CheapBFT: Resource-efficient Byzantine Fault Tolerance*

Rüdiger Kapitza[1] Johannes Behl[2] Christian Cachin[3] Tobias Distler[2] Simon Kuhnle[2]
Seyed Vahid Mohammadi[4] Wolfgang Schröder-Preikschat[2] Klaus Stengel[2]

[1]TU Braunschweig [2]Friedrich–Alexander University Erlangen–Nuremberg
[3]IBM Research – Zurich [4]KTH – Royal Institute of Technology

Abstract

One of the main reasons why Byzantine fault-tolerant (BFT) systems are not widely used lies in their high resource consumption: $3f + 1$ replicas are necessary to tolerate only f faults. Recent works have been able to reduce the minimum number of replicas to $2f + 1$ by relying on a trusted subsystem that prevents a replica from making conflicting statements to other replicas without being detected. Nevertheless, having been designed with the focus on fault handling, these systems still employ a majority of replicas during normal-case operation for seemingly redundant work. Furthermore, the trusted subsystems available trade off performance for security; that is, they either achieve high throughput or they come with a small trusted computing base.

This paper presents CheapBFT, a BFT system that, for the first time, tolerates that *all but one* of the replicas active in normal-case operation become faulty. CheapBFT runs a composite agreement protocol and exploits passive replication to save resources; in the absence of faults, it requires that only $f + 1$ replicas actively agree on client requests and execute them. In case of suspected faulty behavior, CheapBFT triggers a transition protocol that activates f extra passive replicas and brings all non-faulty replicas into a consistent state again. This approach, for example, allows the system to safely switch to another, more resilient agreement protocol. CheapBFT relies on an FPGA-based trusted subsystem for the authentication of protocol messages that provides high performance and comprises a small trusted computing base.

Categories and Subject Descriptors D.4.7 [*Organization and Design*]: Distributed Systems; C.4 [*Performance of Systems*]: Fault Tolerance

General Terms Design, Performance, Reliability

Keywords Byzantine Failures; Resource Efficiency

1. Introduction

In an ongoing process, conventional computing infrastructure is increasingly replaced by services accessible over the Internet. On the one hand, this development is convenient for both users and providers as availability increases while provisioning costs decrease. On the other hand, it makes our society more and more dependent on the well-functioning of these services, which becomes evident when services fail or deliver faulty results to users.

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. In 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, Byzantine fault-tolerant systems have mainly been considered of theoretical interest. However, numerous research efforts in recent years have contributed to making BFT systems practical: their performance has become much better [4, 9, 17, 18], the number of required replicas has been reduced [8, 33, 34], and methods for adding diversity and for realizing intrinsically different replicas with varying attack surfaces have been introduced [3, 24]. Therefore, a debate has been started lately on why, despite all this progress, industry is reluctant to actually exploit the available research [6, 19]. A key outcome of this debate is that economical reasons, mainly the systems' high resource demand, prevent current BFT systems from being widely used. Based on this assessment, our work aims at building resource-efficient BFT systems.

Traditional BFT systems, like PBFT [4], require $3f + 1$ replicas to tolerate up to f faults. By separating request ordering (i. e., the *agreement stage*) from request process-

* This work was partially supported by the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 257243 (TClouds project: `http://www.tclouds-project.eu/`) and by the German Research Council (DFG) under grant no. KA 3171/1.

ing (i. e., the *execution stage*), the number of execution replicas can be reduced to $2f + 1$ [34]. Nevertheless, $3f + 1$ replicas still need to take part in the agreement of requests. To further decrease the number of replicas, systems with a hybrid fault model have been proposed that consist of *untrusted* parts that may fail arbitrarily and *trusted* parts which are assumed to only fail by crashing [5, 8, 21, 23, 30, 31, 33]. Applying this approach, virtualization-based BFT systems can be built that comprise only $f+1$ execution replicas [33]. Other systems [5, 8, 30, 31] make use of a hybrid fault model to reduce the number of replicas at both stages to $2f + 1$ by relying on a trusted subsystem to prevent *equivocation*; that is, the ability of a replica to make conflicting statements.

Although they reduce the provisioning costs for BFT, these state-of-the-art systems have a major disadvantage: they either require a large trusted computing base, which includes the complete virtualization layer [23, 30, 33], for example, or they rely on trusted subsystems for authenticating messages, such as a trusted platform module (TPM) or a smart card [21, 31]. These subsystems impose a major performance bottleneck, however. To address these issues, we present *CheapBFT*, a resource-efficient BFT system that relies on a novel FPGA-based trusted subsystem called *CASH*. Our current implementation of CASH is able to authenticate more than 17,500 messages per second and has a small trusted computing base of only about 21,500 lines of code.

In addition, CheapBFT advances the state of the art in resource-efficient BFT systems by running a composite agreement protocol that requires only $f + 1$ actively participating replicas for agreeing on requests during normal-case operation. The agreement protocol of CheapBFT consists of three subprotocols: the normal-case protocol *CheapTiny*, the transition protocol *CheapSwitch*, and the fall-back protocol *MinBFT* [31]. During normal-case operation, CheapTiny makes use of passive replication to save resources; it is the first Byzantine fault-tolerant agreement protocol that requires only $f+1$ *active* replicas. However, CheapTiny is not able to tolerate faults, so that in case of suspected or detected faulty behavior of replicas, CheapBFT runs CheapSwitch to bring all non-faulty replicas into a consistent state. Having completed CheapSwitch, the replicas temporarily execute the MinBFT protocol, which involves $2f + 1$ active replicas (i. e., it can tolerate up to f faults), before eventually switching back to CheapTiny.

The particular contributions of this paper are:

- To present and evaluate the CASH subsystem (Section 2). CASH prevents equivocation and is used by CheapBFT for message authentication and verification.
- To describe CheapBFT's normal-case agreement protocol CheapTiny, which uses passive replication to save resources (Section 4). CheapTiny works together with the novel transition protocol CheapSwitch, which allows to abort CheapTiny in favor of a more resilient protocol when faults have been suspected or detected (Section 5).
- To evaluate CheapBFT and related BFT systems with different workloads and a Byzantine fault-tolerant variant of the ZooKeeper [16] coordination service (Section 7).

In addition, Section 3 provides an overview of CheapBFT and its system model. Section 6 outlines the integration of MinBFT [31]. Section 8 discusses design decisions, Section 9 presents related work, and Section 10 concludes.

2. Preventing Equivocation

Our proposal of a resource-efficient BFT system is based on a trusted subsystem that prevents *equivocation*; that is, the ability of a node to make conflicting statements to different participants in a distributed protocol. In this section, we give background information on why preventing equivocation allows one to reduce the minimum number of replicas in a BFT system from $3f + 1$ to $2f + 1$. Furthermore, we present and evaluate CheapBFT's FPGA-based CASH subsystem used for message authentication and verification.

2.1 From $3f + 1$ Replicas to $2f + 1$ Replicas

In traditional BFT protocols like PBFT [4], a dedicated replica, the *leader*, proposes the order in which to execute requests. As a malicious leader may send conflicting proposals to different replicas (equivocation), the protocol requires an additional communication round to ensure that all non-faulty replicas act on the same proposal. In this round, each non-faulty replica echoes the proposal it has received from the leader by broadcasting it to all other replicas, enabling all non-faulty replicas to confirm the proposal.

In recent years, alternative solutions have been introduced to prevent equivocation, which eliminate the need for the additional round of communication [31] and/or reduce the minimum number of replicas in a BFT system from $3f + 1$ to $2f + 1$ [5, 8, 31]. Chun et al. [5], for example, present an attested append-only memory (A2M) that provides a trusted log for recording the messages transmitted in a protocol. As every replica may access the log independently to validate the messages, non-faulty replicas are able to detect when a leader sends conflicting proposals.

Levin et al. [21] show that it is sufficient for a trusted subsystem to provide a monotonically increasing counter. In their approach, the subsystem securely assigns a unique counter value to each message and guarantees that it will never bind the same counter value to a different message. Hence, when a replica receives a message, it can be sure that no other replica ever sees a message with the same counter value but different content. As each non-faulty replica validates that the sequence of counter values of messages received from another replica does not contain gaps, malicious replicas cannot equivocate messages. Levin et al. used the trusted counter to build A2M, from which a BFT system with $2f + 1$ replicas has been realized.

