Parsimony-Based Approach for Obtaining Resource-Efficient and Trustworthy Execution ⋆

HariGovind V. Ramasamy, Adnan Agbaria, and William H. Sanders

Coordinated Science Laboratory, University of Illinois at Urbana-Champaign,
1308 W. Main Street, Urbana, IL 61801, USA
{ramasamy,adnan,whs}@crhc.uiuc.edu

Abstract. We propose a resource-efficient way to execute requests in Byzantine-fault-tolerant replication that is particularly well-suited for services in which request processing is resource-intensive. Previous efforts took a failure-masking *all-active* approach of using all $2t+1$ execution replicas to execute all requests, where t is the maximum number of failures tolerated. We describe an asynchronous execution protocol that combines failure masking with imperfect failure detection and checkpointing. Our protocol is parsimony-based since it uses only $t+1$ execution replicas, called the primary committee or *pc*, to execute the requests normally. Under normal conditions, characterized by a stable network and no misbehavior by *pc* replicas, our approach enables a trustworthy reply to be obtained with the same latency as in the all-active approach, but with only about half of the overall resource use of the all-active approach. However, a request that exposes faults among the *pc* replicas will incur a higher latency than the all-active approach mainly due to fault detection latency. Under such conditions, the protocol switches to a recovery mode, in which all $2t+1$ replicas execute the request and send their replies. Then, after selecting a new *pc*, the request latency returns to the same level as that of all-active execution. Practical observations point to the fact that failures and instability are the exception rather than the norm. That motivated our decision to optimize resource efficiency for the common case, even if it means paying a slightly higher performance cost during periods of instability.

1 Introduction

The trustworthiness of a networked information system (NIS) is judged by its ability to provide security and fault tolerance despite software errors, operator errors, and malicious attacks [1]. Since it is difficult to constrain the behavior of a compromised node that is under the control of an adversary, the Byzantine failure model is an attractive way to model such behavior. By using redundancy to mask the effects of up to a threshold number of security-compromised or failed nodes, Byzantine fault tolerance (BFT) is a promising approach to enhance

⋆ This material is based upon work supported in part by the National Science Foundation under Grant No. CNS-0406351 and DARPA contract F30602-00-C-0172.

the trustworthiness of NISs. In BFT replication, the replicas of a service run deterministic state machines [2] and execute client requests in the same order to ensure state consistency. Execution of requests is preceded by an agreement among the replicas on the request-delivery order using Byzantine agreement (or, equivalently, atomic broadcast).

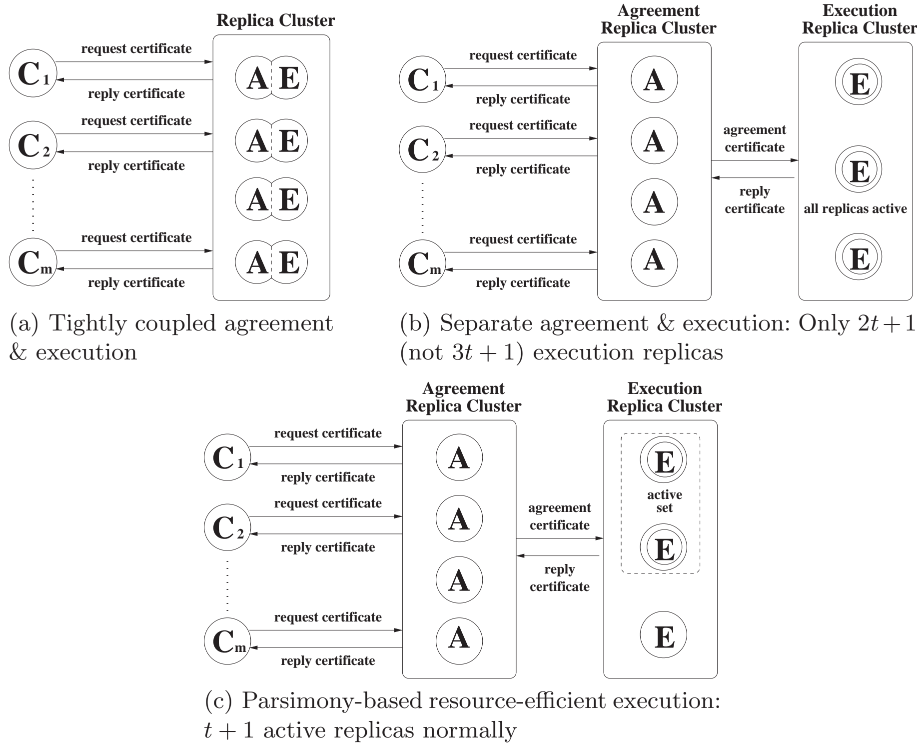
While Byzantine fault tolerance (BFT) as an area has existed for more than two decades, much of the earlier work had significant but mainly theoretical implications. More recent work has focused on removing the barriers that limit the widespread use of BFT to improve security and reliability.

Castro and Liskov's BFT library [3] showed that BFT replication systems can be built that add only modest extra latencies relative to unreplicated systems. They also showed that *proactive recovery* can be used to significantly increase the coverage of the assumption that there are at most a threshold number (one-third) of replicas that can be corrupted by the adversary.

A drawback of BFT replication that limited its applicability in many real-world settings was the requirement that all replicas should run the same service implementation and update their states deterministically. If all replicas ran the same service implementation, then an adversary could exploit the same vulnerabilities or software bugs to cause all replicas to fail simultaneously. The determinism requirement is non-trivial to satisfy in many real-world services. Rodrigues et al. [4] proposed an extension of the BFT library called BASE, which uses abstraction to address that drawback. Specifically, BASE enables the use of diverse COTS-based replica implementations, thereby reducing the possibility of common-mode failures. Their technique uses wrappers to ensure that diverse and non-deterministic implementations of the replicas of a service satisfy a common abstract specification.

Yin et al. [5] improved BASE by enforcing a clean separation between agreement on the request delivery order and execution of requests in the agreed-upon order. Figure 1(b) gives a high-level view of the separation, and contrasts it with traditional BFT (Fig. 1(a)), which tightly couples agreement and execution. While the *agreement cluster* has the usual $3t + 1$ replicas (we call them *agreement replicas*), the separation allowed the number of replicas in the *execution cluster* (we call them *execution replicas*) to be decreased from $3t + 1$ to $2t+1$, where t is the number of simultaneous replica faults that have to be tolerated. The separation also opened up the possibility of including a privacy firewall between the two phases that could be used to enhance confidentiality by preventing a malicious replica in the execution cluster from disclosing unauthorized information to users.

