systems Design and Implementation (NSDI '09)*, 2009, pp. 169–184.

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

[19] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo, "Efficient Byzantine fault tolerance," *IEEE Trans. on Computers*, vol. 62, no. 1, pp. 16–30, 2011.

[20] M. Correia, N. F. Neves, and P. Veríssimo, "How to tolerate half less one Byzantine nodes in practical distributed systems," in *Proc. of the 23rd Symp. on Reliable Distributed Systems (SRDS '04)*, 2004, pp. 174–183.

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

[22] H. P. Reiser and R. Kapitza, "Hypervisor-based efficient proactive recovery," in *Proc. of the 26th Symp. on Reliable Distributed Systems (SRDS '07)*, 2007, pp. 83–92.

[23] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "TrInc: Small trusted hardware for large distributed systems," in *Proc. of the 6th Symp. on Networked Systems Design and Implementation (NSDI '09)*, 2009, pp. 1–14.

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

[25] T. Distler, R. Kapitza, I. Popov, H. P. Reiser, and W. Schröder-Preikschat, "SPARE: Replicas on hold," in *Proc. of the 18th Network and Distributed System Security Symp. (NDSS '11)*, 2011, pp. 407–420.

[26] L. Lamport and M. Massa, "Cheap Paxos," in *Proc. of the 34th Intl. Conf. on Dependable Systems and Networks (DSN '04)*, 2004, pp. 307–314.

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

[28] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "FARSITE: Federated, available, and reliable storage for an incompletely trusted environment," in *Proc. of the 5th Symp. on Operating Systems Design and Implementation (OSDI '02)*, 2002, pp. 1–14.

[29] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet, "ZZ and the art of practical BFT execution," in *Proc. of the 6th Europ. Conf. on Computer Systems (EuroSys '11)*, 2011, pp. 123–138.

[30] 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 *Proc. of the 7th Europ. Conf. on Computer Systems (EuroSys '12)*, 2012, pp. 295–308.

[31] J.-F. Pâris, "Voting with witnesses: A consistency scheme for replicated files." in *Proc. of the 6th Intl. Conf. on Distributed Computing Systems (ICDCS '86)*, 1986, pp. 606–612.

[32] R. van Renesse, C. Ho, and N. Schiper, "Byzantine chain replication," in *Principles of Distributed Systems*, 2012, pp. 345–359.

Tobias Distler received the PhD degree in computer science from Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), where he is also currently an assistant professor in the System Software Group. He has been involved in several national and international research projects, including REFIT, TClouds, VM-FIT, and FOREVER. His main research interests are distributed systems, state-machine replication, and Byzantine fault tolerance.

Christian Cachin is a researcher at IBM Research – Zurich, interested in cryptography, security, and distributed systems. He graduated with a Ph.D. in Computer Science from ETH Zurich and has held visiting positions at MIT and at EPFL. His current research addresses the security of cloud computing, secure protocols for distributed systems, and cryptography.

Rüdiger Kapitza received the PhD degree in computer science from Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU). He is currently a professor at TU Braunschweig, where he leads the Distributed Systems Group. His research interests can be broadly summarized as systems research targeting fault-tolerant and secure distributed systems with special focus on state-machine replication, and Byzantine fault tolerance.