A Parsimonious Approach for Obtaining Resource-Efficient and Trustworthy Execution

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

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 execution replicas to execute all requests; at least $2t+1$ execution replicas are needed to mask t Byzantine faulty ones. We describe an asynchronous protocol that provides resource-efficient execution by combining failure masking with imperfect failure detection and checkpointing. Our protocol is *parsimonious* since it uses only $t+1$ execution replicas, called the primary committee or $\mathcal{PC}$, to execute the requests under normal conditions characterized by a stable network and no misbehavior by $\mathcal{PC}$ replicas; thus, a trustworthy reply can be obtained with the same latency but with only about half of the overall resource use of the all-active approach. However, a request that exposes faults among the $\mathcal{PC}$ replicas will cause the protocol to switch to a recovery mode, in which all $2t+1$ replicas execute the request and send their replies; then, after selecting a new $\mathcal{PC}$, the protocol switches back to parsimonious execution. Such a request will incur a higher latency using our approach than the all-active approach mainly due to fault detection latency. 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.

Index Terms

Distributed systems, fault tolerance, Byzantine faults

I. Introduction

Today, every aspect of our economy is becoming increasingly dependent on networked information systems (NISs). Consequently, it has become crucially important to make these systems trustworthy. The trustworthiness of an NIS is judged by its ability to provide security and fault tolerance despite software errors, operator errors, and malicious attacks [2]. Since it is difficult

The authors are with the University of Illinois at Urbana-Champaign. Email: {ramasamy, adnan, whs}@crhc.uiuc.edu

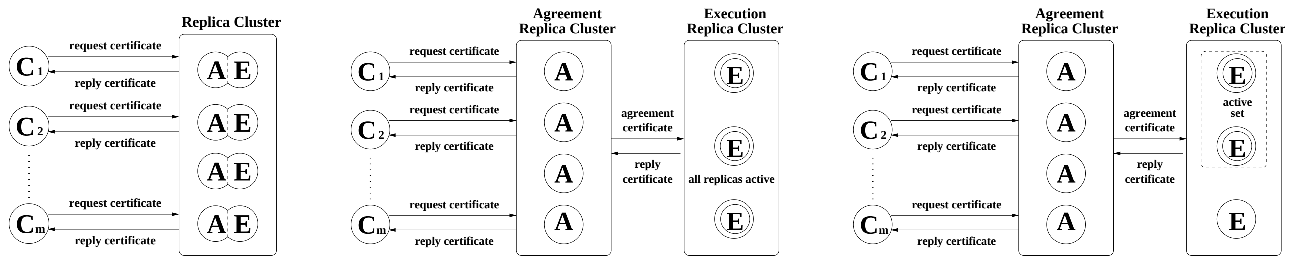
This material is based upon work supported in part by the National Science Foundation under Grant No. CNS-0406351 and by DARPA contract F30602-00-C-0172.

A preliminary version of this paper [1] appears in the Proceedings of the 2^{nd} Latin American Symposium on Dependable Computing (LADC-2005).

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 the trustworthiness of NISs. In BFT replication, the replicas of a service run deterministic state machines [3], [4] 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 BFT has been researched for 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 [5] 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 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. [6] 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. [7] improved BASE by enforcing a clean separation between agreement on the request delivery order and execution of requests in the agreed-upon order. Fig. 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. In order to achieve a fault resilience of t, there must be at least $3t + 1$ distinct participants (we call them *agreement replicas*) in the agreement phase and at least $2t + 1$ distinct participants (we call them *execution replicas*) in the execution phase.



(a) Tightly coupled agreement & execution

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

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

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

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 (Figs. 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 corrupted ones, the client 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 replicas that normally uses only a fraction of the available resources (i.e., $t+1$ out of $2t+1$ replicas) for request execution; hence we call our protocol *parsimonious*.

Our protocol is based on the *optimistic hope* [8] 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: if 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. Another price of our approach (relative to all-active execution) is that it requires execution replicas to execute our parsimonious protocol in addition to the service requests. We argue that in many real-world applications (such as those mentioned in Section VII-B), the savings in overall resource use obtained by our approach justifies this price.

II. System Model

We consider an asynchronous distributed system model equivalent to the one of Cachin et al. [9], in which there are no bounds on relative processing speeds and message delays. The BFT-replicated service consists of n_a replicas participating in the agreement phase and n_e replicas participating in the execution phase. Agreement replicas and execution 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 of the service and replicas occupy different nodes.

Fig. 1(c) shows the dataflow from the clients to the replicated service and back. Clients send authenticated *request certificates* to the agreement replicas. The request certificates will carry some validating information showing that the clients do have the privilege to issue the requested operations. The agreement replicas run a Byzantine agreement or BFT atomic broadcast protocol (e.g., Castro-Liskov's BFT protocol [5] or Cachin et al.'s atomic broadcast protocol [9]) to agree on the order of request execution. The agreed-upon order is conveyed to the execution replicas through *agreement certificates* that show that a sufficient number of agreement replicas approved the order. The execution 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 replicas, which then forward the reply certificates to the client.

A computationally bounded adversary controls up to t agreement replicas and up to t execution 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 replicas needed is $3t + 1$ and the minimum number of execution replicas needed is $2t + 1$ [7]. Thus, $n_a \geq 3t + 1$ and $n_e \geq 2t + 1$. Fig. 1(c) depicts the situation where $t = 1$, $n_a = 4$, and $n_e = 3$.

Every pair of nodes is linked by an *authenticated asynchronous channel* that provides message integrity (e.g., using message authentication codes [10]). The adversary determines the scheduling of messages on all the channels. Timeouts are messages that a party sends to itself; hence, the adversary controls the timeouts as well.

We restrict the adversary such that every run of the system is *complete*, i.e., every message sent by a correct party and addressed to a correct party is delivered unmodified before the adversary terminates. We refer to this property in liveness conditions, when we say that a message is *eventually* delivered or that a protocol instance *eventually* terminates.

A correct party is activated when the adversary delivers a message to the party; the party then updates its internal state, performs some computation, and generates a set of response messages that are given to the adversary. There may be several threads of execution for a given party, but no more than one of them may be active at the same time. When a party is activated, all threads are in *wait states*, which specify a condition defined on the received messages contained in the input buffer, as well as on some local variables. In the pseudocode presentation of the protocol, we specify a wait state using the notation **wait for** *condition*. There is a global implicit **wait for** statement that every protocol instance repeatedly executes; it matches any of the *conditions* given in the clauses of the form **upon** *condition block*. If one or more threads are in a wait state whose condition is satisfied, one of these threads is scheduled (arbitrarily), and this thread runs until it reaches another wait state. This process continues until no more threads are in a wait state whose condition is satisfied. Then, the activation of the party is terminated and control returns to the adversary.

We make use of a digital signature scheme for our protocol. A digital signature scheme consists of algorithms for key generation, signing, and verification. As part of system initialization, the key generation algorithm is invoked to generate the public key/private key pair for each party, and every party is given its private key and the public keys of all parties. 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 [11].

III. The Agreement Phase Abstraction

In the description of the parsimonious execution protocol, we consider the agreement phase as an abstract service that guarantees certain properties related to the ordering of client requests. We use $\mathcal{AC}$ to denote that service. Abstracting the n_a agreement replicas as one logical entity allows us to keep the focus on the execution replicas with whose behavior the parsimonious 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 phase through agreement certificate messages. $\mathcal{AC}$ does not require any information about what execution replicas constitute the $\mathcal{PC}$ 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`.

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

Agreement If a correct execution replica receives an agreement certificate that binds sequence number s to request certificate rc, then no other correct execution replica receives an agreement certificate that binds s to another request certificate rc', where $rc' \neq rc$.

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

The above properties of $\mathcal{AC}$ allow the BFT-replicated service to tolerate an arbitrary number of corrupted clients; even if corrupted clients' requests are executed, those clients cannot the service states of correct execution replicas to become inconsistent. Client access control and request-filtering policies [5], [7] can be enforced in the implementation of $\mathcal{AC}$; the policies can effectively limit the number and scope of requests from corrupted clients.

IV. The Asynchronous Parsimonious Execution (APE) Protocol

We now present Protocol APE, a protocol for the execution replicas that allows for asynchronous parsimonious execution of client requests.

A. Protocol Properties

A protocol for the execution phase satisfies the following properties:

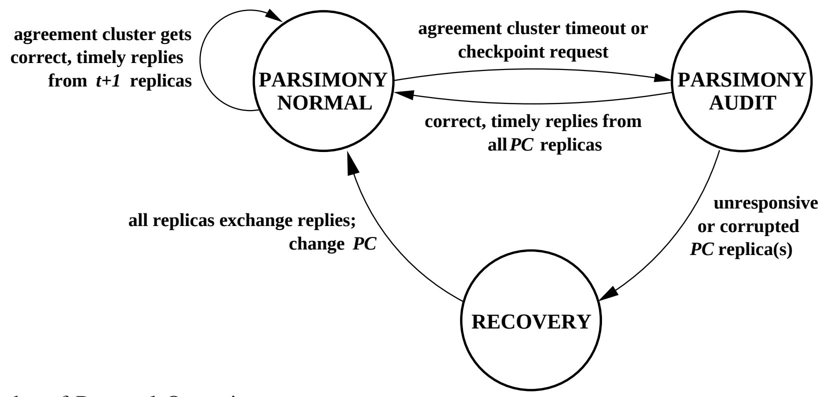

agreement cluster gets correct, timely replies from t+1 replicas
PARSIMONY NORMAL
agreement cluster timeout or checkpoint request
PARSIMONY AUDIT
correct, timely replies from all PC replicas
all replicas exchange replies; change PC
unresponsive or corrupted PC replica(s)
RECOVERY

Fig. 2. The Three Modes of Protocol Operation

Total Order: For any two correct execution replicas E_i and E_j, their internal states just after execution of the request indicated by agree(s) are the same.

