PARSIMONIOUS SERVICE REPLICATION FOR TOLERATING MALICIOUS ATTACKS IN ASYNCHRONOUS ENVIRONMENTS

BY

HARIGOVIND VENKATRAJ RAMASAMY

B.ENGR., Anna University, 1999
M.S., University of Illinois at Urbana-Champaign, 2002

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Abstract

We consider the subject of tolerance of the most severe kind of faults, namely Byzantine faults, through state machine replication in asynchronous environments such as the Internet. In Byzantine-fault-tolerant (BFT) state machine replication, state consistency among the replicas of a service is maintained by first agreeing on the order of requests to be processed (agreement or atomic broadcast phase) and then executing the requests in the agreed-upon order (execution phase).

We propose a methodology for constructing asynchronous BFT replication protocols that leverage perceived normal conditions for parsimony and do not compromise correctness even when such perceptions are inaccurate. Parsimony is to be as frugal as possible for a given metric of interest. We apply this methodology to obtain parsimonious protocols that achieve efficiency in three metrics: (1) overall resource use of request execution, (2) message complexity of atomic broadcast, and (3) latency degree of atomic broadcast. We then present a suite of group management protocols that allow for the dynamic change of the composition of the replication group. Our parsimonious protocols are designed to withstand corruptions of at most one-third of the replicas and do not *require* the removal of suspected faulty replicas in order to provide liveness. Such a design allows for the enforcement of very selective and conservative policies regarding changes to the replication group membership.

We describe the implementation of the protocols within a reusable software framework. We also present the experimental evaluation of our protocols in the context of a representative application in both LAN and WAN (Planetlab) settings under both fault-free and controlled fault injection scenarios.

To Rachel, Amma, Naina, and Ashok.

Acknowledgments

I owe my thanks to several people and organizations for their assistance in making this dissertation possible. Many thanks to my advisor, Professor William H. Sanders, for his invaluable guidance, for sharing his knowledge and experience on several issues (both technical and non-technical), for giving me exposure and opportunities, and for providing constant encouragement and support throughout the five years I spent in his group. A special thanks to Dr. Christian Cachin of IBM Zurich Research Labs for the very productive collaboration that led to the work in this dissertation; the summer I spent in Zurich under his mentorship gave me the boost and direction I needed to take this dissertation to completion. I would also like to thank the rest of my thesis committee, Professors Ravishankar Iyer, Klara Nahrstedt, and Indranil Gupta of Illinois and Dr. Rick Schlichting of AT&T Research for their many important comments and discussions.

Mouna Seri deserves a special word of thanks; without her help, it would not have been possible to complete the software development work and the experimentation for this dissertation. The almost-daily discussions I had with her in the last few months of my stay in Illinois helped me greatly in shaping the software design. Thanks to Luke St. Clair who was also a great help in the software development. I would also like to thank Jenny Applequist, who was immensely helpful in editing and proofreading this thesis and all other work I published during my graduate study. She has been always very patient, helpful, and timely despite being swamped with an incredible amount of work.

I am also very thankful to Dr. Adnan Agbaria of the Information Sciences Institute and Professor Michel Cukier of the University of Maryland for their technical guidance and mentorship and for the many interesting and helpful technical discussions during their postdoctoral stays in the PERFORM research group.

I would also like to thank all other members (past and present) of the PERFORM group with whom I had the honor to interact and learn from: Tod Courtney, David Daly, Shravan Gaonkar, Vishu Gupta, Mark Griffith, Mike Ihde, Kaustubh Joshi, Sudha Krishnamurthy, Vinh Lam, Ryan Lefever, Prashant Pandey, Eric Rozier, Sankalp Singh, and Fabrice Stevens. They have all been excellent colleagues. Thanks particularly to Ryan and Kaustubh for the many stimulating discussions and for their useful comments on this research.

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CNS-0406351 and DARPA contract F30602-00-C-0172. Any opinions, findings, and conclusions or recommendations expressed in this material are mine and do not necessarily reflect the views of the National Science Foundation or DARPA. I am grateful for NSF and DARPA's support of my research.

My wife Rachel, my parents, and my brother Ashok provided invaluable moral support and encouragement. It is through their unconditional love and prayers that I was able to get through the most frustrating times of my Ph.D. process. They gave me the strength to see through the completion of this thesis. I am also very thankful to Rachel's family for their kindness and support.

Finally, I would like to thank my friends Nitin Aggarwal, Prof. Raghupathy Sivakumar, and Neelima Mehta for the memorable times I spent with them in Urbana-Champaign.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

NIS	Networked Information System.
BFT	Byzantine Fault Tolerance
PABC	Parsimonious Asynchronous Atomic Broadcast.
APE	Asynchronous Parsimonious Execution.
GMA	Group Membership Agreement.
CoBFIT	Component-Based Framework for Intrusion Tolerance.
TCP	Transmission Control Protocol.
UDP	User Datagram Protocol.
DoS	Denial-of-Service.
LAN	Local Area Network.
WAN	Wide Area Network.
COTS	Commercial Off-the-shelf.
AVI	Attack-vulnerability-intrusion.

Chapter 1

Introduction

Today, networked information systems (NISs) have become pervasive. Every aspect of society, be it energy, transportation, business, finance, defense, telecommunications, or manufacturing, is dependant on NISs for gathering, processing, and distributing information. Military command and control, communication infrastructure, aviation control, and electric power control grids are all dependant on NISs. The ever-increasing dependency on NISs is naturally accompanied by ever-increasing concerns for the potential consequences of failures of NISs. These concerns are only compounded by the increasingly sophisticated, effective, and numerous attacks by amateurs (through readily available tools and scripts), insiders, cyber-terrorist organizations, and rogue nations on NISs (particularly on high-value targets). In the light of such concerns, the need to make NISs *trustworthy* has never been more crucial. The trustworthiness of an NIS is judged by its ability to continually meet stated goals despite accidents, design and implementation errors, operator errors, and malicious attacks [Sch99].

1.1 Current Approaches for Trustworthiness Despite Malicious Attacks

There are several approaches for enhancing trustworthiness in the presence of malicious attacks. It is now widely recognized that a "defense-in-depth" strategy that combines two or more of the following approaches may be more effective than the disjointed employment of just one specific approach.

The traditional security approach to enhancing trustworthiness aims to build systems that are equipped with defense mechanisms to prevent malicious attacks from becoming intrusions. It also tries to identify vulnerabilities in components either by rigorous testing before deployment or from successful attacks after deployment, and patches them. Although this approach has been effective in handling a good number of malicious attacks, practical experience shows that most (if not all) systems remain vulnerable, at least to some extent. This is particularly true for large and complex NISs, in which the correct functioning of an application can depend on the possibly complex interactions of software running on many nodes.

The fault tolerance approach to enhancing trustworthiness in the face of malicious attacks is to treat security hazards (i.e., attacks, vulnerabilities, and intrusions) as faults and to equip the system with the ability to continue desired operation despite the activation of faults. Verissimo et al. [VNC03] describe an *attack-vulnerability-intrusion* or AVI model in which they define vulnerabilities, attacks, and intrusions as faults in the following manner. A vulnerability is defined as "a malicious or accidental, design[1] or operational[2] fault in the system that could potentially be exploited with malicious intent." An attack is a "malicious operational interaction fault performed with the objective of exploiting one or more vulnerabilities." An intrusion is a "malicious operational interaction fault that results when an attack successfully exploits one or more vulnerabilities."

Fault tolerance can assume two forms: (1) error processing and (2) fault treatment. In the context of intrusions, error processing involves intrusion detection (e.g., detection of a virus) and intrusion response mechanisms or countermeasures (e.g., deletion of virus-infected files, deactivation of certain malicious user accounts, or disabling of ports) to allow recovery from the intrusions. Error processing also includes error masking, e.g., voting among system components, a subset of which may have been compromised by the attacker. Fault treatment

[1]Design faults are those that are created during system design or development.

[2]Operational faults are introduced at run-time during system execution.

includes intrusion diagnosis to identify the cause of the intrusion and isolation/exclusion of the intruded components from the system. Examples of fault isolation include removal of corrupted servers from the system, placement of partially corrupted servers in quarantine, and restarting of corrupted servers from a safe state.