This paper proposes a resource-efficient way to execute requests in BFT replication that is particularly well-suited for services in which request execution is resource-intensive (e.g., computation-intensive). The previous best way was the one proposed by Yin et al. that used $2t+1$ execution replicas. Previous work followed an *all-active* approach (Figures 1(a) and (b)), in which all execution replicas executed the request. We observe that while $2t + 1$ execution replicas is the minimum number of replicas needed to mask t corrupt ones, the client



(a) Tightly coupled agreement & execution

(b) Separate agreement & execution: Only $2t+1$ (not $3t+1$) execution replicas

(c) Parsimony-based resource-efficient execution: $t+1$ active replicas normally

Fig. 1. Successive Steps for Obtaining Efficient Execution in BFT Replication

needs only a set of $t + 1$ identical replies (we call this set the *reply certificate*) before considering the reply to be trustworthy. The reason is that identical replies from $t + 1$ execution replicas will always include the reply from at least one correct replica. Hence, that reply value must be correct. We leveraged the above observation and designed an optimistic protocol for the execution cluster replicas. The protocol is parsimony-based since normally only a fraction of the available resources (i.e., $t + 1$ out of $2t + 1$ replicas) are used for request execution.

Our protocol is based on the *optimistic hope* [6] that normally the network is well-behaved and a designated set of $t + 1$ replicas function properly. When the optimistic hope is satisfied, reply certificates are obtained with the same latency, but with only about half of the overall resource use of the all-active approach. *Overall resource use* is the average resource use at a replica times the number of replicas.

The approach does have a price: in situations when the optimistic hope is not satisfied, the latency for obtaining the reply certificate is higher than it is in the all-active approach due to failure-detection latency. However, even under such situations, our protocol guarantees safety and liveness, subject only to the condition that messages are delivered eventually. Even in NISs that are high-

value attack targets, such situations are expected to be rare. Hence, it makes sense to optimize for the common case, and be prepared for the rare situations in which a higher price may be paid.

The rest of this paper is organized as follows. Section 2 presents the system model and assumptions. Section 3 describes an abstraction of the agreement cluster that simplifies our protocol presentation. Section 4 presents our parsimony-based execution protocol in detail. Section 5 specifies the properties that our protocol is expected to satisfy. We have implemented our protocol and evaluated its performance through fault injection experiments, the results of which are described in Sect. 6. Section 7 lists some practical applications for our protocol and compares it with related work. Finally, Sect. 8 presents our conclusions.

2 System Model

We consider an asynchronous distributed system model in which nodes may operate at arbitrarily different speeds. Every pair of nodes is connected by a *secure asynchronous channel* that provides authenticity. Authenticity can be easily ensured using cheap message authentication codes (MACs) [7]. Asynchronous channels mean that there are no *a priori* bounds on message transmission delays.

The BFT-replicated service consists of n_a replicas forming an agreement cluster and n_e replicas forming an execution cluster. Agreement cluster replicas and execution cluster replicas may occupy different nodes (i.e., there is a physical separation between agreement and execution) or may share nodes (i.e., there is only logical separation between agreement and execution). Clients and replicas occupy different nodes.

Figure 1 shows the dataflow from the clients to the replicated service and back. Clients send authenticated *request certificates* to the agreement cluster replicas. The request certificates will carry some validating information showing that the clients do have the privilege to issue the requested operations. The agreement cluster replicas run a Byzantine agreement or BFT atomic broadcast protocol (e.g., Castro-Liskov's BFT protocol [3] or Cachin et al.'s atomic broadcast protocol [8]) to agree on the order of request execution. The agreed-upon order is conveyed to the execution cluster replicas through *agreement certificates* that show that a sufficient number of agreement cluster replicas approved the order. The execution cluster replicas start with the same initial service state and implement deterministic state machines; they convey the result of executing the requests through reply certificates that contain evidence showing that the result is indeed correct. The reply certificates are sent to the agreement cluster replicas, which then forward the reply certificates to the client.

A computationally bounded adversary controls up to t agreement cluster replicas and up to t execution cluster replicas. We call the replicas controlled by the adversary *corrupt*; other replicas are *correct*. Corrupt replicas may behave in an arbitrary (i.e., Byzantine) manner. Further, it is well-known that to mask t faults, the minimum number of agreement cluster replicas needed is $3t + 1$

and the minimum number of execution cluster replicas needed is $2t + 1$. Thus, $n_a \geq 3t + 1$ and $n_e \geq 2t + 1$. Figure 1(c) depicts the situation where $t = 1$, $n_a = 4$, and $n_e = 3$.

The adversary controls the network and determines the scheduling of messages on all the channels. Timeouts are messages a replica sends to itself; hence, the adversary controls the timeouts as well. However, the parsimony-based execution protocol's properties are guaranteed only to the extent that messages exchanged between correct replicas are eventually delivered without any change in contents.

Besides MAC-authentication for implementing secure channels, we also use public-key signatures [7]. A recipient of a signed message that is convinced of the message's authenticity can convince a third party about the message's authenticity. However, MAC-authentication is not provable to a third party. For the public key signature scheme, each replica possesses a public key, private key pair. The public key of a replica is known to all other replicas. We assume that the signature scheme is secure in the sense of the standard security notion for signature schemes of modern cryptography, i.e., existential forgery against chosen-message attacks [9].

3 The Agreement Cluster Abstraction

In the description of the parsimony-based execution protocol, we consider the agreement cluster as an abstract service that guarantees certain properties relating to the ordering of client requests. We use $\mathcal{AC}$ to denote that service. Abstracting the n_a agreement cluster replicas as one logical entity allows us to keep the focus on the execution cluster replicas with whose behavior the parsimony-based execution protocol is concerned. The functionality provided by $\mathcal{AC}$ is the binding of sequence numbers (starting from 1 and without gaps) to request certificates, and the conveying of the bindings to the execution cluster through agreement certificate messages. The $\mathcal{AC}$ does not require any information about what execution cluster replicas constitute the *pc* and sends the agreement certificate messages to all the replicas. An agreement certificate message binds a sequence number s to a client's request certificate. In our protocol description, the message has the form $(\texttt{agree}, s, o, \mathit{flag})$, where the retransmit *flag* is either `true` or `false`. For notational simplicity, we include only the service operation o contained in the client's request certificate rather than the full certificate. First, $\mathcal{AC}$ sends an `agree` message with the *flag* value `false`. If $\mathcal{AC}$ does not receive a reply certificate before a timeout, then it retransmits the `agree` message with the *flag* value `true`. We use the term *first-time* `agree`(s) to denote the `agree` message with sequence number s and *flag* value `false`. We use the term *retransmit* `agree`(s) to denote the `agree` message with sequence number s and *flag* value `true`.