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

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 replica sent a reply message for agree(s) with result r.

Liveness: If $\mathcal{AC}$ sends to the execution replicas, it eventually receives a reply a reply certificate for agree(s).

B. Protocol Overview

To generate a reply certificate for one client request, the execution replicas may go through at most three modes of protocol operation: a *parsimonious normal mode*, a *parsimonious audit mode*, and a *recovery mode*. Fig. 2 shows a state transition diagram for the three modes.

Normally, Protocol APE will operate in the parsimonious normal mode, in which only a designated active set of $t + 1$ execution replicas execute requests from the agreement phase and reply directly to the agreement phase (Fig. 1(c)). The $t + 1$ active replicas constitute what we call a *primary committee*, or $\mathcal{PC}$ for short. We call the t non-$\mathcal{PC}$ execution replicas *backups*. There is no need for communication among the execution replicas, and the replies to the agreement phase use cheap message authentication codes (MACs) for authentication. Backups do nothing, aside from receiving the requests from the agreement phase. Non-receipt of a reply certificate at the agreement phase before a timeout will cause the agreement phase to resend the request to the execution phase. That will cause the protocol to transition to the parsimonious audit mode.

The parsimonious audit mode is similar to the parsimonious normal mode, in that backups do not execute the request. However, $\mathcal{PC}$ replicas execute the request (if they haven't already) and send their replies with digital signatures to other execution replicas. The use of digital signatures allows any execution replica that has received a reply certificate to forward the reply certificate to the agreement phase in a verifiable manner. All replicas (whether $\mathcal{PC}$ replicas or backups) monitor the progress made by the $\mathcal{PC}$ replicas; hence the name *audit* mode. If the replicas receive a reply certificate in a timely manner from the $\mathcal{PC}$, the protocol will switch back to the parsimonious normal mode. Otherwise, the replicas can effectively determine whether some $\mathcal{PC}$ replica is blocking progress, and if enough replicas determine so, then the protocol switches to the recovery mode.

In the recovery mode, all replicas execute the request and send their signed replies to other execution replicas (if they haven't already). At backups, the execution of the request will be preceded by updating of the state. Checkpointing is used to guarantee that backups can bring their state up to date. Because all $2t + 1$ execution replicas reply, a reply certificate is guaranteed to be obtained eventually, even if t replicas are faulty. The execution replicas then change the $\mathcal{PC}$ and switch back to the parsimonious normal mode for the next request.

C. Protocol Details

We now describe Protocol APE's operation in each of the three modes and the triggers that cause the transitions among the modes. The line numbers refer to the detailed protocol description in Figs. 3–5. In the following description, we use $E_1, E_2, \ldots, E_{n_e}$ to denote the execution 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.

E_i maintains two sets, $slow$ and $corrupt$, initialized to empty sets. The $\mathcal{PC}$ consists of the $(t + 1)$ lowest-ranked replicas that are neither in $slow$ nor in $corrupt$. 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 $\{E_1, E_3, E_4, E_5\}$.

```
Protocol APE for execution replica $E_i$ with input parameter $\delta$ (checkpoint interval)
  initialization:
    1: $\mathcal{PC} \leftarrow \{E_1, E_2, \ldots, E_{t+1}\}$   {current primary committee}
    2: $corrupt \leftarrow \emptyset$   {set of replicas for which $E_i$ has sent convict messages}
    3: $slow \leftarrow \emptyset$   {set of replicas for which $E_i$ has sent indict messages since last reset}
    4: $c \leftarrow 0$   {counter indicating number of resets of the set $slow$}
    5: $certified \leftarrow \emptyset$   {set of sequence numbers of requests for which $E_i$ has reply certificates}
    6: $\mathcal{R} \leftarrow []$   {array of size $(\delta - 1)$ containing operations requested by $\mathcal{AC}$}
    7: $s \leftarrow 0$   {sequence number in the last agree message received from $\mathcal{AC}$}
    8: $u \leftarrow 0$   {sequence number up to which $E_i$'s state is updated}
    9: $replies \leftarrow \emptyset$   {set of reply messages received from replicas}
    10: $suspects \leftarrow \emptyset$   {set of suspect messages for various replicas received}
    11: $must\_do \leftarrow \emptyset$   {set of sequence numbers of requests that $E_i$ must execute even if $E_i \notin \mathcal{PC}$}
  forever:
    12: wait for receipt of first-time agree($s+1$) or retransmit agree($s$) message from $\mathcal{AC}$
    {Mode 1: Parsimonious Normal Mode}
    13: if message received was first-time agree($s+1$) and $(s+1)\ mod\ \delta \neq 0$ then
    14:     $s \leftarrow s+1$
    15:     $\mathcal{R}[s \bmod \delta] \leftarrow o$, where $o$ is the service operation specified in the agree message
    16:     if $E_i \in \mathcal{PC}$ then
    17:         $r \leftarrow obtain\_reply()$
    18:         $send\_reply(r, \texttt{normal})$
    {Mode 2: Parsimonious Audit Mode}
    19: else {retransmit agree($s$) or first-time agree($s+1$) checkpoint request}
    20:     if message received was first-time agree($s+1$) and $(s+1)\ mod\ \delta = 0$ then {checkpoint}
    21:         $s \leftarrow s+1$
    22:     repeat
    23:         $oldpc \leftarrow \mathcal{PC}$
    24:         if $(E_i \in \mathcal{PC})$ then
    25:             $r \leftarrow obtain\_reply()$
    26:             $send\_reply(r, \texttt{audit})$
    27:         $update_{\mathcal{FD}}(\texttt{start-monitor}, s)$
    28:         wait for $s \in certified$ or $s \in must\_do$ or $oldpc \neq \mathcal{PC}$
    29:         $update_{\mathcal{FD}}(\texttt{stop-monitor}, s)$
    30:     until $s \in certified$ or $s \in must\_do$
    31:     if $(s \in certified)$ then
    32:         send reply certificate for agree($s$) to $\mathcal{AC}$
    {Mode 3: Recovery Mode}
    33:     else {$s \in must\_do$}
    34:         $r \leftarrow obtain\_reply()$
    35:         $send\_reply(r, \texttt{recover})$
  function $send\_reply(r, mode)$:
    36: if $(mode = \texttt{normal})$ then
    37:     send the message $(\texttt{reply}, i, s, r)$ to $\mathcal{AC}$
    38: else
    39:     send the message $\langle \texttt{reply}, i, s, r \rangle_{\sigma_i}$ to all replicas
```

Fig. 3. Protocol APE for Byzantine-Fault-Tolerant Request Execution (Part I)

```
function $obtain\_reply()$:
40: if $\exists\ r$: ($replies$ contains $\langle \texttt{reply}, k, s, r\rangle_{\sigma_k}$ messages from $t+1$ distinct $E_k$ or
        $(\texttt{reply}, i, s, r) \in replies$) then
41:     return $r$
42: else {need to actually execute request to obtain result}
43:     if $u \neq (s-1)$ then
44:         $state\_update()$
45:     if $(s \bmod \delta \neq 0)$ then
46:         $o \leftarrow \mathcal{R}[s \bmod \delta]$
47:         $r \leftarrow execute(o)$
48:     else
49:         $r \leftarrow checkpoint()$                {take checkpoint, and return digest of internal state}
50:     $u \leftarrow s$
51:     $store\_reply((\texttt{reply}, i, s, r), i)$
52:     return $r$
function $state\_update()$:        {update state to reflect execution of requests up to seq. number $s-1$}
53: $stable \leftarrow absolute(s/\delta) * \delta$
54: if $(u < stable)$ then
55:     find digest $r$ : ($replies$ contains $\langle \texttt{reply}, k, stable, r\rangle_{\sigma_k}$ messages from $t+1$ distinct $E_k$)
56:     $certifiers \leftarrow$ set of $t+1$ distinct $E_k$ whose reply messages form reply certificate for $stable$
57:     send the message $(\texttt{state-request}, stable)$ to all $E_k \in certifiers$
58:     wait for receipt of state updates such that digest of internal state after applying them equals $r$
59: for $u = (max(stable, u)+1), \ldots, (s-1)$ do
60:     $o \leftarrow \mathcal{R}[u \bmod \delta]$
61:     $execute(o)$
function $store\_reply(m, j)$:     where $m = \begin{cases} \langle \texttt{reply}, j, s_j, r_j\rangle_{\sigma_j} & \text{if } j \neq i \\ (\texttt{reply}, i, s_i, r_i) & \text{otherwise} \end{cases}$
62: $replies \leftarrow replies \cup \{m\}$
63: if $j \neq i$ then
64:     for replicas $E_k$ s.t. $(\langle \texttt{suspect}, k, j, s_j, c\rangle_{\sigma_k} \in suspects)$ do
65:         forward $m$ to $E_k$
66: if $\exists\ r$: $replies$ contains $\langle \texttt{reply}, k, s_j, r\rangle_{\sigma_k}$ messages from $t+1$ distinct $E_k$ then
67:     $certificate \leftarrow$ set of $\langle \texttt{reply}, k, s_j, r\rangle_{\sigma_k}$ messages from $t+1$ distinct $E_k$
68:     if $(s_j \notin certified)$ then
69:         $certified \leftarrow certified \cup \{s_j\}$
70:         if $(s_j \in must\_do)$ then
71:             send $certificate$ to $\mathcal{AC}$
72: $update_{\mathcal{FD}}(\texttt{got-reply}, s_j, j)$
upon receiving message $\langle \texttt{reply}, j, s_j, r_j\rangle_{\sigma_j}$ from $E_j \notin corrupt$ for the first time:
73: if $i \neq j$ then
74:     $store\_reply(m, j)$
function $pc\_refresh()$:
75: if $|corrupt\ \cup slow| > t$ then
76:     $c \leftarrow c+1$
77:     $slow \leftarrow \emptyset$
78:     $suspects \leftarrow \emptyset$
79: $\mathcal{PC} \leftarrow$ set of $(t+1)$-lowest-ranked replicas that are neither in $slow$ nor in $corrupt$
```