Fault removal involves verification, diagnosis, and correction. Verification is the process of checking whether the system satisfies all the properties stated in the system specification. If all stated properties are not satisfied, then diagnosis is performed to identify the reason(s) why the system is unable to satisfy the properties. Diagnosis is followed by the necessary corrective steps until the system does fulfill its specified properties. It is also important to check whether the fault removal procedure has introduced new faults into the system.

Fault forecasting evaluates the fault occurrence and activation history of the system, e.g., attack prediction by analyzing system logs and audits that show increased port scan activity. The forecasting could be probabilistic (e.g., determining the probability that the system will satisfy its dependability properties given that certain components have been successfully compromised by the attacker) or non-probabilistic (e.g., predicting the set of possible attacks based on the current state of the system with respect to the system attack tree). Forecasting may be necessary to warn about impending attacks, to predict the attacker's next course of action given that he/she has intruded into some parts of the system, and to provide useful information about how to respond to the attacks.

1.2 Fault Tolerance Using State Machine Replication

Among the above approaches for enhancing trustworthiness, our focus in this thesis is fault tolerance, specifically, fault tolerance using state machine replication. In the state machine replication approach [Lam78,Sch90], a service that needs to be made trustworthy is modeled as a deterministic state machine and multiple versions (or replicas) of the state machine are developed and deployed across distinct nodes of a distributed system. If the replicas are coordinated to ensure that their states are consistent after executing each operation, then this approach can effectively mask the failure of a fraction of the replicas.

In fault-tolerant systems, *fault models* are assumptions about how components can fail. Fault models abstractly describe the possible behavior of faulty components. The simplest fault model is the crash fault model, in which a faulty component simply stops performing any computation or sending of messages. In the timing fault model, a component always sends the right contents, but the contents may arrive late or not at all. In the value fault model, the messages sent by a component may not comply with the specifications. The most general fault model is the Byzantine fault model, in which faulty components may exhibit arbitrary deviations from their specifications. This model accounts for all possible effects of failure. For example, a Byzantine faulty component may send messages with wrong values, messages that arrive too early or too late, or messages should never be sent at all. The model also accounts for all possible causes of failures. For example, the deviations from specifications may be due to a Trojan horse, malicious intrusion, random bit flips, or programming error. Because of its generality, the Byzantine fault model is an attractive way to model the behavior of a compromised node that is under the control of an adversary and the situation in which multiple compromised nodes collaborate to bring the system down.

1.3 Byzantine Fault Tolerant (BFT) Replication: Status Quo

There is substantial work on fault-tolerant protocols that can tolerate benign faults (for example, [Eln93, Hay98, CT96, DGM02]). Here, we limit our attention to protocols that can tolerate Byzantine faults.

Byzantine fault tolerance as an area has existed for more than two decades, beginning with the seminal work of Lamport, Shostak, and Pease [LSP82], and includes a large body of work (e.g., [BG93, CP02, CR93, CL02, DRS90, MR00, KMMS03, KMMS01, MR97, MR98, MAD02b, Rab83, YMV$^+$03, ZSvR02]). Byzantine-fault-tolerant replication techniques fall broadly into two categories: quorum replication and state machine replication. Byzantine-fault-tolerant quorum replication (e.g., [MR98, MR00, MAD02b, ZSvR02]) uses subsets of replicas (called *quorums*) to implement operations for reading/writing on individual variables of a data repository and for lock acquisition, such that the intersection of any two subsets contains enough correct replicas. Byzantine-fault-tolerant state machine replication [CP02, CL02, YMV$^+$03] can be used to perform arbitrary computations accessing arbitrary numbers of variables, whereas quorum replication is less generic and cannot handle concurrent requests by clients to update the same information.

Consider a service that needs to be made fault-tolerant using state machine replication. To maintain state consistency across all correct replicas of the service, they must first agree on the order of requests (from the clients of the service) to execute and then execute the requests in the agreed-upon order. The problem of reaching agreement among correct replicas in the presence of some Byzantine-faulty ones is called *Byzantine agreement*. This problem is equivalent to the problem of atomic broadcast which is to ensure that, given a set of broadcast messages, all correct parties deliver the same sequence of broadcast messages despite the presence of faulty parties. Fischer, Lynch, and Patterson [FLP85] showed in their seminal work that any asynchronous atomic broadcast protocol must use randomization, since deterministic solutions cannot be guaranteed to terminate [FLP85].

Early work focused on the polynomial-time feasibility of randomized agreement [Rab83, CR93, BG93] and atomic broadcast [BB93], but such solutions are too expensive to use in practice. Many protocols have followed an alternative approach and avoided randomization completely by making stronger assumptions about the system model, in particular by assuming some degree of synchrony (like Rampart [Rei95], SecureRing [KMMS01], and ITUA [RPL$^+$02]). However, most of these protocols have an undesirable feature that makes them inapplicable for wide-area networks such as the Internet: they may violate safety if synchrony assumptions are not met. For example, Kihlstrom et al. [KMMS01] and Reiter [Rei95] describe group communication protocols that can be used to implement state machine replication. Both protocols rely on failure detectors to remove faulty group members in order to make progress. Since failure detectors are not perfect, correct but slow processes may be incorrectly removed from the group, potentially resulting in the invalidation of the assumption that at most one-third of the group members are faulty and thereby violating correctness. These protocols also provide no defense against an adversary that launches denial-of-service attacks by instigating membership changes.

Only recently, Cachin et al. proposed practical asynchronous agreement [CKS05] and atomic broadcast [CKPS01] protocols that have optimal resilience $t < n/3$. Both protocols rely on a trusted initialization process and on public-key cryptography. Cachin et al.'s atomic broadcast protocol proceeds in rounds, with each round involving a randomized Byzantine agreement and resulting in the atomic delivery of some payload messages.

Castro and Liskov's BFT library [CL02] showed that BFT replication systems can be built that add only modest extra latencies relative to unreplicated systems. They also showed that *proactive recovery* can be used to significantly increase the coverage of the assumption that there are at most a threshold number (one-third) of replicas that can be corrupted by the adversary.

A drawback of BFT replication that limited its applicability in many real-world settings was the requirement that all replicas should run the same service implementation and update their states deterministically. If all replicas ran the same service implementation, then an adversary could exploit the same vulnerabilities or software bugs to cause all replicas to fail simultaneously. The determinism requirement is non-trivial to satisfy in many real-world services. Rodrigues et al. [RCL01] 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. The 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. [YMV$^+$03] improved BASE by enforcing a clean separation between agreement on the request delivery order and execution of requests in the agreed-upon order. It is wellknown that agreement in the presence of t Byzantine-faulty replicas requires a minimum of $3t + 1$ replicas [LSP82]. However, separating the agreement and execution phases allows the number of replicas involved in the execution of requests to be reduced from $3t+1$ to $2t+1$. The separation also opened up the possibility of including a privacy firewall between the two phases that could be used to enhance confidentiality by preventing a malicious participant in the execution phase from disclosing unauthorized information to users.

1.4 Objectives, Contributions, and Approach

As is evident from the previous section, while much of the earlier work in BFT 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.

Our research takes a step forward in this direction and aims to make BFT state machine replication more practical in the Internet. The Internet is an adversarial environment where an attacker can compromise and completely take over nodes. Because of the loosely synchronized nature of nodes on the Internet, relying on any synchrony assumptions for correctness is a vulnerability that an adversary can exploit, for example, through denial-of-service attacks. Hence, it is clear that for replication protocols to be applicable in the Internet, the protocols must be designed to work in the asynchronous system model.

Our goal was to develop a comprehensive suite of protocols for BFT replication that use very weak assumptions that are difficult to invalidate, even in wide-area networks such as the Internet, and at the same time have much better average efficiency characteristics than previous approaches. To obtain this goal, we propose a methodology for constructing asynchronous Byzantine-fault-tolerant state machine replication protocols that (1) leverage perceived normal conditions for parsimony and do not compromise correctness even when such perceptions are inaccurate, and (2) have the ability to dynamically change the membership of the replication group without relying on that ability for correctness in the presence of faults.