The $\mathcal{AC}$ provides the following guarantees to the execution cluster replicas:

Agreement: If a correct execution replica receives an agreement certificate binding sequence number s to request certificate rc, then no other correct ex-

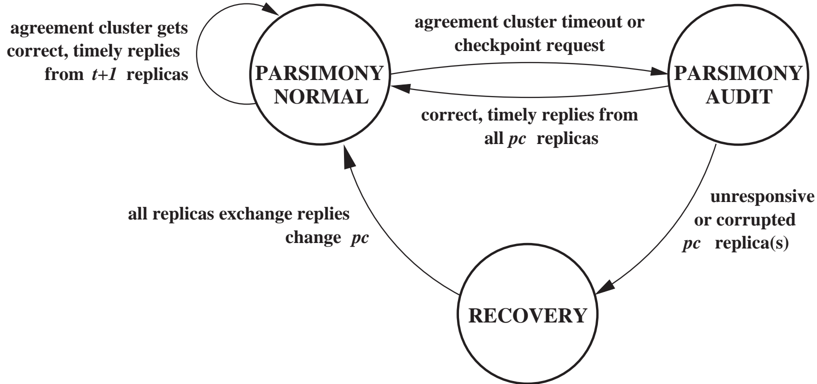


Fig. 2. The Three Modes of Protocol Operation

ecution cluster replica receives an agreement certificate binding s to another request certificate rc', where $rc' \neq rc$.

Liveness: If a client sends a request certificate r to $\mathcal{AC}$, then all correct execution cluster replicas eventually receive an agreement certificate binding some sequence number s to rc.

The above properties of the $\mathcal{AC}$ allow the BFT-replicated service to tolerate an arbitrary number of corrupt clients: even if corrupt clients' requests are executed, those clients cannot cause the service states of correct execution cluster replicas to become inconsistent. Client access control and request-filtering policies [3, 5] can be enforced in the implementation of $\mathcal{AC}$; the policies can effectively limit the number and scope of requests from corrupt clients.

4 The Protocol

To generate a reply certificate for one client request, the execution cluster replicas may go through at most three modes of protocol operation, as shown in Fig. 2. We now describe the protocol's operation in each of those three modes and the triggers that cause the transitions among the modes.

In the following description, we use $E_1, E_2, \ldots, E_{n_e}$ to denote the execution cluster replicas. The *rank* of replica E_i is i. $\langle m \rangle_{\sigma_i}$ is used to denote a message m signed by replica E_i. E_i maintains a local sequence number variable s, where $s - 1$ indicates the highest sequence number for which E_i is sure that $\mathcal{AC}$ has obtained a reply certificate.

Each replica maintains two sets, *slow* and *corrupt*, initialized to empty sets. The $(t + 1)$ lowest-ranked replicas that are neither in *slow* nor in *corrupt* constitute what we call a *primary committee*, or *pc* for short. We call the t non-*pc* execution replicas *backups*. Hence, initially, the primary committee at all replicas consists of the $(t + 1)$ lowest-ranked replicas, namely $\{E_1, E_2, \ldots, E_{t+1}\}$. If two

replicas have the same *slow* and *corrupt* sets, then their respective primary committees will also be the same. For example, if $t = 3$, then the primary committee at all replicas would initially be $\{E_1, E_2, E_3, E_4\}$. If replica E_2 was later added to replica E_4's *slow* or *corrupt* set, then the primary committee at replica E_4 would become $\{E_1, E_3, E_4, E_5\}$.

To simplify the description of the parsimony-based execution protocol, we assume that $\mathcal{AC}$ sends its next request after receiving the reply certificate for its previous request, i.e., the $\mathcal{AC}$ has only one outstanding request. It is easy to extend the protocol to the case where $\mathcal{AC}$ has any fixed constant number of outstanding requests.

4.1 Parsimony-Based Normal Mode

When E_i receives a *first-time* $\texttt{agree}(s+1)$ message, the protocol at E_i moves to the parsimony-based normal mode. E_i maintains a queue of requested operations called *requests*, and adds the service operation indicated in the `agree` message to the queue. Because of our assumption that $\mathcal{AC}$ has at most one outstanding request, E_i can be sure that $\mathcal{AC}$ has obtained a reply certificate for $\texttt{agree}(s)$ when it receives the *first-time* $\texttt{agree}(s + 1)$ message. Hence, E_i increments its sequence number variable s.

If $E_i \notin pc$, then it does nothing more in this mode. On the other hand, if $E_i \in pc$, then it executes the service operation indicated in the `agree` message, and sends the result r of the execution to the $\mathcal{AC}$ in a `reply` message of the form $(\texttt{reply}, i, s, r)$. E_i also adds its `reply` message to *replies*, a data structure that all replicas have to store `reply` messages from themselves and other replicas. Since the replicated state machines are deterministic and the request execution is done in sequence number order, the r values in the `reply` messages sent by all correct replicas will be identical.

In the normal case, the `reply` messages from the *pc* replicas will be sufficient for the $\mathcal{AC}$ to obtain a reply certificate, which it then forwards to the respective client. $\mathcal{AC}$ can then issue the `agree` message with the next sequence number $s + 1$.

Transition from Normal to Audit Mode. The protocol at E_i transitions to the parsimony-based audit mode from the parsimony-based normal mode when

1. E_i receives a *retransmit* $\texttt{agree}(s)$ message from $\mathcal{AC}$ and thereby learns that $\mathcal{AC}$ did not get a reply certificate for $\texttt{agree}(s)$ in a timely manner, or
2. E_i receives a *checkpoint request.*

A checkpoint request is a message of the form $(\texttt{agree}, s + 1, o, \texttt{true})$, where $s+1$ is divisible by the checkpoint interval δ. After every $\delta - 1$ `agree` messages, $\mathcal{AC}$ generates a special `agree` message in which the requested operation o is a *checkpoint* operation[1]. When E_i receives the checkpoint request, E_i knows that

[1] Alternatively, execution cluster replicas can *self-issue* a checkpoint request after δ requests from $\mathcal{AC}$.

$\mathcal{AC}$ must have received a reply certificate for `agree`(s), and hence increments s. Executing a checkpoint operation involves taking a snapshot of the replicated service states and computing the digest of the snapshot. The result field r of the `reply` message for a checkpoint request will contain the checkpoint digest. If E_i has obtained a reply certificate for a checkpoint request with sequence number s, we say that the $(s/\delta)^{\text{th}}$ checkpoint is *stable* at E_i. Checkpointing, as will be shown later, is useful for the efficient update of a backup's state when it has to switch to the recovery mode. Checkpointing also allows the garbage collection of `reply` and `agree` messages with sequence numbers less than that of the last stable checkpoint.