Fig. 4. Protocol APE for Byzantine-Fault-Tolerant Request Execution (Part II)

```
function $fault\_report(k, type, s_k)$:
    80: if $type$ = mute-suspect then
    81:     send $\langle \texttt{suspect}, i, k, s_k, c\rangle_{\sigma_i}$ to all replicas
    82: else if $type$ = implicate then
    83:     find $E_j$ : ($E_j \notin corrupt$ and $\langle \texttt{reply}, j, s_k, r_j\rangle_{\sigma_j} \in replies$ and
                 $\langle \texttt{reply}, k, s_k, r_k\rangle_{\sigma_k} \in replies$ and $r_k \neq r_j$)
    84:     $proof \leftarrow \{\langle \texttt{reply}, j, s_k, r_j\rangle_{\sigma_j}, \langle \texttt{reply}, k, s_k, r_k\rangle_{\sigma_k}\}$
    85:     send ($\texttt{implicate}, s_k, proof$) to all replicas
    86:     $must\_do \leftarrow must\_do \cup \{s_k\}$
    87: else {$type$ = convict}
    88:     $certificate \leftarrow$ set of $\langle \texttt{reply}, j, s_k, r\rangle_{\sigma_j}$ messages from $t + 1$ distinct $E_j$
    89:     $proof \leftarrow certificate \cup \{\langle \texttt{reply}, k, s_k, r_k\rangle_{\sigma_k}\}$
    90:     send ($\texttt{convict}, k, s_k, proof$) to all replicas
    91:     $corrupt \leftarrow corrupt \cup \{E_k\}$
    92:     $must\_do \leftarrow must\_do \cup \{s_k\}$
    93:     $pc\_refresh()$
upon receiving message $\langle \texttt{suspect}, j, k, s_j, c\rangle_{\sigma_j}$ from $E_j \notin corrupt$ for the first time:
    94: if $E_k$'s reply message for agree($s_j$) is in $replies$ then
    95:     forward $E_k$'s reply message for agree($s_j$) to replica $E_j$
    96: $suspects \leftarrow suspects \cup \{\langle \texttt{suspect}, j, k, s_j, c\rangle_{\sigma_j}\}$
    97: if $E_k \notin slow$ and $\langle \texttt{suspect}, h, k, s_j, c\rangle_{\sigma_h}$ messages from $n_e - t$ distinct $E_h$ are in $suspects$ then
    98:     $not\_responsive(s_j, k)$
upon receiving message ($\texttt{indict}, k, s_j, c, proof$) from $E_j \notin corrupt$:
    99: if ($s_j \notin must\_do$) or ($E_k \notin corrupt \cup slow$) then
    100:     if ($proof$ contains $\langle \texttt{suspect}, h, k, s_j, c\rangle_{\sigma_h}$ messages from $n_e - t$ distinct $E_h$) then
    101:         $suspects \leftarrow suspects \cup proof$
    102:         $not\_responsive(s_j, k)$
function $not\_responsive(s', k)$:
    103: send ($\texttt{indict}, k, s', c, proof$) to all replicas, where $proof$ is the set of $\langle \texttt{suspect}, j, k, s', c\rangle_{\sigma_j}$
             messages from $n_e - t$ distinct $E_j$
    104: $slow \leftarrow slow \cup \{E_k\}$
    105: $must\_do \leftarrow must\_do \cup \{s'\}$
    106: $pc\_refresh()$
upon receiving message ($\texttt{implicate}, s_j, proof$) from $E_j \notin corrupt$:
    107: if $\exists\ E_k, E_h$ : $\langle \texttt{reply}, k, s_j, r_k\rangle_{\sigma_k} \in proof$ and $\langle \texttt{reply}, h, s_j, r_h\rangle_{\sigma_h} \in proof$ and $r_h \neq r_k$ then
    108:     for $m \in proof \setminus replies$ do
    109:         $store\_reply(m, \text{sender}(m))$
upon receiving message ($\texttt{convict}, k, s_j, proof$) from $E_j \notin corrupt$:
    110: if $proof$ contains $\langle \texttt{reply}, h, s_j, r\rangle_{\sigma_h}$ messages from $t + 1$ distinct $E_h$ and
           $\langle \texttt{reply}, k, s_j, r_k\rangle_{\sigma_k} \in proof$ and $r \neq r_k$ then
    111:     for $m \in proof \setminus replies$ do
    112:         $store\_reply(m, \text{sender}(m))$
```

Fig. 5. Protocol APE for Byzantine-Fault-Tolerant Request Execution (Part III)

```
Failure Detector $\mathcal{FD}$ for Protocol APE with input parameter $\Delta$ (timeout) at execution replica $E_i$
  initialization:
    113: $\mathcal{H} \leftarrow []$                    {array of size $(\delta - 1)$ containing timer handles}
  function $update_{\mathcal{FD}}(\texttt{command}, s', k)$:
    114: if $command$ = start-monitor then
    115:     $\mathcal{H}[s' \bmod \delta] \leftarrow schedule\_timer(\Delta)$          {start timer and store timer handle}
    116: else if $command$ = stop-monitor then
    117:     $cancel\_timer(\mathcal{H}[s' \bmod \delta])$
    118: else {$command$ = got-reply}
    119:     if $s' \in certified$ then
    120:         find $r$ : ($replies$ contains $\langle \texttt{reply}, j, s', r\rangle_{\sigma_j}$ messages from $t+1$ distinct $E_j$)
    121:         for all $E_h$ : ($\langle \texttt{reply}, h, s', r_h\rangle_{\sigma_h} \in replies$ and $r_h \neq r$) do
    122:             $fault\_report(h, \texttt{convict}, s')$
    123:     else if $\exists\ E_j$ : ($E_j \notin corrupt$ and $\langle \texttt{reply}, j, s', r_j\rangle_{\sigma_j} \in replies$ and
                 $\langle \texttt{reply}, k, s', r_k\rangle_{\sigma_k} \in replies$ and $r_k \neq r_j$) then
    124:         $fault\_report(k, \texttt{implicate}, s')$
  upon receiving signal indicating expiry of timer with handle $\omega$
    125: find $s'$ : ($\mathcal{H}[s'] = \omega$)
    126: for all $k$ : ($E_k \in \mathcal{PC}$ and $\langle \texttt{reply}, k, s', r_k\rangle_{\sigma_k} \notin replies$) do
    127:     $fault\_report(k, \texttt{mute-suspect}, s')$
```

Fig. 6. Failure Detector $\mathcal{FD}$ for Protocol APE at Execution Replica E_i

To simplify the description of Protocol APE, we assume that $\mathcal{AC}$ sends its next request after receiving the reply certificate for its previous request, i.e., $\mathcal{AC}$ has only one outstanding request.

1) Parsimonious Normal Mode: When E_i receives a *first-time* agree($s + 1$) message, the protocol at E_i moves to the parsimonious normal mode (lines 13–18). E_i maintains a queue of requested operations called $\mathcal{R}$, 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 agree(s) when it receives the *first-time* agree($s+1$) message. Hence, E_i increments its sequence number variable s.

If $E_i \notin \mathcal{PC}$, then it does nothing more in this mode. On the other hand, if $E_i \in \mathcal{PC}$, then (lines 16–18) it executes the service operation indicated in the agree message, and sends the result r of the execution to $\mathcal{AC}$ 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 $\mathcal{PC}$ replicas will be sufficient for $\mathcal{AC}$ 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 $s+1$.

2) Transition from Normal to Audit Mode: The protocol at E_i transitions to the parsimonious audit mode from the parsimonious normal mode when

– 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

– 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 $\mathcal{AC}$ must have received a reply certificate for $\texttt{agree}(s)$, 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.

3) Parsimonious Audit Mode: In this mode (lines 19–32), though backups do not execute the request (hence, this mode is labeled parsimonious), both backups and $\mathcal{PC}$ replicas monitor the progress made by the $\mathcal{PC}$ replicas in generating a reply certificate for the request (hence, this is called the audit mode). Specifically, upon switching to the audit mode, an execution replica E_i starts a timer (line 27) 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 $\mathcal{AC}$ obtains a reply certificate. If, on the other hand, the replicas indeed receive a reply certificate in a timely manner from the $\mathcal{PC}$, they forward the certificate to $\mathcal{AC}$, and the protocol will switch back

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

TABLE I

PROTOCOL MESSAGES GENERATED AT E_i DUE TO FAILURE DETECTION

Protocol Message	Trigger for Message Generation
`suspect` for E_k	$\mathcal{PC}$ replica E_k's `reply` message was not received before local timer expiry
`indict` for E_k	E_i received `suspect` messages for E_k from $n_e - t$ distinct replicas directly or indirectly (from another `indict` message)
`implicate` for E_k	the result values in E_k's `reply` message differs from that of replica E_j's, where $E_k, E_j \notin$ *corrupt*
`convict` for E_k	the result values in E_k's `reply` message differs from that in a reply certificate

to the parsimonious normal mode.