The research described in this thesis makes the following contributions.

1. *An asynchronous protocol for the execution phase of BFT replication that is parsimonious in the overall amount of resources used for request execution.* Our work aims to address two deficiencies of the status quo. First, most work on BFT replication has focused almost exclusively on the hard problem of atomic broadcast (or, equivalently, Byzantine agreement), often overlooking the practically significant execution phase of BFT replication. Second, while a lot of focus has been placed on developing protocols with optimal fault resilience and thereby reducing the *deployment* costs of BFT replication, not much emphasis has been placed on reducing the *run-time* or *operational* costs of BFT replication, although those are likely to be at least as important as deployment costs in many long-lived and resource-intensive applications. To address these deficiences, we propose an asynchronous, resource-efficient execution protocol whose amortized overall resource use is $(t+1)\cdot X$, where X is the average resource use per request executed for an unreplicated service. The previous most resource-efficient way to execute requests in BFT replication was proposed by Yin et al. [YMV$^+$03] and had amortized overall resource use of $(2t+1)\cdot X$.

2. *An asynchronous atomic broadcast protocol for the agreement phase of BFT replication that is parsimonious in the number of protocol messages that need to be communicated per atomically delivered payload.* Specifically, we propose an asynchronous, message-efficient atomic broadcast protocol whose amortized *message complexity* is $O(n)$, where n is the number of parties involved. Message complexity of an atomic broadcast protocol is the number of protocol messages generated by correct parties per atomically delivered payload. The most message-efficient previous solutions to the problem of

asynchronous atomic broadcast among n parties (by Castro and Liskov [CL02] and by Kursawe and Shoup [KS05]) have message complexity $\mathcal{O}(n^2)$. While these solutions are certainly useful, given the relatively high message latencies over the Internet, it would be more desirable to have a solution with message complexity $\mathcal{O}(n)$. Linear message complexity appears to be optimal for atomic broadcast, because a protocol needs to send every payload to each party at least once, and this requires n messages (assuming that payloads are not propagated to the parties in batches).

3. *An asynchronous atomic broadcast protocol for the agreement phase of BFT replication that is parsimonious in the number of communication steps involved per atomically delivered payload.* Specifically, we propose an asynchronous, latency-efficient atomic broadcast protocol whose *latency degree* [Sch97] is optimal, i.e., 2. Latency degree of a consensus protocol, or equivalently an atomic broadcast protocol, is the number of communication steps involved in good runs (i.e., failure-free runs) per atomically delivered payload. The most latency-efficient previous solutions to the problem of asynchronous atomic broadcast among n parties (by Castro and Liskov [CL02] and by Kursawe and Shoup [KS05]) have latency degree 3.

4. *A comprehensive suite of protocols that provide dynamic state machine replication but do not make correctness conditional upon the ability to remove members from the group.* Previous asynchronous BFT replication protocols assume static groups, i.e., the number of parties is fixed permanently at system start-up time. However, there are many situations, particularly for long-lived services, in which it is desirable to have the capability to dynamically tune the degree of fault resilience. While the ability to add new parties and remove suspected corrupt parties has been previously explored in the context of Byzantine fault-tolerant group communication systems (e.g., [Rei95][KMMS01][RPL+02]), the group communication protocols of those systems *required* the removal of suspected corrupt nodes in order to make progress and provide

liveness. Since even state-of-the-art intrusion detection systems are unreliable at best, such a requirement opens an easily exploitable vulnerability to denial-of-service attacks. Our replication protocols are designed to withstand t simultaneous corruptions and do not *require* the removal of suspected faulty nodes in order to provide liveness. Such a design allows for the enforcement of very selective and conservative policies regarding changes to the membership of the replication group.

5. *A software toolkit called the* Component-Based Framework for Intrusion Tolerance *or* CoBFIT *that combines the implementation of the parsimonious execution, parsimonious atomic broadcast, and group management protocols within a reusable software framework and that can be used to build trustworthy Internet-scale services.*

6. *Experimental validation of the software toolkit and the protocols in the context of a representative application under both fault-free and controlled fault injections in both LAN and WAN environments.*

The guiding philosophy for the design of our protocols can be summarized in three phrases: *parsimony*, *optimism*, and *worst-case readiness*. Parsimony is to be as frugal as possible for a given metric of interest. Optimism is to hope that we can apply parsimonious techniques for most of the system's lifetime. Worst-case readiness is the ability to handle situations that belie the optimistic hope without compromising correctness. The specific metrics we consider in this dissertation are (1) overall resource use of request execution, (2) message complexity of atomic broadcast, and (3) latency degree of atomic broadcast. However, one can apply our general philosophy to the creation of a distributed protocol that is parsimonious for some other metric as well.

The motivation for our parsimonious protocols is the observation that conditions are "normal" during most of a system's operation. The term "normal conditions" refers to the state of an underlying network that is relatively stable in terms of the message transmission delays. However, the complete specification of normal conditions will vary for different

```
Parsimonious Mode
  hope that conditions are "normal" {specification of normal conditions protocol-specific}
  for a given metric of interest, $\mathcal{M}$, use an approach that,
    when the hope is met, provides protocol functionality with optimal $\mathcal{M}$, and
    when the hope is not met, may not make progress but never violates safety
  determine inability to make progress using failure detection; switch to recovery mode

Recovery Mode
  ensure progress
  reconfigure based on latest failure detection information {redefine "normal" conditions}
  switch back to parsimonious mode for next epoch
```

Figure 1.1: The Informal Generic Structure of a Parsimonious Protocol

parsimonious protocols. For example, in the case of our parsimonious atomic broadcast protocol, the specification includes a leader that is not actively misbehaving. On the other hand, in the case of our parsimonious execution protocol, the specification includes a subset of parties (described later as the *primary committee*) that are not actively misbehaving.

Figure 1.1 informally presents the generic structure of a parsimonious protocol. A parsimonious protocol operates in epochs, with each epoch consisting of a parsimonious mode and a recovery mode. Under conditions that are perceived to be normal, a parsimonious protocol operates in the parsimonious mode of an epoch. For a given metric of interest $\mathcal{M}$, the parsimonious mode employs an approach that provides the protocol functionality with optimal $\mathcal{M}$. Under conditions that are indicative of failures or instability, the parsimonious mode may not be able to make progress, but never violates safety. Indications of failures and instability are obtained based on failure detection mechanisms. Under such conditions, the protocol switches to a more expensive recovery mode, which ensures that some progress is eventually made, after which the protocols switch back to the parsimonious mode for the next epoch. Failure detection mechanisms may be based on timing assumptions. While inaccuracy of these timing assumptions may affect the protocols' ability to provide parsimony, the protocol is designed such that those assumptions never affect the protocol's ability to provide safety and liveness. Since practical observations show that system behavior alternates between long periods of stable conditions and relatively short periods of instability, the hope that timing assumptions based on stable conditions have a high likelihood of being accurate is realistic. Hence, one can expect the parsimonious protocol to have much better average efficiency characteristics than the status quo.

1.5 Thesis Organization

The rest of the dissertation is organized as follows.

Chapter 2 presents the overall system architecture and describes how the various protocols fit together. It also presents the formal system model and assumptions for which our protocols were designed.

Chapter 3 first introduces the primitives on which our atomic broadcast protocol relies. Then, it defines atomic broadcast and presents our parsimonious atomic broadcast protocol. The chapter includes an analysis of the protocol, a discussion on the practical significance of the protocol, and a comparison with related work.

Chapter 4 introduces an abstraction of the agreement (or atomic broadcast) phase that simplifies the presentation of the execution protocol. The parsimonious execution protocol is then presented and analyzed. The chapter includes a discussion on some practical applications for the protocol and a comparison with related work.

Chapter 5 presents the various protocols for the dynamic membership management of the replication group, namely admission control, failure detection, group membership agreement, and reconfiguration. The reconfigurable versions (as opposed to the static versions presented in Chapters 3 and 4) of the atomic broadcast and execution protocols are also presented. The chapter specifies the properties provided by the integrated suite of protocols, and includes a discussion on how the group management capabilities can be utilized without opening denial-of-service vulnerabilities.