We propose CheapBFT, a system with only $f + 1$ active replicas, built directly from the trusted counter. In the following, we present the trusted counter service in CheapBFT.

2.2 The CASH Subsystem

The *CASH* (*C*ounter *A*ssignment *S*ervice in *H*ardware) subsystem is used by CheapBFT for message authentication and verification. To prevent equivocation, we require each replica to comprise a trusted CASH subsystem; it is initialized with a secret key and uniquely identified by a subsystem id, which corresponds to the replica that hosts the subsystem. The secret key is shared among the subsystems of all replicas. Apart from the secret key, the internal state of a subsystem as well as the algorithm used to authenticate messages may be known publicly.

For now, we assume that the secret key is manually installed before system startup. In a future version, every CASH subsystem would maintain a private key and expose the corresponding public key. A shared secret key for every protocol instance may be generated during initialization, encrypted under the public key of every subsystem, and transported securely to every replica.

2.2.1 Trusted Counter Service

CASH prevents equivocation by issuing *message certificates* for protocol messages. A message certificate is a cryptographically protected proof that a certain CASH instance has bound a unique counter value to a message. It comprises the id of the subsystem that issued the certificate, the counter value assigned, and a message authentication code (MAC) generated with the secret key. Note that CASH only needs symmetric-key cryptographic operations for message authentication and verification, which are much faster than public-key operations.

The basic version of CASH provides functions for creating ($createMC$) and verifying ($checkMC$) message certificates (see Figure 1). When called with a message m, the $createMC$ function increments the local counter and uses the secret key K to generate a MAC a covering the local subsystem id S, the current counter value c, and the message (L. 7-8). The message certificate mc is then created by appending S, c, and a (L. 9). To attest a certificate issued by another subsystem s, the $checkMC$ function verifies the certificate's MAC and uses a function $isNext()$ to validate that the sequence of messages the local subsystem has received from subsystem s contains no gaps (L. 14). Internally, the $isNext()$ function keeps track of the latest counter values of all subsystems and is therefore able to decide whether a counter value c_s assigned to a message is the next in line for subsystem s. If this is the case, the $isNext()$ function increments the counter corresponding to subsystem s and returns success; otherwise, the counter remains unchanged.

To support distinct counter instances in a protocol and several concurrent protocols, the full version of CASH supports multiple counters, each specified by a different *counter name*. All counters to be used have to be provisioned during initialization. In the counter implementation, the name becomes a part of the argument passed to the MAC for the creation and verification of message certificates. In the remainder of this paper, the counter name is written as a subscript to CASH operations (e. g., $createMC_c$ for counter c).

```
 1 **upon** initialization **do**
 2     $K$ := secret key;
 3     $S$ := local subsystem id;
 4     $c := 0$;

 6 **upon call** $createMC(m)$ **do**
 7     $c := c + 1$;
 8     $a :=$ MAC($K, S\|c\|m$);
 9     $mc := (S, c, a)$;
10     **return** $mc$;

12 **upon call** $checkMC(mc, m)$ **do**
13     $(s, c_s, a) := mc$;
14     **if** MAC($K, s\|c_s\|m$) $= a$ **and** $isNext(s, c_s)$ **do**
15         **return** TRUE;
16     **else**
17         **return** FALSE;
```

Figure 1. Implementation of CASH's trusted counter.

Furthermore, CASH provides operations for verifying a certificate without checking the correspondence of the counter values and without the side-effect of incrementing the counter in $isNext()$; there are also administrative operations for reading the subsystem id, the configured counter names, and the values of all internal counters. These operations are omitted from Figure 1. There are no means for the host system to modify subsystem id, counter names, or counter values after the initialization stage.

2.2.2 Implementation

We developed CASH to meet the following design goals:

- **Minimal trusted computing base**: The code size of CASH must be small to reduce the probability of program errors that could be exploited by attackers. Given its limited functionality, there is no need to trust an entire (hardened) Linux kernel [8] or hypervisor [23].
- **High performance**: As every interaction between replicas involves authenticated messages, we require CASH to handle thousands of messages per second. Therefore, the use of trusted platform modules or smart cards is not an option, as on such systems a single authentication operation takes more than 100 milliseconds [21, 31].

Our implementation of CASH is based on a commodity Xilinx Spartan-3 XC3S1500 FPGA mounted on a dedicated PCI card. Both the program code and the secret key are stored on the FPGA and cannot be accessed or modified by the operating system of the host machine. The only way to reprogram the subsystem is by attaching an FPGA programmer, which requires physical access to the machine.

As depicted in Figure 2, applications communicate with the FPGA via a character device (i. e., `/dev/cash`). To authenticate a message, for example, the application first writes

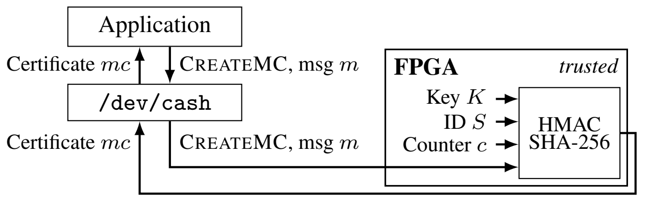


Figure 2. Creation of a message certificate mc for a message m using the FPGA-based trusted CASH subsystem.

both a CREATEMC op code and the message to the device, and then retrieves the message certificate as soon it becomes available. Our current prototype uses an HMAC-SHA-256 for the authentication of messages.

2.2.3 Integration with CheapBFT

In CheapBFT, replicas use the CASH subsystem to authenticate all messages intended for other replicas. However, this does not apply to messages sent to clients, as those messages are not subject to equivocation. To authenticate a message, a replica first calculates a hash of the message and then passes the hash to CASH's $createMC$ function. Creating a message certificate for the message hash instead of the full message increases the throughput of the subsystem, especially for large messages, as less data has to be transferred to the FPGA. To verify a message received from another replica, a replica calls the $checkMC$ function of its local CASH instance, passing the message certificate received as well as a hash of the message. Note that, for simplicity, we omit the use of this hash in the description of CheapBFT.

2.2.4 Performance Evaluation

We evaluate the performance of the CASH subsystem integrated with an 8-core machine (2.3 GHz, 8 GB RAM) and compare CASH with three other subsystems that provide the same service of assigning counter values to messages:

- **SoftLib** is a library that performs message authentication and verification completely in software. As it runs in the same process as the replica and therefore does not require any additional communication, we consider its overhead to be minimal. Note, however, that it is not feasible to use SoftLib in a BFT setting with $2f + 1$ replicas because trusting SoftLib would imply trusting the whole replica.
- **SSL** is a local OpenSSL server running in a separate process on the replica host. Like SoftLib, we evaluate SSL only for comparison, as it would also not be safe to use this subsystem in a BFT system with $2f + 1$ replicas.
- **VM-SSL** is a variant of SSL, in which the OpenSSL server runs in a Xen domain on the same host, similar to the approach used in [30]. Relying on VM-SSL requires one to trust that the hypervisor enforces isolation.

In this experiment, we measure the time it takes each subsystem variant to create certificates for messages of different sizes, which includes computing a SHA-256 hash (32 bytes)

Subsystem	Message size			
	32 B (no hashing)	32 B	1 KB	4 KB
VM-SSL	1013	1014	1015	1014
SSL	67	69	86	139
SoftLib	4	4	17	55
CASH	57	58	77	131

(a) Creation overhead for a certificate depending on message size.

Subsystem	Message size			
	32 B (no hashing)	32 B	1 KB	4 KB
VM-SSL	1013	1013	1013	1012
SSL	67	69	87	140
SoftLib	4	4	17	55
CASH	60	62	80	134

(b) Verification overhead for a certificate depending on message size.

Table 1. Overhead (in microseconds) for creating and verifying a message certificate in different subsystems.

over a message and then authenticating only the hash, not the full message (see Section 2.2.3). In addition, we evaluate the verification of message certificates. Table 1 presents the results for message authentication and verification for the four subsystems evaluated. The first set of values excludes the computation of the message hash and only reports the times it takes the subsystems to authenticate/verify a hash. With all four trusted counter service implementations only relying on symmetric-key cryptographic operations, the results in Tables 1a and 1b show a similar picture.