4.2 Parsimony-Based Audit Mode

Upon switching to the audit mode, an execution cluster replica E_i starts a timer and expects to obtain a reply certificate before the timer expiry. If progress is not being made, the replicas collectively switch the protocol to the recovery mode, in which all correct replicas generate their own reply messages (if they hadn't done so previously) and ensure that the $\mathcal{AC}$ obtains a reply certificate. If, on the other hand, the replicas indeed receive a reply certificate in a timely manner from the *pc*, they forward the certificate to $\mathcal{AC}$ and the protocol will switch back to the parsimony-based normal mode.

To enable the monitoring of progress, the *pc* replicas are required to send signed `reply` messages to all execution cluster replicas. A *pc* replica E_j retrieves the result value of the `reply` message from the *replies* data structure if it had previously sent a `reply` message to $\mathcal{AC}$ in the parsimony-based normal mode. Otherwise, E_j obtains the result value by executing the operation specified in the corresponding `agree` message from $\mathcal{AC}$.

Transition from Audit to Recovery Mode. An execution cluster replica E_i may not be able to obtain a reply certificate before its local timer expiry for one or both of the following reasons:

1. Slow Replies: A *pc* replica is (deliberately or unintentionally) slow in sending its reply message.
2. Wrong Replies: A *pc* replica did send its reply message, but with the wrong result value.

Slow Replies. If E_i does not receive the `reply` message from a *pc* replica E_k in a timely manner, then E_i sends a signed `suspect` message for E_k to all execution cluster replicas. The `suspect` message has the form $\langle \texttt{suspect}, i, k, s, c\rangle_{\sigma_i}$, where s is the sequence number and c is a variable called the *reset counter*. The reset counter is an artefact of imperfect failure detection and is used to keep track of the number of times the *slow* set is reset to account for that imperfection. We discuss this in detail in Sect. 4.5.

If a correct replica E_j (has received or) later receives E_k's `reply` message with sequence number s, then E_j simply forwards E_k's `reply` message to E_i

upon receiving E_i's `suspect` message. This `reply` forwarding ensures that if at least one correct replica has received E_k's `reply` message for `agree`(s), then all correct replicas will eventually receive E_k's message.

On the other hand, if no correct replica has received E_k's `reply` message for `agree`(s) in a timely fashion (determined by the replicas' respective local timers), then each of the $n-t$ correct replicas will generate a `suspect` message for E_k. E_i keeps track of all the `suspect` messages it receives by storing them in a data structure, *suspects*.

After receiving $\langle \texttt{suspect}, j, k, s, c\rangle_{\sigma_j}$ messages from $n-t$ distinct E_js (possibly including itself), replica E_i adds E_k to its *slow* set. E_i then sends an `indict` message of the form $(\texttt{indict}, k, s, c, \mathit{proof})$ to all replicas, where *proof* contains the signed `suspect` messages. E_i also adds s to a set *mustDo* that is used to keep track of the sequence numbers of those requests that caused the protocol to switch to recovery mode; the set is so named because all replicas, whether *pc* or backup, *must* send their own `reply` messages for those requests. Having added the *pc* replica E_k to its *slow* set, E_i updates its *pc* accordingly. E_i then switches to the recovery mode. Any replica E_j at which $E_k \notin \mathit{slow}$ that receives E_i's `indict` message will add E_k to its *slow* set, send its own similar `indict` message for E_k to all replicas, and switch to the recovery mode.

Wrong Replies. Since the state machines are deterministic and request execution is done in sequence number order, any difference in the result values of `reply` messages from two replicas indicates that at least one of them is corrupted. However, to be able to pinpoint in a provable manner which of those two replicas is corrupt, a reply certificate is needed; any replica whose `reply` message contains a result value different from that in a reply certificate is corrupt. Replica E_i sends an `implicate` message of the form $(\texttt{implicate}, s, \mathit{proof})$ to all replicas, where *proof* contains two or more `reply` messages with differing result values. A recipient E_k of E_i's `implicate` message will not know which of the implicated replicas is actually corrupt, but will be convinced of the need to switch to the recovery phase and add s to the *mustDo* set.

Repeated Transitions from Normal to Audit Mode. A corrupt *pc* replica can cleverly degrade protocol performance by repeatedly refraining from sending a `reply` message to the $\mathcal{AC}$, thereby forcing a transition from the normal to audit mode, while behaving properly in the audit mode. That would result in frequent transitions from the normal to audit mode and back to normal mode, without a change in the *pc*.

The protocol addresses the above problem as follows. If the fraction of requests that resulted in a transition from the normal to audit mode exceeds a fixed threshold, the protocol operates semi-permanently in the audit mode until the next transition to the recovery mode. After the *pc* is changed in the recovery mode, the protocol reverts back to the normal mode.

4.3 Recovery Mode

Only the *pc* replicas send `reply` messages in the normal and audit modes. In the recovery mode, however, backups are also required to send signed `reply` messages to other replicas. Because at least $t + 1$ replicas are correct, the recovery mode guarantees that a reply certificate for $\texttt{agree}(s)$ will eventually be obtained. As in the audit mode, the reply certificate is then forwarded to the $\mathcal{AC}$. The execution replicas then change the *pc*, and switch back to the parsimonious normal mode for the next request.

To send a `reply` message, a backup first has to determine the result value corresponding to the request contained in $\texttt{agree}(s)$. As before, the result is obtained from a reply certificate (if previously received), or otherwise by actual execution of the request. Before executing the operation specified in the $\texttt{agree}(s)$ message, however, a backup E_i has to ensure that its state is up-to-date. For this purpose, all replicas maintain a variable *updated* to keep track of how up-to-date their state is. Only when *updated* becomes equal to $s - 1$ can E_i execute the operation specified in the $\texttt{agree}(s)$ message. Bringing the state up to date may involve two steps:

1. If $\mathit{updated} < \mathit{stable}$ at E_i, where *stable* is the sequence number of E_i's last stable checkpoint, then E_i first obtains the state corresponding to the execution of all requests with sequence numbers up to *stable*. E_i determines the $t + 1$ replicas whose `reply` messages form the reply certificate for $\texttt{agree}(\mathit{stable})$. E_i then requests the state corresponding to that checkpoint by sending a message of the form $(\texttt{state}, \mathit{stable})$ to those $t + 1$ replicas. Since at least one of the replicas is correct, E_i is guaranteed eventually to obtain the state corresponding to that checkpoint. E_i can easily verify whether the state transferred is correct; E_i computes the digest of a copy of the state obtained after it has applied the updates indicated in the state transfer, and then compares the digest with the one present in the certificate for the stable checkpoint. If the two digests are equal, then the state transferred is correct. E_i then changes the value of *updated* to be equal to *stable*.
2. E_i updates its state to reflect the execution of requests with sequence numbers from $\mathit{updated} + 1$ to $s - 1$. To perform the update, E_i retrieves those requests from the `agree` messages stored in the local *requests* queue, and then actually executes those requests.