Chapter 6 describes the implementation of the CoBFIT software toolkit. First, it presents the framework components that form the foundation upon which protocol components are built. It describes the implementation of the consistent broadcast protocol and agreement protocols that serve as primitives for implementing the higher-level protocols, such as the atomic broadcast protocol and the group membership agreement protocol. Then, it describes the components that implement the parsimonious atomic broadcast, parsimonious execution, and dynamic membership management protocols. Finally, it describes the implementation of components that interface with the replicated service and the clients of the replicated service.

Chapter 7 presents the experimental evaluation of our protocols in both LAN and WAN settings under both fault-free and controlled fault injection scenarios. The protocols were evaluated in the context of a representative application, namely the Fractal Generator, that generates image files based on input fractal parameters.

Finally, Chapter 8 reviews research accomplishments and discusses future work.

Chapter 2

System Overview

In this chapter, we describe the system architecture at a high level and also present the system model and assumptions under which our protocols provide their specified properties.

2.1 Overall System Architecture

The types of services that can benefit from our protocols are those that can be modeled as deterministic state machines [Lam78, Sch90]. These services can have operations that perform arbitrary computations, but the computations must be *deterministic*, i.e., only the current service state and the input parameters to a requested operation determine the result and new service state produced due to the execution of the operation. It is possible to use the state machine replication approach to obtain a BFT service by developing multiple versions of the state machine that represents the service and deploying the versions or replicas on the distinct nodes of a distributed system. It is permissible for the replicas to exhibit non-deterministic behavior as long as the externally observable behaviors of the replicas are the same, i.e., the same sequence of operations issued at any two correct replicas must produce the same sequence of results. It is essential to minimize common-mode failure among replicas through replica diversity. Several approaches have been suggested in the literature for obtaining replica diversity, such as deployment of the replicas in heterogeneous operating systems [SCS03] and opportunistic N-version programming through diverse COTS implementations of an abstract interface [RCL01].

In state machine replication, maintaining consistency of the service state at all correct

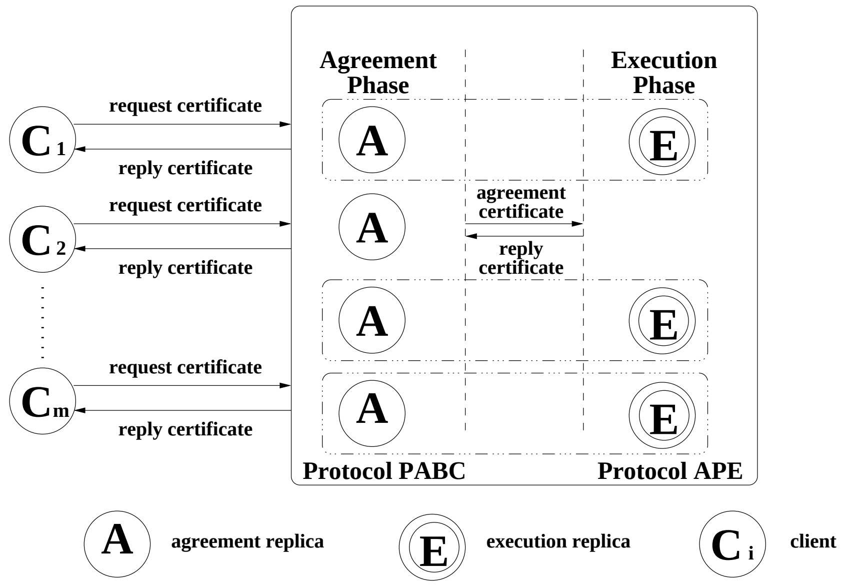


Figure 2.1: State Machine Replication Using Protocols PABC and APE

replicas is a key concern that is addressed by enforcing a total order on the client requests to the service and then executing the requests in exactly that order at all correct replicas. Our approach (just like Yin et al.'s approach [YMV$^+$03]) is to separate agreement on the order of requests to execute from the actual execution of the requests (Figure 2.1). In order to achieve a fault resilience of t, there must be at least $3t + 1$ distinct participants in the agreement phase and at least $2t + 1$ distinct participants in the execution phase. Figure 2.1 depicts the situation for $t = 1$ in which there are $3t + 1 = 4$ participants in the agreement phase and $2t + 1 = 3$ participants in the execution phase. Although agreement phase participants and execution phase participants may be placed in distinct nodes, physical separation is not necessary; a replica may perform a dual role in which it participates in both the agreement and execution phases while also maintaining a logical separation between the two phases. In the rest of this thesis, we consider such a model of logical separation between the agreement and execution phases. All the n replicas participate in the agreement phase, and at least $n - t$ of them participate in the execution phase.

Figure 2.1 shows at a high level how a client request is processed by the replicated service. In the figure, the term *request certificate* refers to an operation requested by the client along with proof of authorization showing that the client is indeed permitted to request that operation.

Agreement phase participants execute Protocol PABC (which stands for *parsimonious asynchronous atomic broadcast*), a message-efficient atomic broadcast protocol. Request certificates form the input to Protocol PABC and atomic deliveries form the output. An atomic delivery binds a request certificate to a unique sequence number, such that all correct agreement phase participants will eventually output the same binding. The sequence of atomic deliveries defines a total order on the request certificates.

Agreement certificates, which are atomic deliveries from the local Protocol PABC, form the input to the Protocol APE, which runs at all replicas that participate in the execution phase. Protocol APE (which stands for *asynchronous parsimonious execution*) is a resource-efficient execution protocol through which requests are executed in the sequence number order defined by the agreement certificates. At an execution replica, execution of a request with sequence number s is always preceded by an update of the service state to reflect execution of all requests with sequence numbers lower than s. Reply messages carrying the results of request processing and the sequence number of the processed request form the output of Protocol APE; the messages are sent to the agreement phase participants. A set of reply messages from $t + 1$ distinct execution phase participants that contain the same result and sequence number constitute a *reply certificate*.

Reply certificates from the execution phase serve as acknowledgments to the agreement certificates from the agreement phase and are forwarded to the appropriate client. The execution phase participants implement deterministic state machines and execute requests in the total order defined by agreement certificates; hence, the result for a given sequence-numbered request will be the same at any correct execution phase participant. A reply certificate carries the result of request processing from at least one correct execution phase participant. Once the client obtains a reply certificate, the client accepts the result value contained in the certificate as the valid response to its request certificate.

2.2 System Model

In this section, we describe the system model and assumptions for the static versions (i.e., those in which the group of agreement and execution replicas is fixed) of our parsimonious protocols described in Chapters 3 and 4. Later, in Chapter 5, we extend the system model for the reconfigurable versions of the protocols in which replicas can be removed or added to the replication group. We use the terms "replica" and "party" interchangeably.

We consider an asynchronous distributed system model equivalent to the one of Cachin et al. [CKPS01], in which there are no bounds on relative processing speeds and message delays. The BFT-replicated service consists of n replicas $P_1, \ldots, P_n$. All the replicas participate in the agreement phase. However, only the first $n-t$ replicas (i.e., $P_1, \ldots, P_{n-t}$) participate in the execution phase. We use the term *agreement replica* to denote the part of a replica that participates in the agreement phase. Similarly, we use the term *execution replica* to denote the part of a replica that participates in the execution phase.

Clients of the service and replicas occupy different nodes. The execution replicas start with the same initial service state and implement deterministic state machines. Up to $t < \frac{n}{3}$ parties may be controlled by an *adversary*. We call such parties *corrupted*; the other parties are called *correct*. We use a *static* corruption model, which means that the adversary must pick the parties it corrupts once and for all before starting the protocol. We assume that the basic unit of corruption is a node. Components or protocols co-located in the same node trust each other to provide the specified functionality. There is also an initialization algorithm that is run by some trusted *dealer* that performs system setup before the start of the protocol. All computations by the parties, the adversary, and the trusted dealer are probabilistic, polynomial-time algorithms. The parameters n and t are given as input to the dealer, which then generates the state information that is used to initialize each party. Note that after the initial setup phase, the static versions of the parsimonious protocols have no need for the dealer.