In the VM-SSL subsystem, the overhead for communication with the virtual machine dominates the authentication process and leads to results of more than a millisecond, independent of message size. Executing the same binary as VM-SSL but requiring only local socket communication, SSL achieves a performance in the microseconds range. In SoftLib, which does not involve any inter-process communication, the processing time significantly increases with message size. In our CASH subsystem, creating a certificate for a message hash takes 57 microseconds, which is mainly due to the costs for communication with the FPGA. As a result, CASH is able to authenticate more than 17,500 messages per second. Depending on the message size, computing the hash adds up 1 to 74 microseconds per operation; however, as hash creation is done in software, this can be done in parallel with the FPGA authenticating another message hash. The results in Table 1b show that in CASH the verification of a certificate for a message hash takes about 5% longer than its creation. This is due to the fact that in order to check a certificate, the FPGA not only has to recompute the certificate but also needs to perform a comparison.

Note that we did not evaluate a subsystem based on a trusted platform module (TPM), as the TPMs currently available only allow a single increment operation every 3.5 seconds to protect their internal counter from burning out too soon [31]. A TPM implementation based on reconfigurable hardware that could be adapted to overcome this issue did not reach the prototype status due to hardware limitations [11]. Alternative implementations either perform substantial parts in software, which makes them comparable to the software-based systems we presented, or suffer from the same problems as commodity solutions [1, 12].

Furthermore, we did not measure the performance of a smart-card-based subsystem: in [21], Levin et al. report a single authentication operation with 3-DES to take 129 milliseconds, and the verification operation to take 86 milliseconds using a smart card. This is orders of magnitude slower than the performance of CASH.

2.2.5 Trusted Computing Base

Besides performance, the complexity of a trusted subsystem is crucial: the more complex a subsystem, the more likely it is to fail in an arbitrary way, for example, due to an attacker exploiting a vulnerability. In consequence, to justify the assumption of the subsystem being trusted, it is essential to minimize its trusted computing base.

Table 2 outlines that the basic counter logic and the routines necessary to create and check message certificates are similar in complexity for both SSL variants and CASH. However, the software-based isolation and execution substrate for SSL and VM-SSL are clearly larger albeit we use the conservative values presented by Steinberg and Kauer [26]. In contrast, the trusted computing base of a TPM is rather small: based on the TPM emulator implementation of Strasser and Stamer [28], we estimate its size to be about 20 KLOC, which is only slightly smaller than the trusted computing base of CASH. For a smartcard-based solution, we assume similar values for the counter logic and certificate handling as for CASH. In addition some runtime support has to be accounted.

Going one step beyond approximating code complexity, it has to be noted that FPGAs, as used by CASH, per se are less resilient to single event upsets (e. g., bit flips caused by radiation) compared to dedicated hardware. However, fault-tolerance schemes can be applied that enable the use of FPGAs even in the space and nuclear sector [27]. Regarding code generation and the verifiability of code, similar tool chains can be used for CASH and building TPMs. Accordingly, their trustworthiness should be comparable.

In summary, our CASH subsystem comprises a small trusted computing base, which is comparable in size to the trusted computing base of a TPM, and similarly resilient to faults, while providing a much higher performance than readily available TPM implementations (see Section 2.2.4).

3. CheapBFT

This section presents our system model and gives an overview of the composite agreement protocol used in CheapBFT to save resources during normal-case operation; the subprotocols are detailed in Sections 4 to 6.

Subsystem		Components	KLOC	Total
VM-SSL	SSL	Linux	200.0	
		Counter logic	0.3	
		Cryptographic functions	0.4	**200.7**
		Virtualization	100.0	**300.7**
CASH		PCI core	18.5	
		Counter logic	2.2	
		Cryptographic functions	0.8	**21.5**

Table 2. Size comparison of the trusted computing bases of different subsystems in thousands of lines of code.

3.1 System Model

We assume the system model used for most BFT systems based on state-machine replication [4, 17, 18, 29–31, 34] according to which up to f replicas and an unlimited number of clients may fail arbitrarily (i. e., exhibit Byzantine faults). Every replica hosts a trusted CASH subsystem with its subsystem id set to the replica's identity. The trusted CASH subsystem may fail only by crashing and its key remains secret even at Byzantine replicas. As discussed in Section 2.2.2, this implies that an attacker cannot gain physical access to a replica. In accordance with other BFT systems, we assume that replicas only process requests of authenticated clients and ignore any messages sent by other clients.

The network used for communication between clients and replicas may drop messages, delay them, or deliver them out of order. However, for simplicity, we use the abstraction of FIFO channels, assumed to be provided by a lower layer, in the description of the CheapBFT protocols. For authenticating point-to-point messages where needed, the operations of CASH are invoked. Our system is safe in an asynchronous environment; to guarantee liveness, we require the network and processes to be partially synchronous.

3.2 Resource-efficient Replication

CheapBFT has been designed with a focus on saving resources. Compared with BFT systems like PBFT [4, 17, 18, 29, 34], it achieves better resource efficiency thanks to two major design changes: First, each CheapBFT replica has a small trusted CASH subsystem that prevents equivocation (see Section 2); this not only allows us to reduce the minimum number of replicas from $3f + 1$ to $2f + 1$ but also minimizes the number of protocol messages [5, 8, 21, 30, 31, 34]. Second, CheapBFT uses a composite agreement protocol that saves resources during normal-case operation by supporting *passive replication*.

In traditional BFT systems [4, 17, 18, 29], all (non-faulty) replicas participate in both the agreement and the execution of requests. As recent work has shown [9, 33], in the absence of faults, it is sufficient to actually process a request on only $f + 1$ replicas as long as it is guaranteed that all other replicas are able to safely obtain changes to the application state. In CheapBFT, we take this idea even further and propose

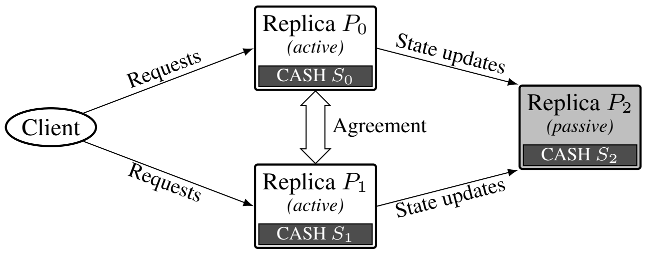


Figure 3. CheapBFT architecture with two active replicas and a passive replica ($f = 1$) for normal-case operation.

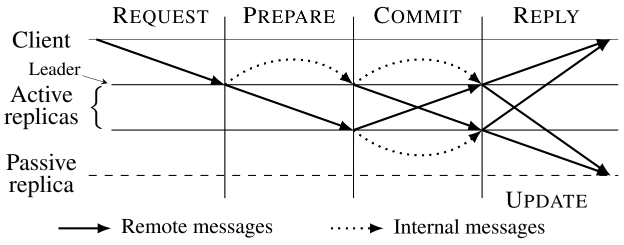


Figure 4. CheapTiny protocol messages exchanged between a client, two active replicas, and a passive replica ($f = 1$).

our CheapTiny protocol, in which only $f + 1$ *active* replicas take part in the agreement stage during normal-case operation (see Figure 3). The other f replicas remain *passive*, that is, they neither agree on requests nor execute requests. Instead, passive replicas modify their states by processing validated *state updates* provided by the active replicas. This approach minimizes not only the number of executions but also the number of protocol messages.

3.3 Fault Handling

With only $f + 1$ replicas actively participating in the protocol, CheapTiny is not able to tolerate faults. Therefore, in case of suspected or detected faulty behavior of one or more active replicas, CheapBFT abandons CheapTiny in favor of a more resilient protocol. The current CheapBFT prototype relies on MinBFT [31] for this purpose, but we could have selected other BFT protocols (e. g., A2M-PBFT-EA [5]) that make use of $2f + 1$ replicas to tolerate f faults.

During the protocol switch to MinBFT, CheapBFT runs the CheapSwitch transition protocol to ensure that replicas start the new MinBFT protocol instance in a consistent state. The main task of non-faulty replicas in CheapSwitch is to agree on a CheapTiny *abort history*. An abort history is a list of protocol messages that indicates the status of pending requests and therefore allows the remaining non-faulty replicas to safely continue agreement. In contrast to Abstract [14], which relies on a similar technique to change protocols, an abort history in CheapBFT can be verified to be correct even if it has only been provided by a single replica.