Computation of checkpoint digests and state transfer can be made efficient through the use of incremental checkpointing techniques described in [3].

Once a reply certificate has been obtained, it is easy to pinpoint which of the previously implicated replicas (if any) are actually corrupt. A correct replica E_i adds such replicas to its local *corrupt* set, and updates its *pc* accordingly. E_i also shares this information with other replicas, by sending a `convict` message to all replicas. The `convict` message has the form $(\texttt{convict}, k, s, \mathit{proof})$, where *proof* contains the reply certificate and replica E_k's `reply` message for $\texttt{agree}(s)$. Once a correct replica has added E_k to its *corrupt* set, it discards any further protocol messages received directly from E_k.

4.4 Primary Committee Changes

At a correct execution cluster replica, any *pc* change is the result of a change in the sets *slow* or *corrupt* and is always accompanied by the sending of `indict` or `convict` messages respectively. Thus, it is not possible for corrupt replicas to force a change in the *pc* when the *pc* replicas are indeed correctly functioning. Those messages contain sufficient proof to convince any other correct execution cluster replica to effect the same change in its own local *slow* or *corrupt* sets. As a result, even though correct replicas may temporarily differ in their perspectives of the primary committee, their perspectives will eventually concur.

4.5 Failure Detection and Its Effect on Protocol Operation

What the parsimony-based audit mode and the recovery mode accomplish when the *pc* is not able to produce a reply certificate is the distributed identification of *pc* replicas that are not functioning properly. The identification is essentially a form of failure detection and is done with the goal of eventually making the *pc* consist only of correct replicas.

In our formal system model, the adversary controls the scheduling of messages and hence the timeouts; thus, the adversary can cause a correctly functioning *pc* replica to be added to the *slow* sets of correct replicas.

Unlike the adversary in our formal model, the network in a real-world setting will not always behave in the worst possible manner. The motivation for an optimistic protocol such as ours is the hope that timer values that are set based on stable network conditions have a high likelihood of being accurate. Such a hope is not unrealistic since practical observations point to the fact that network behavior alternates between long periods of stable conditions and relatively short periods of stability; this indicates that unstable network conditions are the exception rather than the norm. During periods of stability and when the *pc* replicas do not actively misbehave, the optimistic hope will be satisfied and our protocol will provide resource-efficient request execution with roughly the same latency as the all-active approach.

Even if the optimistic hope is not satisfied, our protocol guarantees safety and liveness. Safety mainly relates to replica state consistency. Since replicas always execute a request bound to sequence number s only after a state update that reflects the execution of all lower-sequence-numbered requests, safety is never violated. Liveness, which is the ability to obtain a reply certificate eventually, is also guaranteed; inaccurate failure detection can, at worst, cause correct *pc* replicas to be added to the *slow* sets at correct replicas, but then the protocol will switch to the recovery mode, which guarantees that a reply certificate will be obtained.

Neutralizing the Effect of Inaccurate Failure Detection. Since the adversary corrupts at most t replicas and the only replicas added to the *corrupt* set are those that actually exhibited malicious failures, the *corrupt* set at a correct replica never exceeds t. However, due to inaccurate failure detection, it is

possible that correct replicas will get added to the *slow* set, and subsequently, $|slow \cup corrupt|$ may exceed t. To allow the next *pc* to be chosen, whenever $|slow \cup corrupt| = t+1$, the *slow* set is reset to the empty set, $\emptyset$. A reset counter c is used to keep track of the number of resets. Both `suspect` and `indict` messages carry an indication of the reset counter value. This allows the garbage collection of all `indict` and `suspect` messages with lower reset-counter values, whenever c is incremented.

Since a correct replica E_i sends an `indict` message for each new entry to its local *slow* set and a `convict` message for each new entry to its local *corrupt* set, if E_i encounters a situation in which $|slow \cup corrupt| > t$, then any correct replica E_j will also eventually encounter a situation $|slow \cup corrupt| > t$. Thus, if the reset-counter c at replica E_i is incremented, then eventually all correct replicas will also increment their respective reset-counters to $c+1$.

5 Protocol Properties

Any replication protocol is required to guarantee safety and liveness. Safety is specified by the total order, update integrity, and result integrity properties described below. Liveness is specified by the termination property described below. Parsimony characterizes the resource efficiency obtained under perceived stable conditions, and distinguishes our protocol from the *all-active* approach. Due to space constraints, we omit the proofs here; they can be found in the full version of the paper [10].

Termination: If the $\mathcal{AC}$ sends `agree`(s), it eventually receives a reply certificate.

Total Order: At any two correct execution cluster replicas E_i and E_j, the updates to their internal states due to execution of the request indicated by `agree`(s) are the same.

Update Integrity: Any correct execution cluster replica updates its internal state in response to the request indicated by `agree`(s) at most once, and only if $\mathcal{AC}$ actually sent that message.

Result Integrity: If r is the result value in the reply certificate received by $\mathcal{AC}$ for `agree`(s), then at least one correct execution cluster replica sent a `reply` message with result value r.

Parsimony: A correct execution cluster replica E_i that is not part of the primary committee will execute the request indicated by the `agree`(s) message and then send a corresponding `reply` message to other replicas only if (1) E_i has not yet obtained a reply certificate for the request, and (2) E_i added s to its local *mustDo* set due to a corrupt or slow replica.

6 Experimental Evaluation

We implemented and experimentally evaluated the parsimony-based execution protocol under both fault-free conditions and controlled fault injections. We com-

pare the results for our protocol with those obtained for the all-active execution approach. All implementations were done in C++.

The fact that the execution phase of a BFT-replicated service will be service-specific poses a challenge to obtaining useful results. The resources involved during request processing will be service-specific, and even request-specific. In our experiments, we have tried to account for that fact by varying the range of service-specific parameters, like the resource intensity of request processing. The specific resource type that we emphasized in our experiments is the CPU, but the conclusions we draw are also an indicator of the trends for other resource types (e.g., network bandwidth) that may be involved in request processing. Our intention was to give a flavor of how parsimony-based execution compares with all-active execution for different service types.

We conducted our experiments for execution cluster sizes $n_e = 3$, 5, 7, 9, and 11 that can tolerate $t = 1$, 2, 3, 4, and 5 simultaneous replica faults, respectively. In a real-world setting, the $\mathcal{AC}$ would consist of a set of $3t+1$ agreement replicas; however, to keep the focus on the execution phase of BFT replication, the clients and the agreement cluster replicas were represented by a single $\mathcal{AC}$ process that generated requests and provided the properties given in Sect. 3.