Each pair of parties is linked by an *authenticated asynchronous channel* that provides message integrity (e.g., using message authentication codes [VvOM96]). 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[1]. We refer to this property in liveness conditions when we say that a message is *eventually* delivered or that a protocol instance *eventually* terminates.

There may be multiple protocol instances that are concurrently executing at each party. A protocol instance is invoked either by a higher-level protocol instance or by the adversary. Every protocol instance is identified by a unique string *ID*, called the *tag*, which is chosen by the entity that invokes the instance. The tag of a sub-protocol instance contains the tag of the calling instance as a prefix.

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, which are given to the adversary. There may be several threads of execution for a given party, but only one of them is allowed to be active at any one 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 that are in wait states have their conditions simultaneously 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

[1]A more restrictive formalization of liveness conditions for environments with a computationally bounded scheduler is provided by Cachin et al. [CKPS01] through the notion of *probabilistically uniformly bounded statistics*; this notion can be easily applied to our protocols with some modifications, but we refrain from using it for the sake of readability.

control returns to the adversary.

There are three types of messages that appear in the interface to our protocols, namely (1) *input actions*, which are messages of the form $(ID, \mathtt{in}, \mathit{type}, \ldots)$; (2) *output actions*, which are messages of the form $(ID, \mathtt{out}, \mathit{type}, \ldots)$; and (3) *protocol messages*, which are ordinary protocol messages to be delivered to other parties of the form $(ID, \mathit{type}, \ldots))$. Note that input actions and output actions are local events within a party. Before a party starts to process messages that are tagged with ID, the instance must be *initialized* by a special input action of the form $(ID, \mathtt{in}, \mathtt{open}, \mathit{type})$, where *type* denotes the protocol type and/or its implementation. Such an action must precede any other input action with tag ID. We usually assume that it occurs implicitly with the first regular input action.

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 the system initialization, the dealer generates (using the key generation algorithm) the public key/private key pair for each party and gives every party 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., secure against existential forgery using chosen-message attacks [GMR88]. Since we use the formal model of modern cryptography [Gol04], we allow for a negligible probability of failure in the specification of our protocols.

Chapter 3

Parsimonious Atomic Broadcast

Atomic broadcast is a communication primitive that allows a group of n parties to deliver a common sequence of payload messages despite the failure of some parties. In this chapter, we address the problem of asynchronous atomic broadcast when up to $t < n/3$ parties may exhibit Byzantine behavior.

We provide two protocols, one with an amortized expected message complexity of $\mathcal{O}(n)$ per delivered payload and the other with a latency degree of 2 per delivered payload. The most efficient previous solutions are the BFT protocol by Castro and Liskov and the KS protocol by Kursawe and Shoup, both of which have message complexity $\mathcal{O}(n^2)$ and latency degree 3. Like the BFT and KS protocols, our protocol is leader-based and optimistically hopes that the leader is not actively misbehaving; when network instability or leader misbehavior is detected, it switches to a more expensive recovery mode. The key idea of our solution is to replace reliable broadcast in the KS protocol by consistent broadcast. We propose two variants of consistent broadcast; using one variant reduces the message complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ in the optimistic mode, while the other variant reduces the latency degree from 3 to 2 in the optimistic mode. But since consistent broadcast provides weaker guarantees than reliable broadcast, our recovery mode incorporates novel techniques to ensure that safety and liveness are always satisfied.

3.1 Introduction

Atomic broadcast is a fundamental communication primitive for the construction of fault-tolerant distributed systems. It allows a group of n parties to agree on a set of payload messages to deliver and also on their delivery order, despite the failure of up to t parties. It is possible to construct a fault-tolerant service using the state machine replication approach [Sch90] by replicating the service on all n parties and propagating the state updates to the replicas using atomic broadcast. In this chapter, we present a new atomic broadcast protocol that is suitable for building highly available and intrusion-tolerant services in the Internet [Cac01][SZ04].

Though the problem of Byzantine-fault-tolerant atomic broadcast and the equivalent problem of Byzantine agreement have been widely studied for over two decades, much of the previous work (described in Section 1.3) was either intended to demonstrate only theoretical feasibility or based correctness on assumptions that are easy to invalidate (e.g., synchrony). Hence, the applicability of that work for our purpose is quite limited.

The BFT protocol by Castro and Liskov [CL02] and the protocol by Kursawe and Shoup [KS05] (hereafter referred to as the KS protocol) take an optimistic approach for providing more efficient asynchronous atomic broadcast while never violating safety. Both protocols proceed in *epochs*, where an epoch consists of an *optimistic mode* and a *recovery mode*, and expect to spend most of their time operating in the optimistic mode, which uses an inexpensive mechanism that is appropriate for normal conditions. The protocol switches to the more expensive recovery mode under unstable network or certain fault conditions. In every epoch, a designated party acts as a *leader* for the optimistic mode, determines the delivery order of the payloads, and conveys the chosen delivery order to the other parties through Bracha's reliable broadcast protocol [Bra84], which guarantees delivery of a broadcast payload with the same content at all correct parties. Bracha's protocol is deterministic and involves $\mathcal{O}(n^2)$ protocol messages; it is much more efficient than the most efficient randomized Byzantine agreement protocol [CKPS01], which requires expensive public-key cryptographic operations in addition. Consequently, both the BFT and KS protocols communicate $\mathcal{O}(n^2)$ messages per atomically delivered payload under normal conditions, i.e., they have *message complexity* $\mathcal{O}(n^2)$. Bracha's reliable broadcast protocol involves 3 communication steps, i.e., latency degree 3. Consequently, both the BFT and KS protocols involve 3 communication steps per atomically delivered payload under normal conditions.

No optimally resilient protocol for asynchronous atomic broadcast with message complexity less than $\Theta(n^2)$ or latency degree less than 3 was known prior to our work. We present a protocol for asynchronous atomic broadcast that is the first to achieve optimal resilience $t < n/3$ and $\mathcal{O}(n)$ amortized expected message complexity. We also present a variant protocol that is the first to achieve optimal resilience $t < n/3$ and latency degree 2.

Like the BFT and KS protocols, our protocols are *optimistic* in the sense that they progress very fast during periods when the network is reasonably well-behaved and a party acting as the designated *leader* is correct. Unlike the BFT protocol (and just like the KS protocol), our protocols guarantee both *safety* and *liveness* in asynchronous networks by relying on randomized agreement. The reduced message complexity and the reduced latency degree of our protocols come at the cost of introducing a digital signature computation for every delivered payload. But in a wide-area network (WAN), the cost of a public-key operation is small compared to message latency. And since our protocols are targeted at WANs, we expect the advantage of lower message complexity to outweigh the additional work incurred by the signature computations.

The key idea in both our solutions is to replace reliable broadcast used in the optimistic mode of the BFT and KS protocols with *consistent broadcast*, also known as *echo broadcast* [Rei94]. We use the term *parsimonious mode* to denote the optimistic mode of our atomic broadcast protocols because of the parsimony (either in message complexity or latency degree) achieved and to adhere to the generic parsimonious protocol structure described in Section 1.4. Consistent broadcast is a weaker form of reliable broadcast that guarantees agreement only among those correct parties that actually deliver the payload, but it is possible that some correct parties do not deliver any payload at all. But the replacement also complicates the recovery mode, since a corrupted leader might cause the payload to be consistently delivered at only a single correct party with no way for other correct parties to learn about this fact. Our protocols provide mechanisms to address such complications.

Our message-efficient atomic broadcast protocol and the latency-efficient atomic broadcast protocol are distinguished only by the type of consistent broadcast implementation they use in the parsimonious mode. Both consistent broadcast implementations provide the same properties. We describe the two types of consistent broadcast implementations; however, for the sake of simplicity, we present only the message-efficient atomic broadcast protocol in detail. The latency-efficient atomic broadcast protocol is obtained simply by replacing one consistent broadcast implementation with the other.

Our protocols are related to the reliable broadcast protocol of Malkhi et al. [MMR00] in their use of consistent broadcast as a building block. Malkhi et al.'s protocol addresses reliable broadcast over a WAN, but provides no total order.