4. Normal-case Protocol: CheapTiny

CheapTiny is the default protocol of CheapBFT and designed to save resources in the absence of faults by making use of passive replication. It comprises a total of four phases of communication (see Figure 4), which resemble the phases in PBFT [4]. However, as CheapBFT replicas rely on a trusted subsystem to prevent equivocation, the CheapTiny protocol does not require a pre-prepare phase.

4.1 Client

During normal-case operation, clients in CheapBFT behave similar to clients in other BFT state-machine-replication protocols: Upon each new request, a client sends a $\langle \text{REQUEST}, m \rangle$ message authenticated by the client's key to the leader; m is a request object containing the id of the client, the command to be executed, as well as a client-specific sequence number that is used by the replicas to ensure exactly-once semantics. After sending the request, the client waits until it has received $f + 1$ matching replies from different replicas, which form a proof for the correctness of the reply in the presence of at most f faults.

4.2 Replica

Taking up the separation introduced by Yin et al. [34], the internal architecture of an active CheapBFT replica can be logically divided into two stages: the *agreement stage* establishes a stable total order on client requests, whereas the *execution stage* is responsible for processing requests and for providing state updates to passive replicas. Note that as passive replicas do not take part in the agreement of requests, they also do not execute the CheapTiny agreement stage.

Both stages draw on the CASH subsystem to authenticate messages intended for other replicas. To decouple agreement messages from state updates, a replica uses two trusted counters, called ag and up.

4.2.1 Agreement Stage

During protocol initialization, each replica is assigned a unique id (see Figure 5, L. 2). Furthermore, a set of $f + 1$ active replicas is selected in a deterministic way. The active replica with the lowest id becomes the leader (L. 3-5). Similarly to other PBFT-inspired agreement protocols, the leader in CheapTiny is responsible for proposing the order in which requests from clients are to be executed. When all $f + 1$ active replicas have accepted a proposed request, the request becomes *committed* and can be processed safely.

When the leader receives a client request, it first verifies the authenticity of the request (omitted in Figure 5). If the request is valid and originates from an authenticated client, the leader then broadcasts a $\langle \text{PREPARE}, m, mc_L \rangle$ message to all active replicas (L. 7-9). The PREPARE contains the client request m and a message certificate mc_L issued by the local trusted CASH subsystem. The certificate uses the agreement-stage-specific counter ag and contains the leader's identity in the form of the subsystem id.

```
1  upon initialization do
2      $P :=$ local replica id;
3      $active := \{p_0, p_1, \ldots p_f\}$;
4      $passive := \{p_{f+1}, p_{f+2}, \ldots, p_{2f}\}$;
5      $leader := select\_leader(active)$;

7  upon receiving $\langle \text{REQUEST}, m \rangle$ such that $P = leader$ do
8      $mc_L := createMC_{ag}(m)$;
9      send $\langle \text{PREPARE}, m, mc_L \rangle$ to all in $active$;

11 upon receiving $\langle \text{PREPARE}, m, mc_L \rangle$ such that
       $(mc_L = (leader, \cdot, \cdot))$ and $checkMC_{ag}(mc_L, m)$ do
12     $mc_P := createMC_{ag}(m \| mc_L)$;
13     send $\langle \text{COMMIT}, m, mc_L, mc_P \rangle$ to all in $active$;

15 upon receiving $\mathcal{C} := \{ \langle \text{COMMIT}, m, mc_L, mc_p \rangle$ with
       $mc_p = (p, \cdot, \cdot)$ from every $p$ in $active$ such that
       $checkMC_{ag}(mc_p, m \| mc_L)$ and all $m$ are equal $\}$ do
16     $execute(m, \mathcal{C})$;
```

Figure 5. CheapTiny agreement protocol for active replicas.

```
1  upon call $execute(m, \mathcal{C})$ do
2      $(r, u) := process(m)$;
3      $uc_P := createMC_{up}(r \| u \| \mathcal{C})$;
4      send $\langle \text{UPDATE}, r, u, \mathcal{C}, uc_P \rangle$ to all in $passive$;
5      send $\langle \text{REPLY}, P, r \rangle$ to client;
```

Figure 6. CheapTiny execution-stage protocol run by active replicas to execute requests and distribute state updates.

```
1  upon receiving {
           $\langle \text{UPDATE}, r, u, \mathcal{C}, uc_p \rangle$ with $uc_p = (p, \cdot, \cdot)$
           from every $p$ in $active$
           such that $checkMC_{up}(uc_p, r \| u \| \mathcal{C})$
           and all $r$ are equal and all $u$ are equal
       } do
2      $process(u)$;
```

Figure 7. CheapTiny execution-stage protocol run by passive replicas to process updates provided by active replicas.

Upon receiving a PREPARE (L. 11), an active replica asks CASH to verify that it originates from the leader, that the message certificate is valid, and that the PREPARE is the next message sent by the leader, as indicated by the assigned counter value. This procedure guarantees that the replica only accepts the PREPARE if the sequence of messages received from the leader contains no gaps. If the message certificate has been successfully verified, the replica sends a $\langle \text{COMMIT}, m, mc_L, mc_P \rangle$ message to all active replicas (L. 13). As part of the COMMIT, the replica propagates its own message certificate mc_P for the request m, which is created by authenticating the concatenation of m and the leader's certificate mc_L (L. 12). Note that issuing a combined certificate for m and mc_L helps replicas determine the status of pending requests in case of a protocol abort, as the certificate is a proof that the replica has received and accepted both m and mc_L (see Section 5.3).

When an active replica receives a COMMIT message, it extracts the sender p from mc_p and verifies that the message certificate mc_p is valid (L. 15). As soon as the replica has obtained a set $\mathcal{C}$ of $f + 1$ valid COMMITs for the same request m (one from each active replica, as determined by the subsystem id found in the message certificates), the request is committed and the replica forwards m to the execution stage (L. 15-16). Because of our assumption of FIFO channels and because of the fact that COMMITs from all $f+1$ active replicas have to be available, CheapTiny guarantees that requests are committed in the order proposed by the leader without explicit use of a sequence number.

4.2.2 Execution Stage

Processing a request m in CheapBFT requires the application to provide two objects (see Figure 6, L. 2): a reply r intended for the client and a state update u that reflects the changes to the application state caused by the execution of m. Having processed a request, an active replica asks the CASH subsystem to create an update certificate uc_P for the concatenation of r, u, and the set of COMMITs $\mathcal{C}$ confirming that m has been committed (L. 3). The update certificate is generated using the counter up, which is dedicated to the execution stage. Next, the active replica sends an $\langle \text{UPDATE}, r, u, \mathcal{C}, uc_P \rangle$ message to all passive replicas (L. 4), and finally forwards the reply to the client (L. 5).

Upon receiving an UPDATE, a passive replica confirms that the update certificate is correct and that its assigned counter value indicates no gaps (see Figure 7, L. 1). When the replica has received $f + 1$ matching UPDATEs from all active replicas for the same reply and state update, the replica adjusts its application state by processing the state update (L. 1-2).

4.2.3 Checkpoints and Garbage Collection

In case of a protocol switch, active replicas must be able to provide an abort history indicating the agreement status of pending requests (see Section 5). Therefore, an active replica logs all protocol messages sent to other replicas (omitted in Figures 5 and 6). To prevent a replica from running out of memory, CheapTiny makes use of periodic protocol checkpoints that allow a replica to truncate its message log.

A non-faulty active replica creates a new checkpoint after the execution of every kth request; k is a system-wide constant (e. g., 200). Having distributed the UPDATE for a request q that triggered a checkpoint, the replica first creates an application snapshot. Next, the replica sends a $\langle \text{CHECKPOINT}, ash_q, cc_{ag}, cc_{up} \rangle$ message to all (active and passive) replicas, which includes a digest of the application snapshot ash_q and two checkpoint certificates, cc_{ag} and cc_{up}, issued under the two CASH counters ag and up.

Upon receiving a CHECKPOINT, a replica verifies that its certificates are correct and that the counter values assigned are both in line with expectations. A checkpoint becomes stable as soon as a replica has obtained matching checkpoints from all $f + 1$ active replicas. In this case, an active replica discards all requests up to request q as well as all corresponding PREPARE, COMMIT, and UPDATE messages.