The setup consisted of a testbed of 12 otherwise unloaded machines. Each machine had a single Athlon XP 2400 processor and 512 MB RAM running RedHat Linux 7.2. One machine was devoted to running the $\mathcal{AC}$ process. At most one execution replica ran on the other machines. The computers were connected by a lightly loaded full-duplex 100 Mbps switched Ethernet network. Digital signatures and MACs were generated using 1024-bit RSA and SHA-1 respectively. Each replica maintained about 1 MB of service-specific state, organized into 1 KB blocks and loaded into its main memory at initialization time.

The $\mathcal{AC}$ sends two kinds of requests: *retrieve-compute* requests and *update-compute* requests. Additionally, for the parsimony-based execution protocol, every 200$^{\text{th}}$ $\mathcal{AC}$ request is a checkpoint request. A *retrieve-compute* request specifies a block to be retrieved. A replica performs some computation on the contents of the block, and returns the result in a `reply` message; there is no change to the replica state. An *update-compute* request specifies a block and new contents for the block. A replica updates the specified block with the new contents, performs some computation on the new contents, and returns the result. The argument field of a *retrieve-compute* request is only a few bytes specifying the block number; for an *update-compute* request, the argument field has the size of a block (1 KB). The result field of the `reply` message for either type of request contains the result of the computation and has the size of a block (1 KB). The $\mathcal{AC}$ sends a new request after obtaining a reply certificate for its last request.

6.1 Behavior in Fault-Free Runs

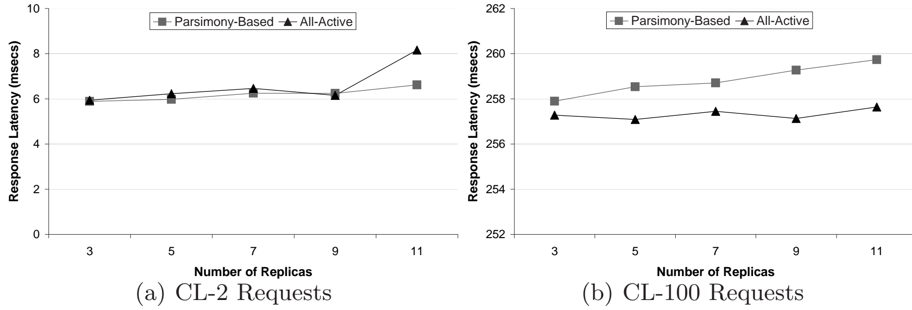
We conducted two sets of experiments that were differentiated by the amount of computation involved in request processing. For the first set of experiments, processing a request involved computation of a public key signature on a specified

block of the service state twice; we call such requests *computation-level 2* or *CL-2* requests. For the second set of experiments, processing a request involved computation of a public key signature on a specified block of the service state 100 times; we call such requests *CL-100* requests. Obviously, one would be hard-pressed to find a real-world application that computes digital signatures 100 times for a request. The intention was to simulate compute-intensive request processing (e.g., an insurance web service that has to solve multi-parameter insurance models to obtain results for auto insurance quotation requests), in which the cost of computing one digital signature (in the audit mode of our protocol) is an insignificant part of the actual request processing overhead.

We measured request latency, which is the time elapsed from when the $\mathcal{AC}$ sends a request until it obtains a reply certificate for the request. Figure 3(a) compares the request latencies of the parsimony-based and the all-active execution approaches for CL-2 requests. Figure 3(b) does the same for CL-100 requests. The latencies were obtained as the average of the last 5,000 values from 20 separate runs, where a run consisted of the $\mathcal{AC}$ sending about 10,000 requests. The $\mathcal{AC}$ generated *retrieve-compute* and *update-compute* requests alternately. The latency for a checkpointing request in parsimony-based execution was amortized among all the requests in the corresponding checkpointing interval.

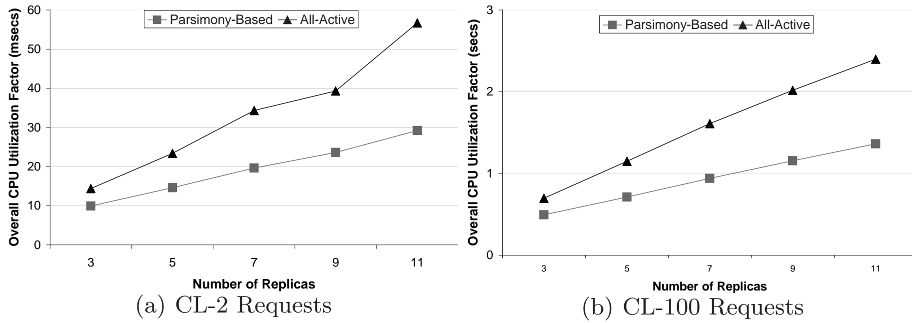
Figures 3(a) and (b) show only a small difference in the request latencies between all-active and parsimony-based execution. For CL-2 requests (Fig. 3(a)), the request latencies for all-active execution are slightly higher than those for parsimony-based execution. The reason is that in all-active execution, even though the $\mathcal{AC}$ needs only $t + 1$ `reply` messages with identical result values to accept the result, it will receive `reply` messages from all replicas (i.e., $2t + 1$ messages), since the runs were fault-free. Though the $\mathcal{AC}$ fully processes only $t + 1$ of those messages and discards the other t, there is overhead involved in receiving the additional unnecessary messages and examining their headers. Thus, one can expect higher latencies for all-active execution if the `reply` message sizes are increased. For compute-intensive CL-100 requests (Fig. 3(b)), the latencies for all-active execution are slightly lower than those for parsimony-based execution. The reason is that all-active execution allows the $\mathcal{AC}$ to choose the fastest $t + 1$ replies among the $2t + 1$ replies that will eventually be received at the $\mathcal{AC}$.

Since in our experiments the CPU is the dominant resource used at a replica in processing $\mathcal{AC}$ requests, we used the UNIX '`ps -aux`' command to measure the percentage of CPU utilization on a replica's host machine that is due to request processing. The CPU utilization percentage at a replica was obtained as the average of samplings made every 5 seconds in each run (a run spanned the time it took to process 10,000 $\mathcal{AC}$ requests). The CPU utilization percentages for *pc* replicas in parsimony-based execution and those for any replicas in all-active execution were roughly the same (in the 75%-85% range for CL-2 requests and in the 85%-95% range for CL-100 requests). The CPU utilization percentages for backups in parsimony-based execution were negligible for both CL-2 and CL-100 requests.