To enable the monitoring of progress, the $\mathcal{PC}$ replicas required to send signed `reply` messages to all execution replicas. A $\mathcal{PC}$ 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 parsimonious normal mode (lines 40–41). Otherwise, E_j obtains the result value by executing the operation specified in the corresponding `agree` message from $\mathcal{AC}$ (lines 43–52).

4) Failure Detection and Transition from Audit to Recovery Mode: An execution 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:

– *Slow Replies:* A $\mathcal{PC}$ replica is (deliberately or unintentionally) slow in sending its `reply`.

– *Wrong Replies:* A $\mathcal{PC}$ replica did send its `reply`, but with the wrong result value.

To determine if the $\mathcal{PC}$ is performing its job correctly, every replica E_i has access to a failure detector oracle $\mathcal{FD}$ [12]. The protocol provides an interface *fault_report* that $\mathcal{FD}$ can asynchronously invoke to notify the protocol about the misbehavior of some replica. $\mathcal{FD}$, in turn, provides an interface $update_{\mathcal{FD}}$ (lines 114–124) that Protocol APE synchronously invokes to convey protocol-specific information. During the execution of the $update_{\mathcal{FD}}$ function, $\mathcal{FD}$ can read the global data structures, variables, and input parameters of Protocol APE. Table I gives an overview of the protocol messages generated at a correct replica E_i due to failure detection.

In a Byzantine setting, a corrupted replica may behave perfectly normally; therefore, one is concerned only with identifying replicas that exhibit *detectable Byzantine failures* [13], i.e., failures that can be observed by replicas based solely on the messages they receive or did not receive (when they were supposed to be received), or failures that can be attributed to some replica. In the context of Protocol APE, there are really only two kinds of detectable Byzantine failures of interest: (1) *malicious failures* (sending a `reply` message with wrong result value), and (2) *muteness failures* (not sending a `reply` message). Malicious failures are exhibited only by replicas actually corrupted by the adversary and can be perfectly detected: the difference between the result value in E_k's signed `reply` message and the value in the reply certificate is proof of E_k's corruption, and hence E_k will be added to the $\mathit{corrupt}$ sets at all correct replicas. However, muteness failures [14] cannot be perfectly detected in an asynchronous system, as there is no way to distinguish between a replica that did not send a `reply` message and a correct but slow replica that did indeed send a `reply` message. It is for detection of muteness failures that Protocol APE uses timers. In an asynchronous system, timer-based muteness failure detection can easily achieve *completeness* (i.e, the ability to detect every exhibited failure) by setting aggressive timeout values; *accuracy* (i.e., avoiding mislabeling correct behavior as a failure), however, is impossible to achieve.

Fig. 6 gives the pseudocode for an implementation of $\mathcal{FD}$ at a replica E_i that checks whether the $\mathcal{PC}$ replicas are functioning properly based on a local timeout and protocol information. When E_i is in the audit mode, the call to $\mathit{update}_{\mathcal{FD}}(\texttt{start-monitor}, s)$ (line 27) starts a timer. The timer is disabled (line 117) by a call to $\mathit{update}_{\mathcal{FD}}(\texttt{stop-monitor}, s)$ (line 29); replica E_i will make such a call if (line 28) (1) the $\mathcal{PC}$ changes due to the receipt of an `implicate`, `convict`, or `indict` message (described below) that results in a $\mathcal{PC}$ replica being added to the slow or $\mathit{corrupt}$ set, or (2) if E_i obtains a reply certificate for $\texttt{agree}(s)$, or (3) if the protocol switches to the recovery mode. Note that the call has no effect on a timer that has already expired. When the timer expires, $\mathcal{FD}$ notifies Protocol APE by invoking $\mathit{fault\_report}(k, \texttt{mute-suspect}, s)$ for each $\mathcal{PC}$ replica E_k whose reply message for $\texttt{agree}(s)$ has not yet been received (lines 125–127). E_i sends a signed `suspect` message for E_k to all execution replicas (line 81). The `suspect` message for E_k 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 (described in Section IV-D) is an artefact of imperfect failure detection and is used to keep track of the number of times the slow set is reset or cleared to account for that imperfection.

Consider the point in time when E_i's `suspect` message for E_k is received at a correct replica E_j. If 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 (lines 64–65, lines 94–95) ensures that if at least one correct replica has received E_k's `reply` message for $\texttt{agree}(s)$, then all correct replicas will eventually receive E_k's message. On the other hand, if no correct replica has received E_k's $\texttt{reply}$ message for $\texttt{agree}(s)$ in a timely fashion (determined by the replicas' respective local timers), then each of the $n_e - t$ correct replicas will generate a $\texttt{suspect}$ message for E_k. A correct replica E_i keeps track of all the $\texttt{suspect}$ messages it receives by storing them in a data structure, *suspects* (line 96).

After receiving $\langle \texttt{suspect}, j, k, s, c\rangle_{\sigma_j}$ messages from $n_e - t$ distinct E_js (lines 97–98), replica E_i adds E_k to its *slow* set and sends an $\texttt{indict}$[2] message of the form $(\texttt{indict}, k, s, c, \mathit{proof})$ to all replicas, where *proof* contains the signed $\texttt{suspect}$ messages. E_i also adds s to a set *must_do* 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 $\mathcal{PC}$ or backup, *must* send their own $\texttt{reply}$ messages for those requests. Having added the $\mathcal{PC}$ replica E_k to its *slow* set, E_i updates its $\mathcal{PC}$ accordingly. Any replica E_j at which $E_k \notin \mathit{slow} \cup \mathit{corrupt}$ that receives E_i's $\texttt{indict}$ message will add E_k to its *slow* set, add s to the *must_do* set, send its own similar $\texttt{indict}$ message for E_k to all replicas, and update its $\mathcal{PC}$ (lines 99–102).

To identify wrong replies, Protocol APE invokes the $\mathit{update}_{\mathcal{FD}}(\texttt{got-reply}, s_j, j)$ function (line 72) when the protocol receives E_j's $\texttt{reply}$ message for $\texttt{agree}(s_j)$. $\mathcal{FD}$ compares the result value in the $\texttt{reply}$ message with the values in the $\texttt{reply}$ messages received from other replicas for $\texttt{agree}(s_j)$. Because the state machines are deterministic and request execution is done in sequence number order, any difference in the result values of $\texttt{reply}$ 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 $\texttt{reply}$ message contains a result value different from that in a reply certificate is corrupt. When E_j's result value differs from that in a previously received $\texttt{reply}$ (say from E_k) but a reply certificate for $\texttt{agree}(s_j)$ has not yet been obtained (lines 123–124), $\mathcal{FD}$ notifies Protocol APE by invoking $\mathit{fault\_report}(k, \texttt{implicate}, s_k)$. Replica E_i then sends an $\texttt{implicate}$[3] message of the form $(\texttt{implicate}, s, \mathit{proof})$ to all replicas, where *proof* contains E_j and E_k's $\texttt{reply}$ messages (lines 83–85). A recipient of E_i's $\texttt{implicate}$ message will not know which of the implicated replicas is actually corrupt, but will be convinced of the need to switch to the recovery mode and

[2]The legal term "indict" means "to make a formal accusation against a party by the findings of a jury." In Protocol APE, the "jury" comprising the $n_e - t$ replicas, having not received replica E_k's reply message for $\texttt{agree}(s)$ in a timely manner, accuses replica E_k of being slow.

[3]The legal term "implicate" means "to bring into incriminating connection." In Protocol APE, the $\texttt{implicate}$ message brings two or more replicas into connection with a malicious fault, yet does not pinpoint which replica(s) are actually the corrupted one(s).

add s to the $must\_do$ set. That happens at the recipient through the following control path in the pseudocode: lines 107–109, lines 62–72, lines 118–124, and lines 82–86.

Repeated Transitions from Normal to Audit Mode: A corrupted $\mathcal{PC}$ replica can cleverly degrade protocol performance by repeatedly refraining from sending a `reply` message to $\mathcal{AC}$, 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 $\mathcal{PC}$.

To address the above problem, we observe that in the audit mode, a corrupt PC replica cannot avoid sending replies or send wrong replies without being eventually detected. Leveraging this observation, an implementation of Protocol APE may operate semi-permanently in the audit mode under certain conditions (e.g., if the fraction of requests or the number of consecutive requests that resulted in a transition from the normal to audit mode exceeds a fixed threshold) until failure detection convicts or indicts or implicates some $\mathcal{PC}$ replica, thereby causing the next transition to the recovery mode.

5) Recovery Mode: Only the $\mathcal{PC}$ replicas send `reply` messages in the normal and audit modes. In the recovery mode (lines 33–35), 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 `agree`(s) will eventually be obtained. As in the audit mode, the reply certificate is then forwarded to $\mathcal{AC}$.