4.2.4 Optimizations

CheapTiny allows to apply most of the standard optimizations used in Byzantine fault-tolerant protocols related to PBFT [4]. In particular, this includes batching, which makes it possible to agree on multiple requests (combined in a batch) within a single round of agreement. In the following, we want to emphasize two additional optimizations to reduce communication costs.

***Implicit Leader* COMMIT** In the protocol description in Figure 4, the leader sends a COMMIT to all active replicas after having received its own (internal) PREPARE. As this COMMIT carries no additional information, the leader's PREPARE and COMMIT can be merged into a single message that is distributed upon receiving a request; that is, all replicas treat a PREPARE from the leader as an implicit COMMIT.

Use of Hashes PBFT reduces communication costs by selecting one replica for each request to send a full reply. All other replicas only provide a hash of the reply that allows the client to prove the result correct. The same approach can be implemented in CheapTiny. Furthermore, only a single active replica in CheapTiny needs to include a full state update in its UPDATE for the passive replicas.

5. Transition Protocol: CheapSwitch

CheapTiny is optimized to save resources during normal-case operation. However, the subprotocol is not able to make progress in the presence of suspected or detected faulty behavior of replicas. In such cases, CheapBFT falls back to the MinBFT protocol, which relies on $2f + 1$ active replicas and can therefore tolerate up to f faults. In this section, we present the CheapSwitch transition protocol responsible for the safe protocol switch.

5.1 Initiating a Protocol Switch

In CheapBFT, all nodes are eligible to request the abortion of the CheapTiny protocol. There are two scenarios that trigger a protocol switch:

- A client asks for a protocol switch in case the active replicas fail to provide $f+1$ matching replies to a request within a certain period of time.
- A replica demands to abort CheapTiny if it suspects or detects that another replica does not behave according to the protocol specification, for example, by sending a false message certificate, or by not providing a valid checkpoint or state update in a timely manner.

In these cases, the node requesting the protocol switch sends a $\langle$PANIC$\rangle$ message to all (active and passive) replicas (see Figure 8). The replicas react by rebroadcasting the message

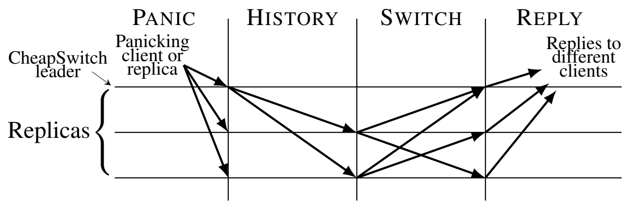


Figure 8. CheapSwitch protocol messages exchanged between clients and replicas during protocol switch ($f = 1$).

to ensure that all replicas are notified (omitted in Figure 8). Furthermore, upon receiving a PANIC, a non-faulty active replica stops to send CheapTiny protocol messages and waits for the leader of the new CheapSwitch protocol instance to distribute an abort history. The CheapSwitch leader is chosen deterministically as the active replica with the lowest id apart from the leader of the previous CheapTiny protocol.

5.2 Creating an Abort History

An abort history is used by non-faulty replicas to safely end the active CheapTiny instance during a protocol switch. It comprises the CHECKPOINTs of all active replicas proving that the latest checkpoint has become stable, as well as a set of CheapTiny protocol messages that provide replicas with information about the status of pending requests. We distinguish three status categories:

- **Decided**: The request has been committed prior to the protocol abort. The leader proves this by including the corresponding UPDATE (which comprises the set of $f+1$ COMMITs from all active replicas) in the history.
- **Potentially decided**: The request has not been committed, but prior to the protocol abort, the leader has received a valid PREPARE for the request and has therefore sent out a corresponding COMMIT. Accordingly, the request may have been committed on some active replicas. In this case, the leader includes its own COMMIT in the history.
- **Undecided**: The leader has received a request and/or a PREPARE for a request, but has not yet sent a COMMIT. As a result, the request cannot have been committed on any non-faulty replica. In this case, the leader includes the request in the abort history.

When creating the abort history, the leader of the CheapSwitch protocol instance has to consider the status of all requests that are not covered by the latest stable checkpoint. When a history h is complete, the leader asks the CASH subsystem for two history certificates $hc_{L,ag}$ and $hc_{L,up}$, authenticated by *both* counters. Then it sends a $\langle \text{HISTORY}, h, hc_{L,ag}, hc_{L,up} \rangle$ message to all replicas.

5.3 Validating an Abort History

When a replica receives an abort history from the leader of the CheapSwitch instance, it verifies that the history is correct. An abort history is deemed to be correct by a correct replica when all of the following four criteria hold:

- Both history certificates verify correctly.
- The CHECKPOINTs contained in the abort history prove that the latest checkpoint has become stable.
- Using only information contained in the abort history, the replica can reconstruct *the complete sequence* of authenticated protocol messages that the CheapSwitch leader has sent in CheapTiny since the latest checkpoint.
- The reconstructed sequence of messages does not violate the CheapTiny protocol specification.

Note that although an abort history is issued by only a single replica (i. e., the new leader), all other replicas are able to verify its correctness independently: each UPDATE contains the $f + 1$ COMMIT certificates that prove a request to be decided; each COMMIT in turn comprises a certificate that proves that the old leader has sent a PREPARE for the request (see Section 4.2). As replicas verify that all these certificates are valid and that the sequence of messages sent by the leader has no gaps, a malicious leader cannot modify or invent authenticated protocol messages and include them in the history without being detected. As a result, it is safe to use a correct abort history to get replicas into a consistent state (see Section 5.4).

Figure 9 shows an example of an abort history deemed to be correct, containing the proof CHK that the latest checkpoint has become stable, UPDATEs for three decided requests a, b, and c, and a COMMIT for a potentially decided request d. After verifying that all certificates are correct, a replica ensures that the messages in the history do not violate the protocol specification (e. g., the UPDATE for request a must comprise $f + 1$ matching COMMITs for a). Finally, a replica checks that the abort history proves the complete sequence of messages sent by the leader since the latest checkpoint; that is, the history must contain an authenticated message for every counter value of both the agreement-stage counter ag as well as the execution-stage counter up, starting from the counter values assigned to the last checkpoint and ending with the counter values assigned to the abort history.

The requirement to report a complete sequence of messages prevents equivocation by a malicious leader. In particular, a malicious leader cannot send inconsistent authenticated abort histories to different replicas without being detected: in order to create diverging histories that are both deemed to be correct, the leader would be forced to include the first authenticated history into all other histories. Furthermore, the complete message sequence ensures that all decided or potentially decided requests are included in the history: if a malicious leader, for example, sends a COMMIT for a request e after having created the history, all non-faulty replicas will detect the gap in the sequence of agreement counter values (caused by the history) and ignore the

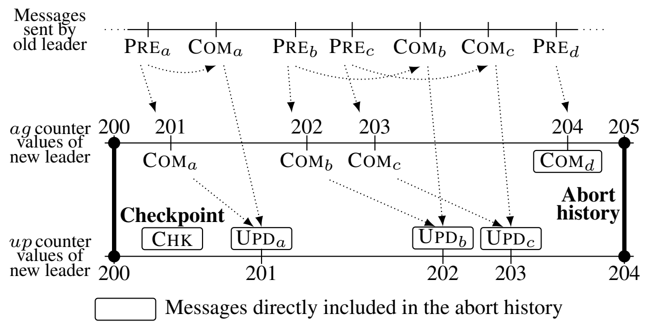


Figure 9. Dependencies of UPDATE (UPD_*) and COMMIT (COM_*) messages contained in a correct CheapTiny abort history for four requests a, b, c, and d ($f = 1$).

COMMIT. As a result, it is impossible for e to have been decided in the old CheapTiny instance. This property depends critically on the trusted counter.

5.4 Processing an Abort History

Having concluded that an abort history is correct, a replica sends a $\langle \mathrm{SWITCH}, hh, hc_{L,ag}, hc_{L,up}, hc_{P,ag}, hc_{P,up} \rangle$ message to all other replicas (see Figure 8); hh is a hash of the abort history, $hc_{L,ag}$ and $hc_{L,up}$ are the leader's history certificates, and $hc_{P,ag}$ and $hc_{P,up}$ are history certificates issued by the replica and generated with the agreement-stage counter and the update-stage counter, respectively. Note that a SWITCH is to a HISTORY what a COMMIT is to a PREPARE. When a replica has obtained a correct history and f matching SWITCH messages from different replicas, the history becomes stable. In this case, a replica processes the abort history, taking into account its local state.