(a) CL-2 Requests (b) CL-100 Requests

Fig. 3. Request Latency



(a) CL-2 Requests (b) CL-100 Requests

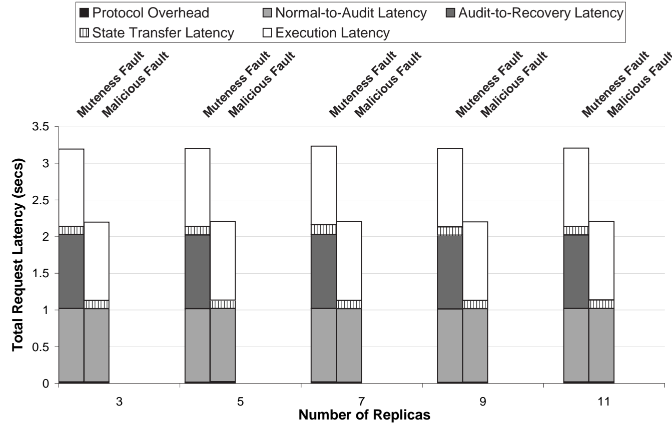
Fig. 4. Overall CPU Utilization Factor Per Request

After obtaining the average CPU utilization percentages at the individual replicas, we computed the *overall CPU utilization factor*, which we obtained by summing over all replicas the product of the CPU utilization percentage and the time taken to process a request. Figures 4(a) and (b) show the overall CPU utilization factor for CL-2 and CL-100 requests. The utilization factor for parsimony-based execution is roughly half of that for all-active execution, and the reduction is more pronounced as the number of replicas increases. This is a practically significant result. For example, in the Application Service Provider (ASP) business model (see Section 7.1), the overall CPU utilization factor would be an indicator of the total amount of CPU resources spent by the ASP servers per request, and could form the basis for pricing, especially if request processing is compute-intensive.

6.2 Behavior in the Presence of Fault Injections

We conducted fault injection experiments on our protocol. We did not fault-inject the implementation of all-active execution, since its behavior in the presence of faults would not be much different from its behavior when there are no faults.

Figure 5(a) shows the different factors that contributed to the $\mathcal{AC}$ request latency when a *pc* member was fault-injected after servicing a sufficiently large



(a) Request Latency when a *pc* Member is Faulty

(b) Settling Latency for Multiple Correlated Faults

Fig. 5. Behavior Under Fault Injections

number of CL-2 requests (about 5,000). We injected both *muteness* faults and *malicious* faults in our protocol. A muteness fault injection was done by crashing a *pc* member upon receipt of a specified $\mathcal{AC}$ request. A malicious fault injection was done by making a *pc* member send wrong values in its `reply` messages.

The highest latency is obtained when a muteness fault injection is done. The latency comprises two timeout values, state update latency, and the protocol-specific overhead. The first timeout value of 1 second (represented by "normal-to-audit latency" in the graph) is used at the $\mathcal{AC}$ before the $\mathcal{AC}$ sends a *retransmit* message for the request. Receipt of that message will cause the protocol to switch from parsimony-based normal mode to parsimony-based audit mode. The second timeout value of 1 second (represented by "audit-to-recovery latency" in the graph) causes a replica to send a `suspect` message for the crashed *pc* member, as the replica would not have received a `reply` message from the *pc* member for

the request. Once $n-t$ `suspect` messages for the crashed *pc* member have been received, the protocol switches from the parsimony-based audit mode to recovery mode. In the recovery mode, backups bring their states up to date in two steps before sending their own `reply` messages for the $\mathcal{AC}$ request. The first step (represented by "state transfer latency" in the graph) is the transfer of state from a correct *pc* member up to the last stable checkpoint. The second step (represented by "execution latency" in the graph) is the actual execution of all the requests after the checkpoint request up to the request for which $\mathcal{AC}$ sent a *retransmit* message. To bring out the worst-case behavior, we injected the muteness fault into a *pc* member upon receiving the request just prior to the checkpoint request (so that $\delta - 1$ requests would actually have to be executed), and all backups requested state transfer from the same correct *pc* member. The portion marked "protocol overhead" in the graph includes a round-trip transmission time from the $\mathcal{AC}$ to the replica (for the request message from the $\mathcal{AC}$ and the `reply` message from the replica) plus other overhead related to the parsimony-based protocol (such as exchange of `suspect` and indict messages, and selection of a new *pc*). We see that an overwhelmingly large portion of the request latency when a muteness fault is injected depends on tunable system parameters (like timeout) and service-specific values, such as the size of the application state, the checkpointing technique used, the number of requests beyond the last stable checkpoint that have to be executed to bring the state up to date, and the normal request processing latency. The actual overhead due to the parsimony-based protocol is less than 20 milliseconds.

The $\mathcal{AC}$ request latencies for malicious fault injection are essentially the $\mathcal{AC}$ request latencies for muteness fault injection minus the timeout value used at replicas (i.e., the audit-to-recovery latency). Fault detection is much faster for malicious faults because it is based on examination of the contents of the `reply` message rather than on timeouts.

Figure 5(b) quantifies the effect that multiple correlated fault injections have at the replicas. After servicing a sufficiently large number of requests (about 5,000), we injected multiple faults at the replicas so that a new *pc* member fault was activated every time an $\mathcal{AC}$ request arrived until the fault resiliency t of the replication group was exhausted. As before, the $\mathcal{AC}$ sent a request only after accepting a result for its previous request. We injected both muteness and malicious faults, and thus there are two rows of bars in the graph. The first fault was activated at a checkpoint request to bring out the worst-case behavior. At each correct replica, we measured the *settling latency*, i.e., the time from when the first fault is detected at a replica until the time when the *pc* consists only of non-fault-injected replicas. The time includes the fault detection latency for $t-1$ faults (i.e., for all faults except the first fault), the state update latency, the time to execute t $\mathcal{AC}$ requests, and the overhead due to the parsimony-based execution protocol. Since the multiple faults are activated at consecutive requests, backups have to bring their state up to date only once, after the first fault detection.

As expected, both rows of bars in the graph show an increase in the settling latency as the number of replicas (and hence the number of fault injections, t)

increases. For a given t, the settling latency for muteness faults is higher than that for malicious faults. The reason is that the fault detection latency for t muteness fault injections includes $2(t-1)$ timeouts (the factor of 2 being due to the $\mathcal{AC}$ timeout plus the timeout at replicas), as opposed to only $(t-1)$ $\mathcal{AC}$ timeouts for t malicious fault injections.

7 Discussion

In this section, we give examples of applications that would benefit from our protocol and compare our protocol with related work.

7.1 Practical Applications

Our protocol can yield significant benefits in many applications. Below are two examples:

1. The web service infrastructures for many companies are no longer operated by the companies themselves, but are outsourced to third parties called *Application Service Providers* or *ASPs*. The ASPs own, operate, and maintain the servers running the applications that provide the companies' web services, saving the companies the cost burden of having to set up specialized information technology infrastructures. The ASPs' servers may be shared among several companies. Usually, an ASP charges an outsourcing company a consumption fee based on the actual resource use. In such a situation, BFT replication can be very useful in enhancing the trustworthiness of computations, and our protocol can be used to obtain significant reductions in overall execution costs and thereby the fee that the outsourcing company has to pay to the ASP. The benefits are especially pronounced if the web service application is resource-intensive. An example of a web service for which request processing is computation-intensive would be a financial web service that has to solve multi-parameter financial models to predict stock trends.