The rest of the chapter is organized as follows. Section 3.2 describes the protocol primitives on which our algorithm relies, and the definition of atomic broadcast. The protocol is presented in Section 3.3 and analyzed in Section 3.4. Section 3.5 discusses the practical significance of our parsimonious protocol and compares it with related work. Finally, Section 3.7 summarizes the chapter.

3.2 Preliminaries

3.2.1 Protocol Primitives

Our atomic broadcast protocol relies on a consistent broadcast protocol with special properties and on a Byzantine agreement protocol.

Strong Consistent Broadcast

We enhance the notion of consistent broadcast found in the literature [CKPS01] to develop the notion that we call *strong consistent broadcast.* Ordinary consistent broadcast provides a way for a designated sender P_s to broadcast a payload to all parties and requires that any two correct parties that deliver the payload agree on its content.

The standard protocol for implementing ordinary consistent broadcast is Reiter's *echo broadcast* [Rei94]; it involves $\mathcal{O}(n)$ messages, has a latency of three message flows, and relies on a digital signature scheme. The sender starts the protocol by sending the payload m to all parties; then it waits for a quorum of $\lceil \frac{n+t+1}{2} \rceil$ parties to issue a signature on the payload and to "echo" the payload and the signature to the sender. When the sender has collected and verified enough signatures, it composes a final protocol message containing the signatures and sends it to all parties.

With a faulty sender, an ordinary consistent broadcast protocol permits executions in which some parties fail to deliver the payload when others succeed. Therefore, a useful enhancement of consistent broadcast is a transfer mechanism, which allows any party that has delivered the payload to help others do the same.

For reasons that will be evident later, we introduce another enhancement and require that when a correct party terminates a consistent broadcast and delivers a payload, there must be a quorum of at least $n - t$ parties (instead of only $\lceil \frac{n+t+1}{2} \rceil$) who participated in the protocol and approved the delivered payload. We call consistent broadcast with such a transfer mechanism and the special quorum rule *strong consistent broadcast.*

Formally, every broadcast instance is identified by a tag ID. At the sender P_s, strong consistent broadcast is invoked by an input action of the form $(ID, \texttt{in}, \texttt{sc-broadcast}, m)$, with $m \in \{0,1\}^*$. When that occurs, we say *P_s sc-broadcasts m with tag ID*. Only P_s executes this action; all other parties start the protocol only when they initialize instance ID in their role as receivers. A party terminates a consistent broadcast of m tagged with ID by generating an output action of the form $(ID, \texttt{out}, \texttt{sc-deliver}, m)$. In that case, we say P_i *sc-delivers* m *with tag* ID.

For the transfer mechanism, a correct party that has *sc-delivered* m with tag ID should be able to output a bit string M_{ID} that *completes* the *sc-broadcast* in the following sense: any correct party that has not yet *sc-delivered* m can run a *validation algorithm* on M_{ID} (this may involve a public key associated with the protocol), and if M_{ID} is determined to be *valid*, it can also *sc-deliver* m from M_{ID}.

Definition 1 (Strong consistent broadcast) *A protocol for strong consistent broadcast satisfies the following conditions except with negligible probability.*

Termination: *If a correct party* sc-broadcasts m *with tag* ID*, then all correct parties eventually* sc-deliver m *with tag* ID.

Agreement: *If two correct parties* P_i *and* P_j sc-deliver m *and* m' *with tag* ID*, respectively, then* $m = m'$.

Integrity: *Every correct party* sc-delivers *at most one payload* m *with tag* ID*. Moreover, if the sender* P_s *is correct, then* m *was previously* sc-broadcast *by* P_s *with tag* ID.

Transferability: *After a correct party has* sc-delivered m *with tag* ID*, it can generate a string* M_{ID} *such that any correct party that has not* sc-delivered *a message with tag* ID*is able to* sc-deliver *some message immediately upon processing* M_{ID}.

Strong unforgeability: *For any* ID*, it is computationally infeasible to generate a value* M *that is accepted as valid by the validation algorithm for completing* ID *unless* $n - 2t$ *correct parties have initialized instance* ID *and actively participated in the protocol.*

Note that the termination, agreement, and integrity properties are the same as in ordinary consistent broadcast [Rei94][CKPS01].

Given the above implementation of consistent broadcast, one can obtain strong consistent broadcast with two simple modifications. The completing string M_{ID} for ensuring transferability consists of the final protocol message; the attached signatures are sufficient for any other party to complete the *sc-broadcast*. Strong unforgeability is obtained by setting the signature quorum to $n - t$.

With signatures of size K bits, the echo broadcast protocol has communication complexity $\mathcal{O}\big(n(|m| + nK)\big)$ bits, where $|m|$ denotes the bit length of the payload m. By replacing the quorum of signatures with a threshold signature [Des88], it is possible to reduce the communication complexity to $\mathcal{O}\big(n(|m| + K)\big)$ bits [CKPS01], under the reasonable assumption that the lengths of a threshold signature and a signature share are also at most K bits [Sho00].

In the rest of the chapter, we assume that strong consistent broadcast is implemented by applying the above modifications to the echo broadcast protocol with threshold signatures. Hence, the length of a completing string is $\mathcal{O}(|m| + K)$ bits.

Multi-Valued Byzantine Agreement

We use a protocol for multi-valued Byzantine agreement (MVBA) as defined by Cachin et al. [CKPS01], which allows agreement values from an arbitrary domain instead of being restricted to binary values. Unlike previous multi-valued Byzantine agreement protocols, their protocol does not allow the decision to fall back on a *default* value if not all correct parties propose the same value, but uses a protocol-external mechanism instead. This so-called *external validity condition* is specified by a global, polynomial-time computable predicate Q_{ID}, which is known to all parties and is typically determined by an external application or higher-level protocol. Each party proposes a value that contains certain validation information. The protocol ensures that the decision value was proposed by at least one party, and that the decision value satisfies Q_{ID}.

When a party P_i starts an MVBA protocol instance with tag ID and an input value $v \in \{0,1\}^*$ satisfying predicate Q_{ID}, we say that P_i *proposes v for multi-valued agreement with tag ID and predicate Q_{ID}*. Correct parties only propose values that satisfy Q_{ID}. When P_i terminates the MVBA protocol instance with tag ID and outputs a value v, we say that it *decides* v *for* ID.

Definition 2 (Multi-valued Byzantine agreement) *A protocol for* multi-valued Byzantine agreement *with predicate* Q_{ID} *satisfies the following conditions except with negligible probability.*

External Validity: *Any correct party that decides for* ID *decides* v *such that* $Q_{ID}(v)$ *holds.*

Agreement: *If some correct party decides* v *for* ID*, then any correct party that decides for* ID *decides* v.

Integrity: *If all parties are correct and if some party decides* v *for* ID*, then some party proposed* v *for* ID.

Termination: *All correct parties eventually decide for* ID.

The MVBA protocol of Cachin et al. [CKPS01] builds upon a protocol for binary Byzantine agreement (such as the one of Cachin et al. [CKS05]), which relies on threshold signatures and a threshold coin-tossing protocol (e.g., [CKS05]). The expected message complexity of the MVBA protocol is $\mathcal{O}(n^2)$ and the expected communication complexity is $\mathcal{O}\big(n^3 + n^2(K + L)\big)$, where K is the length of a threshold signature and L is a bound on the length of the values that can be proposed.

3.2.2 Definition of Atomic Broadcast

Atomic broadcast provides a "broadcast channel" abstraction [HT93], such that all correct parties deliver the same set of messages broadcast on the channel in the same order. A party P_i *atomically broadcasts* (or *a-broadcasts*) a payload m with tag ID when an input action of the form $(ID, \texttt{in}, \texttt{a-broadcast}, m)$ with $m \in \{0,1\}^*$ is delivered to P_i. Broadcasts are parameterized by the tag ID to identify their corresponding broadcast channel. A party *atomically delivers* (or *a-delivers*) a payload m with tag ID by generating an output action of the form $(ID, \texttt{out}, \texttt{a-deliver}, m)$. A party may *a-broadcast* and *a-deliver* an arbitrary number of messages with the same tag.