First, a replica executes all decided requests that have not yet been processed locally, retaining the order determined by the history, and sends the replies back to the respective clients. Former passive replicas only execute a decided request if they have not yet processed the corresponding state update. Next, a replica executes all unprocessed potentially decided requests as well as all undecided requests from the history. This is safe, as both categories of requests have been implicitly decided by $f + 1$ replicas accepting the abort history. Having processed the history, all non-faulty replicas are in a consistent state and therefore able to safely switch to the new MinBFT protocol instance.

5.5 Handling Faults

If an abort history does not become stable within a certain period of time after having received a PANIC, a replica suspects the leader of the CheapSwitch protocol to be faulty. As a consequence, a new instance of the CheapSwitch protocol is started, whose leader is chosen deterministically as the active replica with the smallest id that has not already been leader in an immediately preceding CheapSwitch instance. If these options have all been exploited the leader of the last

CheapTiny protocol instance is chosen. To this end, the suspecting replica sends a $\langle\text{SKIP}, p_{NL}, sc_{P,ag}, sc_{P,up}\rangle$ message to all replicas, where p_{NL} denotes the replica that will now become the leader; $sc_{P,ag}$ and $sc_{P,up}$ are two skip certificates authenticated by both trusted counters ag and up, respectively. Upon obtaining $f + 1$ matching SKIPs with correct certificates, p_{NL} becomes the new leader and reacts by creating and distributing its own abort history.

The abort history provided by the new leader may differ from the old leader's abort history. However, as non-faulty replicas only accept an abort history from a new leader after having received at least $f + 1$ SKIPs proving a leader change, it is impossible that a non-faulty replica has already processed the abort history of the old leader.

Consider two abort histories h_0 and h_1 that are both deemed to be correct, but are provided by different replicas P_0 and P_1. Note that the extent to which they can differ is limited. Making use of the trusted CASH subsystem guarantees that the order (as indicated by the counter values assigned) of authenticated messages that are included in both h_0 and h_1 is identical across both histories. However, h_0 may contain messages that are not in h_1, and vice versa, for example, because one of the replicas has already received $f + 1$ COMMITs for a request, but the other replica has not yet done so. As a result, both histories may report a slightly different status for each pending request: In h_0, for example, a request may have already been decided, whereas in h_1 its is reported to be potentially decided. Also, a request may be potentially decided in one history and undecided in the other.

However, if both histories are deemed to be correct, h_0 will never report a request to be decided that is undecided in h_1. This is based on the fact that for the request to become decided on P_0, P_1 must have provided an authenticated COMMIT for the request. Therefore, P_1 is forced to include this COMMIT in h_1 to create a correct history, which upgrades the status of the request to potentially decided (see Section 5.2). In consequence, it is safe to complete the CheapSwitch protocol by processing any correct abort history available, as long as all replicas process the same history, because all correct histories contain all requests that have become decided on at least one non-faulty replica.

It is possible that the abort history eventually processed does not contain all undecided requests, for example, because the CheapSwitch leader may not have seen all PREPAREs distributed by the CheapTiny leader. Therefore, a client retransmits its request if it is not able to obtain a stable result after having demanded a protocol switch. All requests that are not executed prior to or during the CheapSwitch run are handled by the following MinBFT instance.

6. Fall-back Protocol: MinBFT

After completing CheapSwitch, a replica is properly initialized to run the MinBFT protocol [31]. In contrast to CheapTiny, all $2f + 1$ replicas in MinBFT are active, which allows the protocol to tolerate up to f faults. However, as we expect permanent replica faults to be rare [9, 14, 33], the protocol switch to MinBFT will in most cases be performed to make progress in the presence of temporary faults or periods of asynchrony. To address this issue, CheapBFT executes MinBFT for only a limited period of time and then switches back to CheapTiny, similarly to the approach proposed by Guerraoui et al. in [14].

6.1 Protocol

In MinBFT, all replicas actively participate in the agreement of requests. Apart from that, the protocol steps are similar to CheapTiny: when the leader receives a client request, it sends a PREPARE to all other replicas, which in turn respond by multicasting COMMITs, including the PREPARE certificate. Upon receiving $f + 1$ matching COMMITs, a replica processes the request and sends a reply back to the client. Similar to CheapTiny, replicas in MinBFT authenticate all agreement-stage messages using the CASH subsystem and only accept message sequences that contain no gaps and are verified to be correct. Furthermore, MinBFT also relies on stable checkpoints to garbage collect message logs.

6.2 Protocol Switch

In CheapBFT, an instance of the MinBFT protocol runs only a predefined number of agreement rounds x. When the xth request becomes committed, a non-faulty replica switches back to the CheapTiny protocol and handles all subsequent requests. Note that if the problem that led to the start of MinBFT has not yet been removed, the CheapTiny fault-handling mechanism ensures that the CheapSwitch transition protocol will be triggered once again, eventually initializing a new instance of MinBFT. This new instance uses a higher value for x to account for the prolonged period of asynchrony or faults.

7. Evaluation

In this section, we evaluate the performance and resource consumption of CheapBFT. Our test setting comprises a replica cluster of 8-core machines (2.3 GHz, 8 GB RAM) and a client cluster of 12-core machines (2.4 GHz, 24 GB RAM) that are all connected with switched Gigabit Ethernet.

We have implemented CheapBFT by adapting the BFT-SMaRt library [2]. Our CheapBFT implementation reuses BFT-SMaRt's communication layer but provides its own composite agreement protocol. Furthermore, CheapBFT relies on the CASH subsystem to authenticate and verify messages. In addition to CheapBFT and BFT-SMaRt, we evaluate an implementation of plain MinBFT [31]; note that to enable a protocol comparison the MinBFT implementation also uses our CASH subsystem. All of the following experiments are conducted with system configurations that are able to tolerate a single Byzantine fault (i. e., BFT-SMaRt: four replicas, MinBFT and CheapBFT: three replicas). In all cases, the maximum request-batch size is set to 20.

7.1 Normal-case Operation

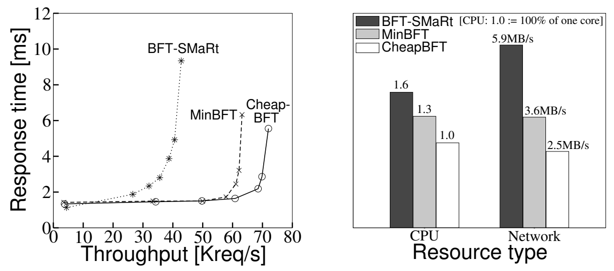
We evaluate BFT-SMaRt, MinBFT, and CheapBFT during normal-case operation using a micro benchmark in which clients continuously send empty requests to replicas; each client waits to receive an empty reply before sending a subsequent request. In the CheapBFT configuration, each client request triggers an empty update. Between test runs, we vary the number of clients from 5 to 400 to increase load and measure the average response time of an operation. With no execution overhead and only small messages to be sent, the focus of the benchmark lies on the throughput of the agreement protocols inside BFT-SMaRt, MinBFT, and CheapBFT.

The performance results in Figure 10a show that requiring only four instead of five communication steps and only $2f + 1$ instead of $3f + 1$ agreement replicas, MinBFT achieves a significantly higher throughput than BFT-SMaRt. With only the $f + 1$ active replicas taking part in the agreement of requests, a CheapBFT replica needs to handle fewer protocol messages than a MinBFT replica. As a result, CheapBFT is able to process more than 72,000 requests per second, an increase of 14% over MinBFT.

Besides performance, we evaluate the CPU and network usage of BFT-SMaRt, MinBFT, and CheapBFT. In order to be able to directly compare the three systems, we aggregate the resource consumption of all replicas in a system and normalize the respective value at maximum throughput to a throughput of 10,000 requests per second (see Figure 10b). Compared to MinBFT, CheapBFT requires 24% less CPU, which is mainly due to the fact that a passive replica does not participate in the agreement protocol and neither processes client requests nor sends replies. CheapBFT replicas also send 31% less data than MinBFT replicas over the network, as the simplified agreement protocol of CheapBFT results in a reduced number of messages. Compared to BFT-SMaRt, the resource savings of CheapBFT add up to 37% (CPU) and 58% (network).