2. In the computational Grid, many services are computation-intensive. BFT replication can be used to obtain a trustworthy system from untrusted participating Grid nodes. Since Grid nodes may be shared among several Grid services, our protocol can help significantly reduce the performance impact that one service has on other services running in the same Grid node.

7.2 Related Work

BFT replication techniques are of two categories: quorum replication and state machine replication. Quorum replication (e.g., [11]) uses subsets of replicas (called *quorums*) to implement read/write operations on the variables of a data repository, such that any two subsets intersect in enough correct replicas. State machine replication can be used to perform arbitrary computations accessing arbitrary numbers of variables; quorum replication is less generic and cannot

handle concurrent requests by clients to update the same information. Our protocol is similar to quorum systems in that it uses a subset of replicas to perform operations. However, the similarity is only superficial; our protocol is concerned with the execution phase of state machine replication, our use of a $(t+1)$-subset of replicas to execute requests is based on whether the system is stable or not (a distinction that quorum systems do not make), and (unlike quorum systems) we do not use different subset sizes for read and write operations.

Our protocol is both unique and novel. While most work on BFT replication has focused on the hard problem of Byzantine agreement (e.g., [3, 12]), our work focuses on the often-overlooked but practically significant execution phase of BFT replication. Yin et al.'s work reduces the *deployment costs* of BFT replication by reducing the number of execution replicas from $3t + 1$ to $2t + 1$. However, our work deals with reducing the *run-time or operational costs* of BFT replication, which are likely to be at least as important as deployment costs in many long-lived and resource-intensive applications. While the parsimony principle has been routinely used in primary-backup systems that tolerate benign faults (e.g., [13]), our protocol is novel in that it is the first to apply parsimony to Byzantine fault tolerance.

Since our protocol is for the execution cluster, our work is complementary to the BASE work [4] and the BASE extension by Yin et al. [5]. In particular, one could combine our protocol with (1) the proactive recovery and abstraction techniques of BASE to overcome the drawbacks of state machine replication in many applications (namely, the determinism requirement and the assumption that at most one-third of the replicas are corrupt), and (2) the privacy firewall architecture of [5] to obtain BFT confidentiality.

In the context of parallel computing, Sarmenta [14] proposed mechanisms for tolerating erroneous results submitted by malicious volunteers in the Grid, SETI@home, and other *volunteer computer systems*. The mechanisms, called *credibility-based fault-tolerance* mechanisms, estimate the credibility of a node and use these probability estimates in limiting the amount of redundant computations necessary to meet desired error rates. However, their scheme trades off correctness for performance and is not relevant to applications that are stateful or cannot tolerate any errors at all (e.g., banking or financial applications). Also, their mechanisms operate in a system and fault model that is very restrictive (e.g., it requires synchronous computations and non-collusion among malicious nodes) and less generic than our system and fault model.

8 Conclusion

We described a protocol for executing requests in a resource-efficient way while providing trustworthy results in the presence of up to t Byzantine faults. Previous best solutions were based on the all-active approach, which requires at least $2t+1$ replicas to execute a request. Our protocol reduces service-specific resource use costs to about half of what they are for all-active execution under perceived normal conditions by using only a *pc* consisting of $t+1$ execution replicas to ex-

ecute the request. The benefits are more pronounced for larger group sizes, and when request processing is resource-intensive. The trade-off for the benefits is the higher latencies during perceived failure or instability conditions due to fault detection and service-specific state update latencies. It is reasonable to expect that a system's operation will alternate between long periods of normal conditions and short periods of instability. That motivated our decision to optimize our protocol for the common case, even if it means paying a slightly higher cost during periods of instability.
Acknowledgments: We thank Christian Cachin, Kaustubh Joshi, and Ryan Lefever for many insightful discussions and valuable suggestions for improving the quality of the paper. We thank Jenny Applequist for her editorial comments.

References

1. Schneider, F.B., ed.: Trust in Cyberspace. National Academy Press (1999)
2. Lamport, L.: Time, Clocks and Ordering of Events in Distributed Systems. Communications of the ACM **21** (1978) 558–565
3. Castro, M., Liskov, B.: Practical Byzantine Fault Tolerance and Proactive Recovery. ACM Transactions on Computer Systems (TOCS) **20** (2002) 398–461
4. Rodrigues, R., Castro, M., Liskov, B.: BASE: Using Abstraction to Improve Fault Tolerance. In: Proceedings of the 18th Symposium on Operating System Principles. (2001) 15–28
5. Yin, J., Martin, J.P., Venkataramani, A., Alvisi, L., Dahlin, M.: Separating Agreement from Execution for Byzantine Fault Tolerant Services. In: Proc. 19th Symp. on Operating Systems Principles. (2003) 253–267
6. Kursawe, K.: Optimistic Byzantine Agreement. In: Proc. 21st Symposium on Reliable Distributed Systems. (2002) 262–267
7. Vanstone, S.A., van Oorschot, P.C., Menezes, A.: Handbook of Applied Cryptography. CRC Press (1996)
8. Cachin, C., Kursawe, K., Petzold, F., Shoup, V.: Secure and Efficient Asynchronous Broadcast Protocols. In: Advances in Cryptology: CRYPTO 2001 (J. Kilian, ed.), LNCS-2139, Springer (2001) 524–541
9. Goldwasser, S., Micali, S., Rivest, R.L.: A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. SIAM Journal on Computing **17** (1988) 281–308
10. Ramasamy, H.V., Agbaria, A., Sanders, W.H.: A Parsimonious Approach for Obtaining Resource-Efficient and Trustworthy Execution. Submitted for publication in the IEEE Transactions on Dependable and Secure Computing (2005)
11. Malkhi, D., Reiter, M.: Byzantine Quorum Systems. Journal of Distributed Computing **11** (1998) 203–213
12. Reiter, M.K.: The Rampart Toolkit for Building High-Integrity Services. In: Selected Papers from the International Workshop on Theory and Practice in Distributed Systems, LNCS 938, Springer-Verlag (1995) 99–110
13. Budhiraja, N., Schneider, F., Toueg, S., Marzullo, K.: The Primary-Backup Approach. In Mullender, S., ed.: Distributed Systems, ACM Press - Addison Wesley (1993) 199–216
14. Sarmenta, L.F.G.: Sabotage-tolerance mechanisms for volunteer computing systems. Future Generation Computer Systems **18** (2002) 561–572