Definition 3 (Atomic broadcast) *A protocol for atomic broadcast satisfies the following properties except with negligible probability.*

Validity: *If $t+1$ correct parties* a-broadcast *some payload m with tag ID, then some correct party eventually* a-delivers *m with tag ID.*

Agreement: *If some correct party has* a-delivered *m with tag ID, then all correct parties eventually* a-deliver *m with tag ID.*

Total Order: *If two correct parties both* a-delivered *distinct payloads m_1 and m_2 with tag ID, then they have* a-delivered *them in the same order.*

Integrity: *For any payload m, a correct party P_j* a-delivers *m with tag ID at most once. Moreover, if all parties are correct, then m was previously* a-broadcast *by some party with tag ID.*

The above properties are similar to the definitions of Cachin et al. [CKPS01] and of Kursawe and Shoup [KS05]. We do not formalize their *fairness* condition, which requires that the protocol "makes progress" towards delivering a payload as soon as $t + 1$ correct parties have *a-broadcast* it. However, our protocol actually satisfies an equivalent notion (cf., Lemma 4).

To analyze our protocol and to compare it with related protocols in the literature, we use two measures, message complexity and communication complexity. The message complexity of a protocol instance with tag *ID* is defined as the total number of all protocol messages with the tag *ID* or any tag starting with $ID|\ldots$ that correct parties generate. The communication complexity of a protocol instance with tag *ID* is defined as the total bit length of all protocol messages with the tag *ID* or any tag starting with $ID|\ldots$ that correct parties generate.

3.3 The Parsimonious Asynchronous Atomic Broadcast Protocol

3.3.1 Overview

The starting point for the development of our Protocol PABC is the BFT protocol [CL02], which can be seen as the adaptation of Lamport's Paxos consensus protocol [Lam98] to tolerate Byzantine faults. In the BFT protocol, a leader determines the delivery order of payloads and conveys the order using reliable broadcast to other parties. The parties then atomically deliver the payloads in the order chosen by the leader. If the leader appears to be slow or exhibits faulty behavior, a party switches to the recovery mode. When enough correct parties have switched to recovery mode, the protocol ensures that all correct parties eventually start the recovery mode. The goal of the recovery mode is to start the next epoch in a consistent state and with a new leader. The difficulty lies in determining which payloads have been delivered in the optimistic mode of the past epoch. The BFT protocol delegates this task to the leader of the new epoch. But since the recovery mode of BFT is also deterministic, it may be that the new leader is evicted immediately, before it can do any useful work, and the epoch passes without delivering any payloads. This denial-of-service attack against the BFT protocol violates liveness but is unavoidable in asynchronous networks.

The KS protocol [KS05] prevents this attack by ensuring that at least one payload is delivered during the recovery mode. It employs a round of randomized Byzantine agreement to agree on a set of payloads for atomic delivery, much like the asynchronous atomic broadcast protocol of Cachin et al. [CKPS01]. During the optimistic mode, the epoch leader conveys the delivery order through reliable broadcast as in BFT, which leads to an amortized message complexity of $\mathcal{O}(n^2)$.

Our approach is to replace reliable broadcast in the KS protocol with strong consistent broadcast; the replacement directly leads to an amortized message complexity of only $\mathcal{O}(n)$.

But the replacement also introduces complications in the recovery mode, since a corrupted leader may cause the fate of some payloads to be undefined in the sense that there might be only a single correct party that has *sc-delivered* a payload, but no way for other correct parties to learn about this fact. We solve this problem by delaying the atomic delivery of an *sc-delivered* payload until more payloads have been *sc-delivered*. However, the delay introduces an additional problem of payloads getting "stuck" if no further payloads arrive. We address this by having the leader generate *dummy* payloads when no further payloads arrive within a certain time window.

The recovery mode in our protocol has a structure similar to that of the KS protocol, but is simpler and more efficient. At a high level, a first MVBA instance ensures that all correct parties agree on a synchronization point. Then, the protocol ensures that all correct parties *a-deliver* the payloads up to that point; to implement this step, every party must store all payloads that were delivered in the parsimonious mode, together with information that proves the fact that they were delivered. A second MVBA instance is used to *a-deliver* at least one payload, which guarantees that the protocol makes progress in every epoch.

3.3.2 Details

We now describe the parsimonious and the recovery modes in detail. The line numbers refer to the detailed protocol description in Figures 3.1–3.4.

Parsimonious Mode

Every party keeps track of the current epoch number e and stores all payloads that it has received to *a-broadcast* but not yet *a-delivered* in its *initiation queue* $\mathcal{I}$. An element x can be appended to $\mathcal{I}$ by an operation $append(x, \mathcal{I})$, and an element x that occurs anywhere in $\mathcal{I}$ can be removed by an operation $remove(x, \mathcal{I})$. A party also maintains an array *log* of size B that acts as a buffer for all payloads to *a-deliver* in the current epoch. Additionally, a party stores a set $\mathcal{D}$ of all payloads that have been *a-delivered* so far.

```
Protocol PABC for party $P_i$ and tag $ID$

initialization:
 1: $e \leftarrow 0$                                              {current epoch}
 2: $\mathcal{I} \leftarrow []$        {initiation queue, list of a-broadcast but not a-delivered payloads}
 3: $\mathcal{D} \leftarrow \emptyset$                             {set of a-delivered payloads}
 4: $init\_epoch()$

function $init\_epoch()$:
 5: $l \leftarrow (e \bmod n) + 1$                                 {$P_l$ is leader of epoch $e$}
 6: $log \leftarrow []$        {array of size $B$ containing payloads committed in current epoch}
 7: $s \leftarrow 0$                         {sequence number of next payload within epoch}
 8: $complained \leftarrow$ false        {indicates if this party already complained about $P_l$}
 9: $start\_recovery \leftarrow$ false              {signals the switch to the recovery mode}
10: $c \leftarrow 0$                {number of complain messages received for epoch leader}
11: $\mathcal{S} \leftarrow \mathcal{D}$         {set of a-delivered or already sc-broadcast payloads at $P_l$}

forever: {parsimonious mode}
12: if $\neg complained$ then {leader $P_l$ is not suspected}
13:   initialize an instance of strong consistent broadcast with tag $ID|$bind$.e.s$
14: $m \leftarrow \perp$
15: if $i = l$ then
16:   wait for $timeout(T)$ or receipt of a message $(ID,$ initiate$, e, m)$ such that $m \notin \mathcal{S}$
17:   if $timeout(T)$ then
18:     $m \leftarrow$ dummy
19:   else
20:     $\mathcal{S} \leftarrow \mathcal{S} \cup \{m\}$
21:     $stop(T)$
22:   sc-broadcast the message $m$ with tag $ID|$bind$.e.s$
23: wait for $start\_recovery$ or sc-delivery of some $m$ with tag $ID|$bind$.e.s$
      such that $m \notin \mathcal{D} \cup log$
24: if $start\_recovery$ then
25:   $recovery()$
26: else
27:   $log[s] \leftarrow m$
28:   if $s \geq 2$ then
29:     $update_{\mathcal{F}_l}($deliver$, log[s-2])$
30:     $deliver(log[s-2])$
31:   if $i = l$ and $(log[s] \neq$ dummy or $(s > 0$ and $log[s-1] \neq$ dummy$))$ then
32:     $start(T)$
33:   $s \leftarrow s + 1$
34:   if $s \bmod B = 0$ then
35:     $recovery()$
```