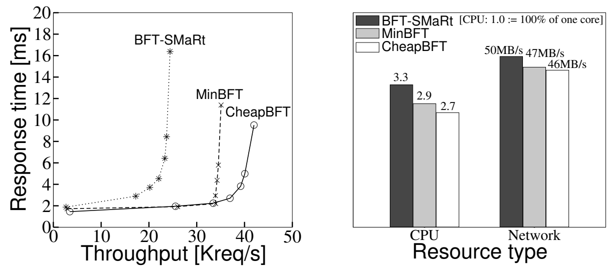
We also evaluate the three BFT systems in an experiment in which clients send empty requests and receive replies of 4 kilobyte size. Note that in this scenario, as discussed in Section 4.2.4, only a single replica responds with the actual full reply while all other replicas only provide a reply hash to the client. Figure 11 shows the results for performance and resource usage for this experiment. In contrast to the previous benchmark, this benchmark is dominated by the overhead for reply transmission: as full replies constitute the majority of network traffic, CheapBFT replicas only send 2% less data than MinBFT replicas and 8% less data than BFT-SMaRt replicas over the network. Furthermore, the need to provide a passive replica with reply hashes reduces the CPU savings of CheapBFT to 7% compared to MinBFT and 20% compared to BFT-SMaRt.

In our third micro-benchmark experiment, clients send requests of 4 kilobyte size and receive empty replies; Figure 12 reports the corresponding performance and resource-usage



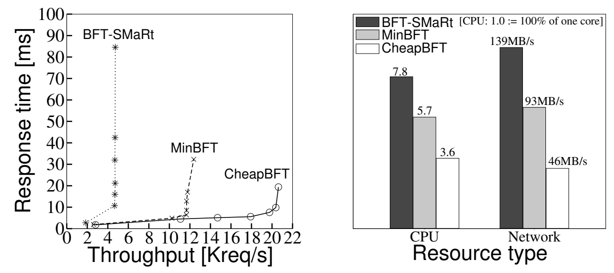
(a) Throughput vs. response time for an increasing number of clients. (b) Average resource usage per 10 Kreq/s normalized by throughput.

Figure 10. Performance and resource-usage results for a micro benchmark with empty requests and empty replies.



(a) Throughput vs. response time for an increasing number of clients. (b) Average resource usage per 10 Kreq/s normalized by throughput.

Figure 11. Performance and resource-usage results for a micro benchmark with empty requests and 4 kilobyte replies.



(a) Throughput vs. response time for an increasing number of clients. (b) Average resource usage per 10 Kreq/s normalized by throughput.

Figure 12. Performance and resource-usage results for a micro benchmark with 4 kilobyte requests and empty replies.

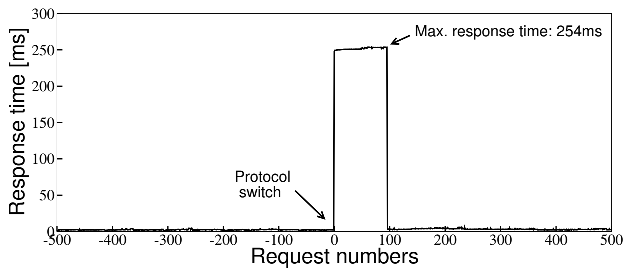


Figure 13. Response time development of CheapBFT during a protocol switch from CheapTiny to MinBFT.

results for this experiment. For such a workload, transmitting requests to active replicas is the decisive factor influencing both performance and resource consumption. With the size of requests being much larger than the size of other protocol messages exchanged between replicas, compared to BFT-SMaRt, CheapBFT replicas need to send 67% less data over the network (50% less data compared to MinBFT). In addition, CheapBFT consumes 54% less CPU than BFT-SMaRt and 37% less CPU than MinBFT.

7.2 Protocol Switch

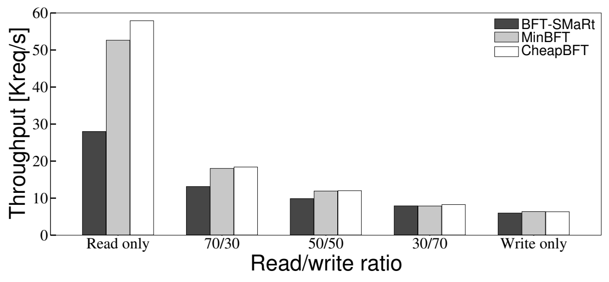
To evaluate the impact of a fault on the performance of CheapBFT, we execute a protocol switch from CheapTiny to MinBFT during a micro benchmark run with 100 clients; the checkpoint interval is set to 200 requests. In this experiment, we trigger the protocol switch shortly before a checkpoint becomes stable in CheapTiny to evaluate the worst-case overhead caused by an abort history of maximum size. Figure 13 shows the response times of 1,000 requests handled by CheapBFT around the time the replicas run the CheapSwitch transition protocol. While verifying and processing the abort history, replicas are not able to execute requests, which leads to a temporary service disruption of max. 254 milliseconds. After the protocol switch is complete, the response times drop back to the normal level for MinBFT.

7.3 ZooKeeper Use Case

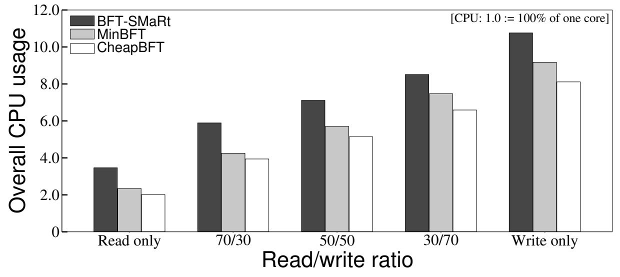
ZooKeeper [16] is a crash-tolerant coordination service used in large-scale distributed systems for crucial tasks like leader election, synchronization, and failure detection. This section presents an evaluation of a ZooKeeper-like BFT service that rely on BFT-SMaRt, MinBFT, and CheapBFT for fault-tolerant request dissemination, respectively.

ZooKeeper allows clients to store and retrieve (usually small) chunks of information in data nodes, which are managed in a hierarchical tree structure. We evaluate the three implementations for different mixes of read and write operations. In all cases, 1,000 clients repeatedly access different data nodes, reading and writing data chunks of random sizes between one byte and two kilobytes. Figure 14 presents the performance and resource-usage results for this experiment.

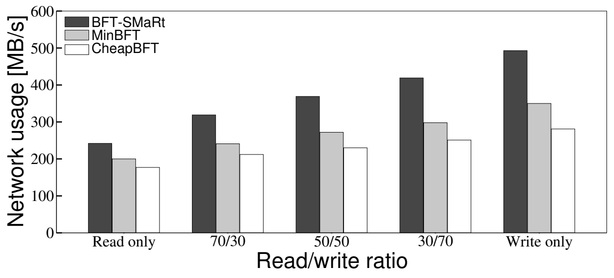
The results show that with the execution stage (i. e., the ZooKeeper application) performing actual work (and not just sending replies as in the micro-benchmark experiments of Section 7.1), the impact of the agreement protocol on system performance is reduced. In consequence, all three ZooKeeper implementations provide similar throughput for write-heavy workloads. However, the resource footprints significantly differ between variants: in comparison to the MinBFT-based ZooKeeper, the replicas in the CheapBFT-based variant save 7-12% CPU and send 12-20% less data over the network. Compared to the BFT-SMaRt implementation, the resource savings of the CheapBFT-based ZooKeeper add up to 23-42% (CPU) and 27-43% (network).



(a) Realized throughput for 1,000 clients.



(b) CPU usage per 10 Kreq/s normalized by throughput.



(c) Network transfer volume per 10 Kreq/s normalized by throughput.

Figure 14. Performance and resource-usage results for different BFT variants of our ZooKeeper service for workloads comprising different mixes of read and write operations.

8. Discussion

As described in Section 5.1, the first PANIC received by a replica triggers the abort of the CheapTiny protocol. In consequence, a single faulty client is able to force a protocol switch, even if all replicas are correct and the network delivers messages in time. In general, we expect such faulty behavior to be rare, as only authenticated clients get access to the system (see Section 3.1). Nevertheless, if an authenticated client repeatedly panics, human intervention may be necessary to revoke the access permissions of the client. However, even if it takes some time to remove the client from the system, unnecessary switches to the MinBFT protocol only increase the resource consumption of CheapBFT but do not compromise safety.