To send a `reply` message, a backup first has to determine the result value corresponding to the request contained in `agree`(s). As before, the result is obtained from a reply certificate (if previously received), or otherwise by actual execution of the request (lines 40–52). Before executing the operation specified in the `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 u to keep track of how up-to-date their state is. Only when u becomes equal to $s-1$ can E_i execute the operation specified in the `agree`(s) message. Bringing the state up to date may involve two steps (lines 53–61):

Step 1: If $u < stable$ 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 $stable$ (lines 54–58). 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 $(\texttt{state-request}, \mathit{stable})$ 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 u to be equal to stable.

Step 2: E_i updates its state to reflect the execution of requests with sequence numbers from $u+1$ to $s-1$ (lines 59–61). To perform the update, E_i retrieves those requests from the `agree` messages stored in the local $\mathcal{R}$ 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 [5].

Once a reply certificate has been obtained, it is easy to pinpoint which of the previously implicated replicas (if any) are actually corrupt. If the call to $\mathit{update}_{\mathcal{FD}}(\texttt{got-reply}, s_k, k)$ function detects that E_k's result value for $\texttt{agree}(s_k)$ differs from that in a reply certificate (lines 119–122), then $\mathcal{FD}$ notifies Protocol APE that E_k is corrupted by invoking the $\mathit{fault\_report}(k, \texttt{convict}, s_k)$ function (lines 87–93). Replica E_i adds E_k to its local *corrupt* set, updates its $\mathcal{PC}$ accordingly, and shares this information about E_k with other replicas by sending a `convict` message to all replicas (lines 87–93). The `convict`[4] 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.

6) Primary Committee Changes: At a correct execution replica, any $\mathcal{PC}$ change (line 79) 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 corrupted replicas to force a change in the $\mathcal{PC}$ when the $\mathcal{PC}$ indeed consists of correct and timely replicas. Those messages contain sufficient proof to convince any other correct execution replica to effect the same change in its own local *slow* or *corrupt* sets. As a result, even though correct replicas may temporarily

[4]The legal term "convict" means "to find or prove guilty."

differ in their perspectives of the primary committee, their perspectives will eventually concur.

D. Neutralizing the Effect of Inaccurate Muteness 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 (line 75). To allow the next $\mathcal{PC}$ to be chosen, whenever $|slow \cup corrupt| = t+1$, the $slow$ set is reset to the empty set, $\emptyset$ (line 77). A reset counter c is used to keep track of the number of resets (line 76). Both `suspect` and `indict` messages carry an indication of the reset counter value. This allows the garbage collection of all `suspect` messages with lower reset-counter values, whenever c is incremented (line 78).

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 $c + 1$.

V. Analysis

In this section, we prove the following theorem.

Theorem 1: Given an agreement phase abstraction, Protocol APE provides BFT replication for $n_e > 2t$.

We first establish some technical lemmas that describe the properties of Protocol APE.

Lemma 2 (Total Order): At any two correct execution replicas E_i and E_j, their internal states just after executing the request indicated by `agree`(s) are the same.

Proof: The replicas implement deterministic state machines and are initialized to the same internal state. Recall that $\mathcal{AC}$ satisfies the agreement property specified in Section III. These facts directly imply that the lemma is trivially satisfied for $s = 1$. For $s > 1$, E_i and E_j will execute the request indicated by `agree`(s) (line 47) only after the internal state has been brought up to date to reflect the execution of all requests up to sequence number $s - 1$, i.e., u must be equal to $s - 1$ (ensured by lines 43–44). As can be seen in the $state\_update()$ function (lines 53–61), the state update may involve two parts. If $u < stable$, then the state update is done up to the last stable checkpoint. Any replica that obtains state updates in response to its `state-request` message checks (using the digest of the last stable checkpoint) whether the received state updates are correct before actually applying them (line 58). When $u = \mathit{stable}$, the requests corresponding to sequence numbers from $\mathit{stable}+1$ to $s-1$ are executed in sequence number order (lines 59–61). Thus, when $u = s-1$ at two correct execution replicas E_i and E_j, their internal states will be identical. Then, it follows from the determinism of the state machines and the agreement property of $\mathcal{AC}$ that E_i and E_j's internal states will be the same after they have executed the request indicated by `agree`(s). ■

Lemma 3 (Update Integrity): A correct execution replica updates its internal state in response to the request indicated by `agree`(s) at most once, and only if $\mathcal{AC}$ actually sent `agree`(s).

Proof: We first show that the request indicated by the `agree`(s) message is executed at most once a correct execution replica E_i. The only time when the request indicated by the `agree`(s) message is executed by E_i is during the $\mathit{obtain\_reply}()$ function at line 47. After obtaining the result value r, E_i stores its own `reply` message for `agree`(s) in the *replies* data structure (line 51). Any subsequent invocations of the $\mathit{obtain\_reply}()$ function (lines 17, 25, and 34) will not execute the request again, since the function will retrieve the reply from the *replies* data structure (lines 40–41). Hence, E_i executes the request at most once.

The second part of the update integrity property which states that E_i only updates its internal state in response to the requests actually sent by $\mathcal{AC}$ is trivially satisfied by the protocol. The reason is that the execution of the request indicated by `agree`(s) (line 47 in the $\mathit{obtain\_reply}()$ function) is always preceded by receipt of the `agree`(s) message directly from $\mathcal{AC}$ (line 12). Hence, Protocol APE satisfies update integrity. ■

Lemma 4 (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 replica sent a `reply` message for `agree`(s) with result value r.

Proof: Result integrity is trivially satisfied since, by definition, the reply certificate consists of `reply` messages from $t+1$ distinct replicas, out of which at most t are corrupt. ■

Lemma 5 (Liveness): If $\mathcal{AC}$ sends `agree`(s) to the execution replicas, it eventually receives a reply certificate for `agree`(s).

Proof: Once the *first-time* `agree`(s) message has been received from $\mathcal{AC}$, the next message a correct replica E_i waits to receive from $\mathcal{AC}$ is a *first-time* `agree`($s+1$) or a *retransmit* `agree`(s) message from $\mathcal{AC}$ (line 12). If E_i receives a *first-time* `agree`($s+1$) message, then by our assumption that $\mathcal{AC}$ has only one outstanding request, it is clear that $\mathcal{AC}$ has obtained a reply certificate for `agree`(s); hence, liveness is trivially satisfied.

On the other hand, if E_i receives a *retransmit* `agree`(s) message, then $\mathcal{AC}$ has not obtained a timely reply certificate. In that case, all replicas will eventually switch to the parsimonious audit mode, in which `reply` messages for `agree`(s) are expected from all the $\mathcal{PC}$ replicas.

It is clear that if s is added to E_i's $must\_do$ set when E_i is in the parsimonious audit mode, then E_i switches to the recovery mode (lines 33–35), in which, irrespective of whether E_i is a $\mathcal{PC}$ replica or not, E_i's `reply` message will be received at $\mathcal{AC}$. At replica E_i, the addition of s to its $must\_do$ set (lines 86, 92, and 105) is always preceded by E_i sending an `implicate`, `convict`, or `indict` message with valid proof that will convince any correct recipient replica E_j to add s to its $must\_do$ set as well. Hence, all other correct replicas will also eventually add s to their respective $must\_do$ sets, switch to the recovery mode, and send their own `reply` messages for `agree`(s); hence, $\mathcal{AC}$ will eventually obtain a reply certificate for `agree`(s).

It is easy to see that if s is added to E_i's *certified* set when E_i is in the parsimonious audit mode, then liveness is satisfied, since E_i will forward the reply certificate to $\mathcal{AC}$.

From the above arguments, if we are able to show that the clause ($s \in$ *certified* **or** $s \in must\_do$) eventually holds at a replica E_i that is in the parsimonious audit mode, then that would prove the liveness of Protocol APE. We show exactly that by proving Claim 1 below. Hence, Protocol APE satisfies liveness. ■

Claim 1: At a replica E_i that enters the parsimonious audit mode because of receipt of a *retransmit* `agree`(s) message from $\mathcal{AC}$, ($s \in$ *certified* **or** $s \in must\_do$) holds eventually.

Proof: Suppose that the claim is false, i.e., the clause ($s \in$ *certified* **or** $s \in must\_do$) is never true. Then that implies that E_i never comes out of the **repeat until** loop (lines 22–30). That is possible only in two cases: (1) the **repeat until** loop iterates infinitely and the **wait for** clause in line 28 is satisfied by the clause ($oldpc \neq \mathcal{PC}$) becoming true at each iteration of the **repeat until** loop, or (2) the control gets stuck at the **wait for** clause in line 28. We now show that both cases result in contradictions.

Case (1): In this case, the **repeat until** loop iterates infinitely and there is a $\mathcal{PC}$ change at each iteration of the loop. Every $\mathcal{PC}$ change at E_i is the result of adding some new replica to the set $\mathit{slow} \cup \mathit{corrupt}$. Recall that the next $\mathcal{PC}$ consists of the $t+1$ lowest-ranked replicas that are not in the set. Hence, after $n_e - (t+1)$ $\mathcal{PC}$ changes, all correct replicas will have been on the $\mathcal{PC}$ at least once. As mentioned before, a correct $\mathcal{PC}$ replica in the parsimonious audit mode sends its `reply` message for `agree`(s) to all replicas. Thus, after at most $n_e - (t+1)$ $\mathcal{PC}$ changes in the parsimonious audit mode, all correct replicas will have sent their `reply` messages for `agree`(s). Hence, a reply certificate for `agree`(s) is guaranteed to be obtained eventually at a correct replica, which implies that $s \in \mathit{certified}$ eventually, contradicting our assumption.

Case (2): The control can get stuck at the **wait for** clause in line 28 only if the clause $(\mathit{oldPC} \neq \mathcal{PC})$ never became true at E_i, i.e., the $\mathcal{PC}$ set at E_i never changed. The $\mathcal{PC}$ set at E_i would never change only if there had been no further additions to E_i's slow or $\mathit{corrupt}$ sets. That would be possible only if the slow and $\mathit{corrupt}$ sets at any other correct replica were proper subsets of E_i's slow and $\mathit{corrupt}$ sets respectively.

Since every addition to E_i's slow set is preceded by E_i sending an `indict` message with valid proof to all replicas, all correct replicas will eventually have the same slow set as E_i. Similarly, since every addition to E_i's $\mathit{corrupt}$ set is preceded by E_i sending a `convict` message with valid proof to all replicas, all correct replicas will eventually have the same $\mathit{corrupt}$ set at E_i. Hence, the $\mathcal{PC}$ set at all correct replicas will eventually be the same as E_i's.

To recap, we have so far shown that if the control at E_i gets stuck at the **wait for** clause in line 28, then the $\mathcal{PC}$ set at all correct replicas will eventually be the same as E_i's. Consider the point in time when $\mathcal{PC}$ sets at all correct replicas are the same; then, from that point on, the $\mathcal{PC}$ sets at all correct replicas will remain the same by our assumption that the $\mathcal{PC}$ set at E_i never changes. Recall that the `reply` forwarding done in lines 64–65 and lines 94–95 ensures that if at least one correct replica has received a $\mathcal{PC}$ replica E_k's `reply` message for `agree`(s), then all correct replicas will eventually receive E_k's message. This reply-forwarding logic coupled with our initial assumption that s is never added to the $\mathit{certified}$ set implies that there must be some $\mathcal{PC}$ replica E_k that did not send its `reply` message to any correct replica. However, the failure detectors at all correct replicas would then eventually timeout causing all correct replicas to send `suspect` messages for E_k with sequence number s. When those `suspect` messages are eventually received at E_i, E_i invokes the $not\_responsive(s, k)$ function, resulting in the addition of s to the set $must\_do$. However, that contradicts our initial assumption that the clause ($s \in certified$ **or** $s \in must\_do$) is never true. ■

Proof of Theorem 1: Lemmas 2, 3, 4, and 5 have shown that Protocol APE satisfies the total order, update integrity, result integrity, and liveness properties of BFT replication respectively. Hence, Protocol APE provides BFT replication for $n_e > 2t$. ■

VI. Experimental Evaluation

We implemented and experimentally evaluated Protocol APE under both fault-free conditions and controlled fault injections. We compare 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 parsimonious execution compares with all-active execution for different service types.

A. Experimental Setup

We conducted our experiments for $n_e =$ 3, 5, 7, 9, and 11 execution replicas that can tolerate $t =$ 1, 2, 3, 4, and 5 simultaneous replica faults, respectively. In a real-world setting, $\mathcal{AC}$ would consist of a set of $3t + 1$ agreement replicas; however, to keep the focus on the execution phase of BFT replication, we used a single $\mathcal{AC}$ process to represent the clients and the agreement replicas. The process generated requests and provided the properties of the agreement cluster abstraction given in Section III. We assumed that the samples were from a student's t-distribution and computed confidence intervals for 95$\%$ confidence.

The setup consisted of a testbed of 12 otherwise unloaded machines. Each machine had a single Athlon XP 2400 processor and 512 MB DDR2700 SDRAM 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

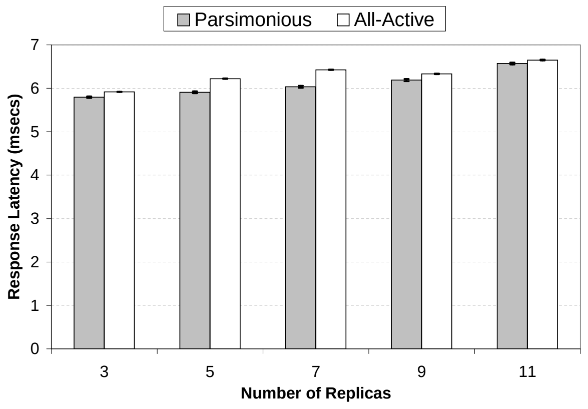
Ethernet network. Digital signatures were generated using 1024-bit RSA cryptography. MACs for realizing secure authenticated channels were computed using the SHA-1 algorithm. Each replica maintained about 1 MB of service-specific state, organized into 1 KB blocks and loaded into its main memory at initialization time.

$\mathcal{AC}$ sends two kinds of requests: *retrieve-compute* requests and *update-compute* requests. A *retrieve-compute* request specifies a block to be retrieved. A replica performs some computation on the block contents, and returns the result of the computation 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). $\mathcal{AC}$ sends a new request after obtaining a reply certificate for its last request. We chose the message sizes to be consistent with the few kilobyte message sizes used in the evaluation of related work (e.g., [5], [7], [15]).

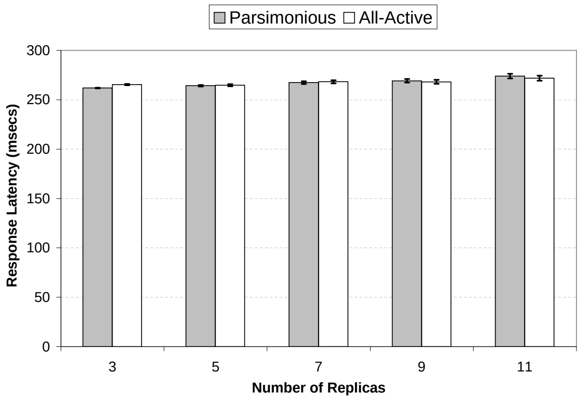
Additionally, for Protocol APE, the checkpoint interval was 200. In choosing the checkpoint interval, there is a tradeoff between increased protocol-related overhead (network traffic and checkpoint execution time) with short checkpoint intervals and the impact on the time to bring a backup replica's state up to date (when switching to the recovery mode) with long checkpoint intervals [16], [17]. There are also other service-related factors involved, e.g., the average percentage of requests that result in a state update; if this number was rather small, then a longer checkpoint interval may be chosen without much impact on the recovery time. In a real-world setting, the checkpoint interval will have to be chosen carefully taking into account such factors and the replicated service's requirements and priorities.

B. 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



(a) CL-2 Requests



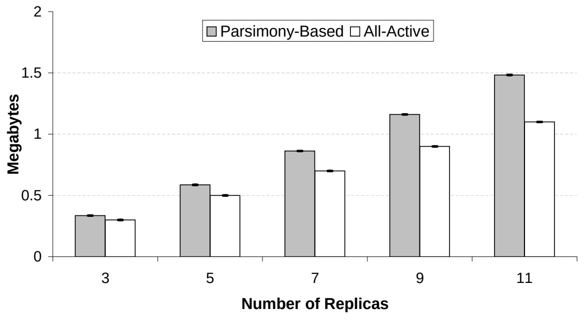
(b) CL-100 Requests

Fig. 7. Request Latency

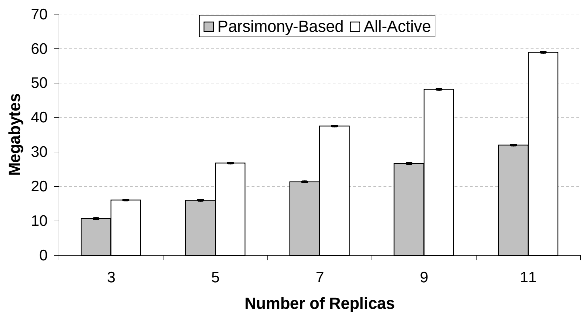
request involved computation of a public key signature on a specified block of the service state 100 times; we call such requests *CL-100* requests. Obviously, one would be hard-pressed to find a real-world application that computes digital signatures 100 times for a request. The intention was to simulate 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 Protocol APE) is an insignificant part of the actual request processing overhead.

We measured request latency, which is the time elapsed from when $\mathcal{AC}$ sends a request until it obtains a reply certificate for the request. Fig. 7(a) compares the request latencies of the parsimonious and the all-active execution approaches for CL-2 requests. Fig. 7(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 $\mathcal{AC}$ process sending about 10,000 requests. $\mathcal{AC}$ generated *retrieve-compute* and *update-compute* requests alternately. The latency for a checkpointing request in parsimonious execution was amortized among all the requests in the corresponding checkpointing interval.

Figs. 7(a) and (b) show that the request latencies for all-active and parsimonious execution are roughly the same. There are two factors in action here: (1) the additional overhead incurred in all-active execution due to the $\mathcal{AC}$ receiving t extra unnecessary `reply` messages and examining their headers and (2) the additional overhead incurred in parsimonious execution due to periodic checkpointing. For the generated workload, the chosen checkpoint interval, and the checkpointing overhead, these two factors seem to balance each other out. If a more aggressive checkpoint interval



(a) Overall Ingress Traffic



(b) Overall Egress Traffic

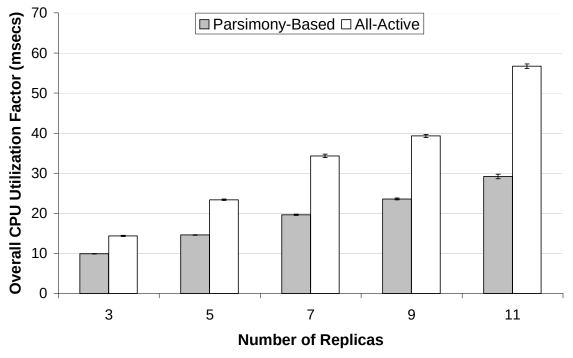
Fig. 8. Overall Traffic Across All Replicas for 5000 CL-2 Requests

was chosen (say, 128—as chosen by Castro and Liskov in their experiments [5]—instead of 200) or if the checkpointing overhead was higher, then the request latency of parsimonious execution would be higher than that of all-active execution.