Having completed the CheapSwitch transition protocol, all non-faulty replicas are in a consistent state. Following this, the default procedure in CheapBFT is to run the MinBFT protocol for a certain number of requests before switching back to CheapTiny (see Section 6.2). The rationale of this approach is to handle temporary faults and/or short periods of asynchrony which usually affect only a number of subsequent requests. Note that in case such situations are not characteristic for the particular use-case scenario, different strategies of how to remedy them may be applied. In fact, if faults are typically limited to single requests, for example, it might even make sense to directly start a new instance of CheapTiny after CheapSwitch has been completed.

CheapTiny has a low resource footprint, however, the resource usage is asymmetrically distributed over active and passive replicas. Accordingly, the active replicas, especially the leader, can turn into a bottleneck under high load. This issue can be solved by dynamically alternating the leader role between the active replicas similar to Aardvark [7] and Spinning [29]. Furthermore, one could dynamically assign the role of passive and active replicas thereby distributing the load of agreement and execution over all nodes.

9. Related Work

Reducing the 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 $3f+1$ nodes that actually run agreement as well as execution stage [14, 18]. Note that this is the same as in the pioneering work of Castro and Liskov [4]. The high resource demand of BFT was first addressed by Yin et al. [34] with their separation of agreement and execution that enables the system to run on only $2f + 1$ execution nodes. In a next step, systems were subdivided in trusted and untrusted components for preventing equivocation; based on a trusted subsystem, these protocols need only $2f + 1$ replicas during the agreement and execution stages [5, 8, 23]. The trusted subsystems may become as large as a complete virtual machine and its virtualization layer [8, 23], or may be as small as the trusted counter abstraction [30, 31].

Subsequently, Wood et al. [33] presented ZZ, a system that constrains the execution component to $f + 1$ nodes and starts new replicas on demand. However, it requires $3f + 1$ nodes for the agreement task and relies on a trusted hypervisor and a machine-management system. In a previous work, we increased throughput by partitioning request execution among replicas [9]. Here, a system relies on a selector component that is co-located with each replica, and no additional trust assumptions are imposed. Moreover, we introduced passive execution nodes in SPARE [10]; these nodes passively obtain state updates and can be activated rapidly. The system uses a trusted group communication, a virtualization layer, and reliable means to detect node crashes. Of all these works, CheapBFT is the first BFT system that limits the execution *and* agreement components for all requests to only $f + 1$ replicas, whereas only f passive replicas witness progress during normal-case operation. Furthermore, it relies only on a lightweight trusted counter abstraction.

The idea of witnesses has mainly been explored in the context of the fail-stop fault model so far [22]. In this regard, CheapBFT is conceptually related to the Cheap Paxos protocol [20], 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.

Related to our approach, Guerraoui et al. [14] have proposed to optimistically employ a very efficient but less robust protocol and to resort to a more resilient algorithm if needed. CheapBFT builds on this work and is the first to exploit this approach for changing the number of nodes actively involved (rather than only for changing the protocol), with the goal of reducing the system's resource demand.

PeerReview [15] omits replication at all by enabling accountability. It needs a sufficient number of witnesses for discovering actions of faulty nodes and, more importantly, may detect faults only *after* they have occurred. This is an interesting and orthogonal approach to ours, which aims at tolerating faults. Several other recent works aim at verifying services and computations provided by a single, potentially faulty entity, ranging from database executions [32] and storage integrity [25] to group collaboration [13].

10. Conclusion

CheapBFT is the first Byzantine fault-tolerant system to use $f + 1$ active replicas for both agreement and execution during normal-case operation. As a result, it offers resource savings compared with traditional BFT systems. In case of suspected or detected faults, replicas run a transition protocol that safely brings all non-faulty replicas into a consistent state and allows the system to switch to a more resilient agreement protocol. CheapBFT relies on the CASH subsystem for message authentication and verification, which advances the state of the art by achieving high performance while comprising a small trusted computing base.

Acknowledgments

We thank the anonymous reviewers for their comments and our shepherd, Eric Jul, for his guidance. Furthermore, we are grateful to Michael Gernoth and Christian Spann for technical support, and to Alysson Bessani, Allen Clement as well as Marco Vukolić for interesting discussions and helpful comments on drafts of the paper.

References

[1] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: Virtualizing the trusted platform module. In *Proceedings of the 15th USENIX Security Symposium*, pages 305–320, 2006.

[2] BFT-SMaRt. `http://code.google.com/p/bft-smart/`.

[3] C. Cachin. Distributing trust on the Internet. In *Proceedings of the Conference on Dependable Systems and Networks*, pages 183–192, 2001.

[4] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[5] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of 21st Symposium on Operating Systems Principles*, pages 189–204, 2007.

[6] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. BFT: The time is now. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 1–4, 2008.

[7] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation*, pages 153–168, 2009.

[8] M. Correia, N. F. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd Symposium on Reliable Distributed Systems*, pages 174–183, 2004.

[9] T. Distler and R. Kapitza. Increasing performance in Byzantine fault-tolerant systems with on-demand replica consistency. In *Proceedings of the 6th EuroSys Conference*, pages 91–105, 2011.

[10] T. Distler, R. Kapitza, I. Popov, H. P. Reiser, and W. Schröder-Preikschat. SPARE: Replicas on hold. In *Proceedings of the 18th Network and Distributed System Security Symposium*, pages 407–420, 2011.

[11] T. Eisenbarth, T. Güneysu, C. Paar, A.-R. Sadeghi, D. Schellekens, and M. Wolf. Reconfigurable trusted computing in hardware. In *Proceedings of the 2007 Workshop on Scalable Trusted Computing*, pages 15–20, 2007.

[12] P. England and J. Loeser. Para-virtualized TPM sharing. In *Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies*, pages 119–132, 2008.

[13] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, pages 337–350, 2010.

[14] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. In *Proceedings of the 5th EuroSys Conference*, pages 363–376, 2010.

[15] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the 21st Symposium on Operating Systems Principles*, pages 175–188, 2007.

[16] 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*, pages 145–158, 2010.

[17] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *Proceedings of the 2004 Conference on Dependable Systems and Networks*, pages 575–584, 2004.

[18] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):1–39, 2009.

[19] P. Kuznetsov and R. Rodrigues. BFTW3: Why? When? Where? Workshop on the theory and practice of Byzantine fault tolerance. *SIGACT News*, 40(4):82–86, 2009.

[20] L. Lamport and M. Massa. Cheap Paxos. In *Proceedings of the Conference on Dependable Systems and Networks*, pages 307–314, 2004.

[21] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation*, pages 1–14, 2009.

[22] J.-F. Paris. Voting with witnesses: A consistency scheme for replicated files. In *Proceedings of the 6th Int'l Conference on Distributed Computing Systems*, pages 606–612, 1986.

[23] H. P. Reiser and R. Kapitza. Hypervisor-based efficient proactive recovery. In *Proceedings of the 26th Symposium on Reliable Distributed Systems*, pages 83–92, 2007.

[24] F. B. Schneider and L. Zhou. Implementing trustworthy services using replicated state machines. *IEEE Security & Privacy Magazine*, 3:34–43, 2005.

[25] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *Proceedings of the 2010 Workshop on Cloud Computing Security*, pages 19–30, 2010.

[26] U. Steinberg and B. Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th EuroSys Conference*, pages 209–222, 2010.

[27] E. Stott, P. Sedcole, and P. Cheung. Fault tolerance and reliability in field-programmable gate arrays. *IET Computers & Digital Techniques*, 4(3):196–210, 2010.

[28] M. Strasser and H. Stamer. A software-based trusted platform module emulator. In *Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies*, pages 33–47, 2008.

[29] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proceedings of the 28th Symposium on Reliable Distributed Systems*, pages 135–144, 2009.

[30] 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*, pages 10–19, 2010.

[31] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo. Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, 2011.

[32] P. Williams, R. Sion, and D. Shasha. The blind stone tablet: Outsourcing durability to untrusted parties. In *Proceedings of the 16th Network and Distributed System Security Symposium*, 2009.

[33] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. ZZ and the art of practical BFT execution. In *Proceedings of the 6th EuroSys Conference*, pages 123–138, 2011.

[34] 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*, pages 253–267, 2003.