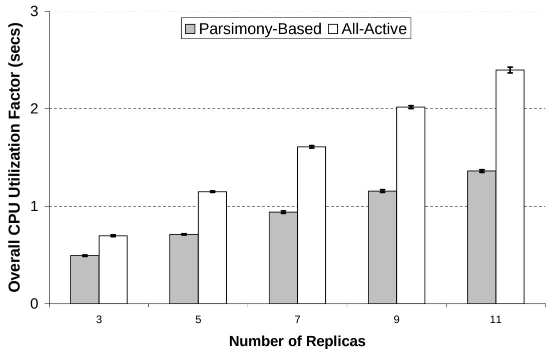
To quantify communication costs, we measured the total number of bytes received at each replica (ingress network traffic) and the total number of bytes sent by each replica (egress network traffic). Figs. 8(a) and (b) show the overall incoming network traffic and the overall outgoing network traffic across all replicas for 5,000 CL-2 *retrieve-compute* requests. For the purpose of measuring communication costs, it doesn't matter whether the requests are CL-2 or CL-100. If *update-compute* requests were used, one would expect similar trends, as the only difference would be the size of `request` messages from $\mathcal{AC}$.

Fig. 8(a) shows that the overall ingress traffic for all-active execution is smaller than that of parsimonious execution, and that the difference grows modestly as the number of replicas increases. The only incoming messages at a replica for all-active execution are the request messages from $\mathcal{AC}$. For parsimonious execution, in addition to the request messages, `reply` messages are received for infrequent checkpoint requests from $\mathcal{PC}$ members. If a more aggressive checkpoint interval was chosen, then the difference would be higher. However, that is not a major disadvantage, since `reply` messages for checkpoint requests carry only the digests of the checkpoints (which are just a few hundred bytes each).

Fig. 8(b) shows that parsimonious execution has about half the overall egress network traffic of all-active execution. That is expected, since only $t + 1$ out of $2t + 1$ replicas send the `reply` messages to $\mathcal{AC}$. The difference (and hence the advantage of parsimonious execution) is more pronounced as t increases. If overall network traffic (i.e., both ingress plus egress traffic) were to



(a) CL-2 Requests



(b) CL-100 Requests

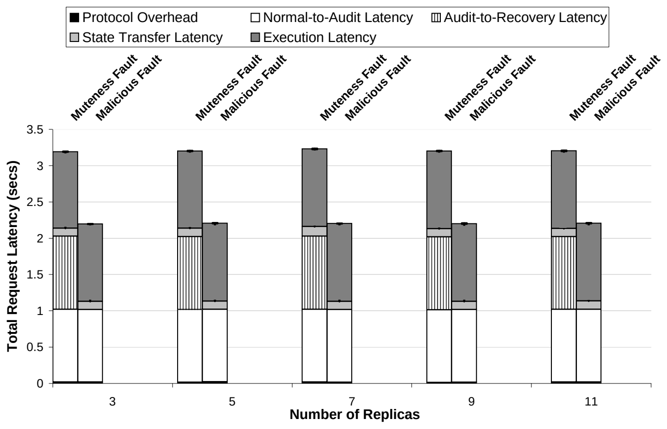
Fig. 9. Overall CPU Utilization Factor Per Request

be considered, this benefit of parsimonious execution would far outweigh its drawback with respect to overall ingress traffic.

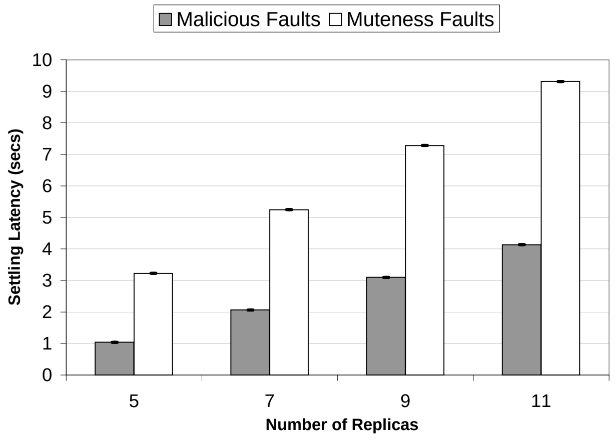
In our experiments, we did not vary the sizes of the `request` and `reply` messages. However, the average size of the `reply` messages certainly affects the difference in the overall network traffic generated by the all-active and parsimonious approaches. Large `reply` message sizes, which is a trend observed in HTTP and web services traffic [18], will amplify the benefits of the parsimonious approach in the normal case. The difference is less influenced by the `request` message size, since all replicas receive `request` messages in both approaches.

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 $\mathcal{PC}$ replicas in parsimonious 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 parsimonious execution were negligible for both CL-2 and CL-100 requests.

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. Figs. 9(a) and (b) show the overall CPU utilization factor for CL-2 and CL-100 requests. The utilization factor for parsimonious



(a) Request Latency under Faulty $\mathcal{PC}$ Member



(b) Settling Latency for Multiple Correlated Faults

Fig. 10. Behavior Under Fault Injections

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 VII-B), 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.

C. Behavior in the Presence of Fault Injections

We injected both muteness faults and malicious faults in our protocol. A muteness fault injection was done by crashing a $\mathcal{PC}$ member upon receipt of a specified $\mathcal{AC}$ request. A malicious fault injection was done by making a $\mathcal{PC}$ member send wrong values in its `reply` messages. 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.

Fig. 10(a) shows the different factors that contributed to the $\mathcal{AC}$ request latency when a $\mathcal{PC}$ member was fault-injected after servicing a sufficiently large number of CL-2 requests (about 5,000). The highest latency is obtained when a muteness fault injection is done. The latency comprises two timeout values, state update latency, and the overhead due to receiving and processing protocol messages. The first timeout value of 1 second (represented by "normal-to-audit latency" in the graph) is used at $\mathcal{AC}$ before $\mathcal{AC}$ sends a *retransmit* message for the request. Receipt of that message will cause the protocol to switch from parsimonious normal mode to parsimonious 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 $\mathcal{PC}$ member, as the replica would not have received a reply message from the $\mathcal{PC}$ member for the request. Once $n_e - t$ suspect messages for the crashed $\mathcal{PC}$ member have been received, the protocol switches from the parsimonious 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 $\mathcal{PC}$ 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 $\mathcal{PC}$ 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 $\mathcal{PC}$ member. The portion marked "protocol overhead" in the graph includes a round-trip transmission time from $\mathcal{AC}$ to the replica (for the request message from $\mathcal{AC}$ and the reply message from the replica) plus other overhead related to Protocol APE (such as exchange of suspect and indict messages, and selection of a new $\mathcal{PC}$). The graph shows that the overhead due to receiving and processing protocol messages (represented by "protocol overhead") is roughly 20 milliseconds. In a real-world setting, all other parts of the request latency under $\mathcal{PC}$ faults will vary based on environment- and workload-specific parameters such as the timeout value and service-specific parameters, 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 $\mathcal{AC}$ 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.

Fig. 10(b) quantifies the effect of multiple correlated fault injections. After servicing a sufficiently large number of requests (5,000), we injected multiple faults at the replicas so that a new $\mathcal{PC}$ 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 $\mathcal{AC}$ process 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 $\mathcal{PC}$ 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 Protocol APE. 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.

VII. Discussion

In this section, we discuss the practical significance of our protocol, present examples of applications that would benefit from our protocol, and compare our protocol with related work.

A. Practical Significance

The all-active approach is clearly a failure masking one; despite t corrupted execution replicas, a reply certificate will eventually be obtained at $\mathcal{AC}$ for every request, since all replicas execute all requests and send the results to $\mathcal{AC}$. Protocol APE has the same fault resilience and guarantees the same property as the all-active approach, but does so in a manner that is "normally" much more resource-efficient by additionally employing failure detection and checkpointing.

Protocol APE was designed using the parsimonious approach for constructing fault-tolerant protocols [19]. In the parsimonious approach, we design the protocol with the explicit aim of achieving frugality or efficiency with respect to a given metric of interest while never violating correctness (i.e., safety and liveness). The emphasis on always guaranteeing correctness is reflected by our formal system model in which the adversary controls the scheduling of messages including the timeouts of the failure detector. For Protocol APE, the metric of interest is overall resource usage. When certain operational assumptions are satisfied, the protocol operates in the parsimonious modes using only $t+1$ replicas to actually execute the request. What are these operational assumptions of the parsimonious modes? They are the assumptions that the primary committee behaves correctly and that the failure detection has a high likelihood of being accurate. The optimistic hope is that those assumptions are satisfied more often than not, thereby making the chosen parsimonious mechanism applicable for most of the system's lifetime. In other words, the optimistic hope is that, unlike the adversary in our formal model, the network in a real-world setting will not always behave in the worst possible manner. Such a hope is not unrealistic, since practical observation shows that in many systems, network behavior alternates between long periods of stable conditions and relatively short periods of instability [20].

Even if the optimistic hope is not satisfied, the parsimonious modes never violate safety and Protocol APE satisfies liveness because the failure detection eventually triggers a switch to the recovery mode. 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 $\mathcal{PC}$ 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.

Although inaccuracy and latency of failure detection do not affect safety and liveness, the protocol's performance characteristics are, in large part, influenced by how quickly and accurately the failure detection signals the occurrence of faults or unstable conditions. Although relying on the accuracy of failure detection mechanisms for performance and not for correctness seems more reasonable than approaches that rely on those mechanisms for correctness (such as [5], [21]–[23]), that reliance limits the practical applicability of Protocol APE to environments with known stable characteristics, in which the failure detection can be preset for high accuracy. In environments that are dynamic or evolve over time, static failure detectors may lose their accuracy over time. In such environments, despite the optimal resource usage obtained in the parsimonious modes of operation, it is very likely that Protocol APE will lose out in terms of request latency to the all-active approach which does not rely on any failure detection. That is because, even in fault-free conditions, the parsimonious protocol may be able to make progress only after switching to the recovery mode of each epoch, and under faulty conditions, the detection latency may be high. Thus, in such evolving and dynamic environments, to be able to obtain both an overall resource usage that is optimal (when amortized over the lifetime of the system) and acceptable request latency under faulty and/or unstable conditions, a real-world deployment of Protocol APE must be complemented with mechanisms (such as [24], [25]) for adaptively tuning the parameters of the failure detection.

B. Practical Applications

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.

Until recently, the common practice at ASP data centers has been to deploy one server per application. This led to an uncontrolled increase in the number of grossly under-utilized servers resulting in manageability problems, a situation commonly referred to as *server sprawl*. *Virtualization* is an architectural approach that adds a *hypervisor* layer between the OS and hardware. It allows hosting multiple well-isolated virtual machines (VMs) on the same physical machine, with each VM running its own OS and applications. Vendors like VMware, IBM, and HP have released their own virtual infrastructure management software that provide a unified view of the VMs to the data center administrator and allow migration of VMs from one physical machine to another. VMs belonging to multiple organizations can be co-hosted on the same physical machine. In virtualized data centers, a pricing model that charges the outsourcing company based on the number of physical machines used seems less relevant, and consequently there is an increasing shift towards consumption fees based on the actual overall resource use.

In this context, BFT replication can be useful for enhancing the trustworthiness of computations, and our protocol can be useful for both the outsourcing company and the ASP. The outsourcing company benefits from the significant reductions in overall execution costs (compared to all-active execution) and, thereby, the reduced fee to be paid to the ASP. Although our results show that the benefits of the parsimonious approach are amplified with increasing replication degree, the approach gives savings in the overall resource usage for all replication degrees. That is true even if the replication degree is small, as it typically tends to be for BFT replication due to reasons such as as the increasing difficulty of ensuring independent replica failures for higher degrees. For a fixed replication degree, if there are multiple independent replicated services that use parsimonious execution on a given physical infrastructure, then the overall gains from our approach are more pronounced as the number of such services increases. For example, if the ASP caters to multiple outsourcing companies each having its own replicated service(s), then, as the number of services increases, the overall resource usage savings for the ASP increases. The resource savings obtained may be used to accommodate more VMs (perhaps belonging to other outsourcing companies), thereby enabling the ASP to obtain better utilization for its physical infrastructure and better server consolidation. That, in turn, could help control the problem of server sprawl and its associated management and space costs.

C. Related Work

BFT replication techniques are of two categories: quorum replication and state machine replication. Quorum replication (e.g., [26], [27]) 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, when just one quorum is aware of an update, it is difficult to implement concurrent updates of the same information on different quorums. 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., [5], [21]), 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 parsimonious execution has been routinely used in primary-backup systems that tolerate benign faults (e.g., [28], [29]), our protocol is novel in that it is the first to apply parsimony to Byzantine fault tolerance.

Since our protocol is for the execution phase, our work is complementary to the BASE work [6] and the BASE extension by Yin et al. [7]. Specifically, 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 Yin et al. [7] to obtain BFT confidentiality.

VIII. 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 all $n_e > 2t$ execution replicas to execute a request. Despite failures and instability, our protocol always guarantees both safety and liveness as long as no more than $t < n_e/2$ execution replicas are corrupted. 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 $\mathcal{PC}$ consisting of $t + 1$ execution replicas to execute 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.

References

[1] H. V. Ramasamy, A. Agbaria, and W. H. Sanders, "A Parsimony-Based Approach for Obtaining Resource-Efficient and Trustworthy Execution," in *Proc. 2nd Latin-American Symposium on Dependable Computing*, Oct 2005, pp. 206–225.

[2] F. B. Schneider, Ed., *Trust in Cyberspace*. National Academy Press, 1999.

[3] L. Lamport, "Time, Clocks and Ordering of Events in Distributed Systems," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[4] F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, December 1990.

[5] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, November 2002.

[6] M. Castro, R. Rodrigues, and B. Liskov, "BASE: Using Abstraction to Improve Fault Tolerance," *ACM Transactions on Computer Systems*, vol. 21, no. 3, 2003.

[7] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating Agreement from Execution for Byzantine Fault Tolerant Services," in *Proc. 19th Symposium on Operating Systems Principles*, October 2003, pp. 253–267.

[8] K. Kursawe, "Optimistic Byzantine Agreement," in *Proc. 21st Symposium on Reliable Distributed Systems*, October 2002, pp. 262–267.

[9] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and Efficient Asynchronous Broadcast Protocols," in *Advances in Cryptology: CRYPTO 2001 (J. Kilian, ed.), LNCS 2139*. Springer-Verlag, 2001, pp. 524–541.

[10] S. A. Vanstone, P. C. van Oorschot, and A. Menezes, *Handbook of Applied Cryptography*. CRC Press, 1996.

[11] S. Goldwasser, S. Micali, and R. L. Rivest, "A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks," *SIAM Journal on Computing*, vol. 17, no. 2, pp. 281–308, 1988.

[12] T. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[13] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "Byzantine Fault Detectors for Solving Consensus," *Computer J.*, vol. 46, no. 1, pp. 16–35, 2003.

[14] A. Doudou, B. Garbinato, and R. Guerraoui, "Failure Detection: From Crash-Stop to Byzantine Failures," in *Proc. Intl. Conf. on Reliable Software Technologies (LNCS 2361)*, May 2002, pp. 24–50.

[15] M. K. Reiter, "Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart," in *Proceedings of the 2nd ACM Conference on Computer and Communication Security*, November 1994, pp. 68–80.

[16] A. Agbaria, A. Freund, and R. Friedman, "Evaluating Distributed Checkpointing Protocols," in *Proc. 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, May 2003, pp. 266–273.

[17] N. H. Vaidya, "Impact of Checkpoint Latency on Overhead Ratio of a Checkpointing Scheme," *IEEE Transactions on Computers*, vol. 46, no. 8, pp. 942–947, August 1997.

[18] S. M. Kim and M.-C. Rosu, "A Survey of Public Web Services," in *Proceedings of the 5th International Conference on E-Commerce and Web Technologies*, ser. Lecture Notes in Computer Science, B. P. K. Bauknecht, M. Bichler, Ed., vol. 3182. Springer, Aug 2004, pp. 96–105.

[19] H. V. Ramasamy, C. Cachin, A. Agbaria, and W. H. Sanders, "The Parsimonious Approach to Constructing Fault-Tolerant Protocols," in *Supplemental Proceedings of the 2006 International Conference on Dependable Systems and Networks (DSN-2006)*, June 2006, pp. 218–219.

[20] F. Cristian and C. Fetzer, "The Timed Asynchronous Distributed System Model," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 642–657, June 1999.

[21] M. K. Reiter, "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, pp. 99–110.

[22] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "Solving Consensus in a Byzantine Environment Using an Unreliable Fault Detector," in *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, December 1997, pp. 61–75.

[23] P. Pal, P. Rubel, M. Atighetchi, F. Webber, W. H. Sanders, M. Seri, H. Ramasamy, J. Lyons, T. Courtney, A. Agbaria, M. Cukier, J. Gossett, and I. Keidar., "An Architecture for Adaptive Intrusion-Tolerant Applications," *Software Practice and Experience, Special Issue on Auto-Adaptive and Reconfigurable Systems*, 2006.

[24] M. R. C. Fetzer and F. Tronel, "An Adaptive Failure Detection Protocol," in *Proceedings of the 8th Pacific Rim International Symposium on Dependable Computing*, 2001, pp. 146–153.

[25] P. Verissimo and A. Casimiro, "The Timely Computing Base Model and Architecture," *IEEE Transactions on Computers*, vol. 51, no. 8, pp. 916–930, 2002.

[26] D. Malkhi and M. Reiter, "Byzantine Quorum Systems," *J. Distributed Computing*, vol. 11, no. 4, pp. 203–213, 1998.

[27] J.-P. Martin, L. Alvisi, and M. Dahlin, "Minimal Byzantine Storage," in *Proceedings of the 16th International Symposium on Distributed Computing (DISC 2002), LNCS 2508*. Springer-Verlag, 2002, pp. 311–325.

[28] N. Budhiraja, F. Schneider, S. Toueg, and K. Marzullo, "The Primary-Backup Approach," in *Distributed Systems*, S. Mullender, Ed. ACM Press - Addison Wesley, 1993, pp. 199–216.

[29] X. Défago and A. Schiper., "Specification of Replication Techniques, Semi-Passive Replication, and Lazy Consensus," EPFL, Switzerland, Tech. Rep. IC-2002-07, February 2002.