Figure 3.1: Protocol PABC for Atomic Broadcast (Part I)

```
upon $(ID, \texttt{in}, \texttt{a-broadcast}, m)$:
        36: send $(ID, \texttt{initiate}, e, m)$ to $P_l$
        37: $append(m, \mathcal{I})$
        38: $update_{\mathcal{F}_l}(\texttt{initiate}, m)$

function $deliver(m)$:
        39: if $m \neq \texttt{dummy}$ then
        40:     $remove(m, \mathcal{I})$
        41:     $\mathcal{D} \leftarrow \mathcal{D} \cup \{m\}$
        42:     output $(ID, \texttt{out}, \texttt{a-deliver}, m)$

function $complain()$:
        43: send $(ID, \texttt{complain}, e)$ to all parties
        44: $complained \leftarrow \texttt{true}$

upon receiving message $(ID, \texttt{complain}, e)$ from $P_j$ for the first time:
        45: $c \leftarrow c + 1$
        46: if $(c = t + 1)$ and $\neg complained$ then
        47:     $complain()$
        48: else if $c = 2t + 1$ then
        49:     $start\_recovery \leftarrow \texttt{true}$

let $Q_{ID|\texttt{watermark}.e}$ be the following predicate:
        $Q_{ID|\texttt{watermark}.e}\Big(\big[(s_1, C_1, \sigma_1), \ldots, (s_n, C_n, \sigma_n)\big]\Big) \equiv$
            $\big($for at least $n - t$ distinct $j$, $s_j \neq \perp\big)$ and
                $\big($for all $j = 1, \ldots, n$, either $s_j = \perp$ or
                    $(\sigma_j$ is a valid signature by $P_j$ on $(ID, \texttt{committed}, e, s_j, C_j)$ and
                        $(s_j = -1$ or $C_j$ completes the $sc\text{-}broadcast$ with tag $ID|\texttt{bind}.e.s_j))\big)$

Let $Q_{ID|\texttt{deliver}.e}$ be the following predicate:
        $Q_{ID|\texttt{deliver}.e}\Big(\big[(\mathcal{I}_1, \sigma_1), \ldots, \mathcal{I}_n, \sigma_n)\big]\Big) \equiv$ for at least $n - t$ distinct $j$,
                $\big(\mathcal{I}_j \cap \mathcal{D} = \emptyset$ and $\sigma_j$ is a valid signature by $P_j$ on $(ID, \texttt{queue}, e, j, \mathcal{I}_j)\big)$
```

Figure 3.2: Protocol PABC for Atomic Broadcast (Part II)

```
function recovery():
    {Part 1: agree on watermark}
    50: compute a signature $\sigma$ on $(ID, \texttt{committed}, e, s-1)$
    51: send the message $(ID, \texttt{committed}, e, s-1, C, \sigma)$ to all parties, where $C$ denotes
            the bit string that completes the sc-broadcast with tag $ID|\texttt{bind}.e.(s-1)$
    52: $(s_j, C_j, \sigma_j) \leftarrow (\bot, \bot, \bot)$        $(1 \le j \le n)$
    53: wait for $n-t$ messages $(ID, \texttt{committed}, e, s_j, C_j, \sigma_j)$ from distinct $P_j$ such that $C_j$ completes
            the sc-broadcast instance $ID|\texttt{bind}.e.s_j$ and $\sigma_j$ is a valid signature on $(ID, \texttt{committed}, e, s_j)$
    54: $W \leftarrow [(s_1, C_1, \sigma_1), \ldots, (s_n, C_n, \sigma_n)]$
    55: propose $W$ for multi-valued Byzantine agreement with tag $ID|\texttt{watermark}.e$
            and predicate $Q_{ID|\texttt{watermark}.e}$
    56: wait for the Byzantine agreement protocol with tag $ID|\texttt{watermark}.e$
            to decide some $\bar{W} = [(\bar{s}_1, \bar{C}_1, \bar{\sigma}_1), \ldots, (\bar{s}_n, \bar{C}_n, \bar{\sigma}_n)]$
    57: $w \leftarrow \max\{\bar{s}_1, \ldots, \bar{s}_n\} - 1$

    {Part 2: synchronize up to watermark}
    58: $s' \leftarrow s-2$
    59: while $s' \le \min\{s-1, w\}$ do
    60:     if $s' \ge 0$ then
    61:         deliver(log[s'])
    62:     $s' \leftarrow s'+1$
    63: if $s > w$ then
    64:     for $j = 1, \ldots, n$ do
    65:         $u \leftarrow \max\{s_j, \bar{s}_j\}$
    66:         $\mathcal{M} \leftarrow \{M_v\}$ for $v = u, \ldots, w$, where $M_v$ completes the sc-broadcast instance
                $ID|\texttt{bind}.e.v$
    67:         send message $(ID, \texttt{complete}, \mathcal{M})$ to $P_j$
    68: while $s \le w$ do
    69:     wait for a message $(ID, \texttt{complete}, \bar{\mathcal{M}})$ such that $\bar{M}_s \in \bar{\mathcal{M}}$ completes sc-broadcast
                with tag $ID|\texttt{bind}.e.s$
    70:     use $\bar{M}_s$ to sc-deliver some $m$ with tag $ID|\texttt{bind}.e.s$
    71:     deliver(m)
    72:     $s \leftarrow s+1$
```

Figure 3.3: Protocol PABC for Atomic Broadcast (Part III)

```
function $recovery()$: (continued...)
  {$Part\ 3$: deliver some messages}
    73: compute a digital signature $\sigma$ on $(ID, \texttt{queue}, e, i, \mathcal{I})$
    74: send the message $(ID, \texttt{queue}, e, i, \mathcal{I}, \sigma)$ to all parties
    75: $(\mathcal{I}_j, \sigma_j) \leftarrow (\perp, \perp) \qquad (1 \le j \le n)$
    76: wait for $n - t$ messages $(ID, \texttt{queue}, e, j, \mathcal{I}_j, \sigma_j)$ from distinct $P_j$ such that
          $\sigma_j$ is a valid signature from $P_j$ and $\mathcal{I}_j \cap \mathcal{D} = \emptyset$
    77: $Q \leftarrow [(\mathcal{I}_1, \sigma_1), \ldots, (\mathcal{I}_n, \sigma_n)]$
    78: propose $Q$ for multi-valued Byzantine agreement with tag $ID|\texttt{deliver}.e$
          and predicate $Q_{ID|\texttt{deliver}.e}$
    79: wait for the Byzantine agreement with tag $ID|\texttt{deliver}.e$ to decide some
          $\bar{Q} = [(\bar{\mathcal{I}}_1, \bar{\sigma}_1), \ldots, (\bar{\mathcal{I}}_n, \bar{\sigma}_n)]$
    80: for $m \in \bigcup_{j=1}^{n} \bar{\mathcal{I}}_j \setminus \mathcal{D}$, in some deterministic order do
    81:     $deliver(m)$
    82: $init\_epoch()$
    83: for $m \in \mathcal{I}$ do
    84:     send $(ID, \texttt{initiate}, e, m)$ to $P_l$
```

Figure 3.4: Protocol PABC for Atomic Broadcast (Part IV)

We describe the parsimonious mode of Protocol PABC by first detailing the normal protocol operation when the leader functions properly, and then explaining the mechanisms that ensure that the protocol switches to the recovery mode when the leader is not functioning properly.

Normal Protocol Operation. When a party receives a request to *a-broadcast* a payload m, it appends m to $\mathcal{I}$ and immediately forwards m using an `initiate` message to the leader P_l of the epoch, where $l = e \mod n$ (lines 36–38). When this happens, we say P_i *initiates* the payload.

The leader binds sequence numbers to the payloads that it receives in `initiate` messages, and conveys the bindings to the other parties through strong consistent broadcast. For this purpose, all parties execute a loop (lines 12–35) that starts with an instance of strong consistent broadcast (lines 12–23). The leader acts as the sender of strong consistent broadcast and the tag contains the epoch e and a sequence number s. Here, s starts from 0 in every epoch. The leader *sc-broadcasts* the next available initiated payload, and every party waits to *sc-deliver* some payload m. When m is *sc-delivered*, P_i stores it in *log*, but does not yet *a-deliver* it (line 27). At this point in time, we say that P_i has *committed* sequence number s to payload m in epoch e. Then, P_i *a-delivers* the payload to which it has committed the sequence number $s - 2$ (if available, lines 28–30). It increments s (line 33) and returns to the start of the loop.

Delaying the *a-delivery* of the payload committed to s until sequence number $s + 2$ has been committed is necessary to prevent the above problem of payloads whose fate is undefined. However, the delay results in another problem if no further payloads, those with sequence numbers higher than s, are *sc-delivered*. We solve this problem by instructing the leader to send `dummy` messages to eject the original payload(s) from the buffer. The leader triggers such a `dummy` message whenever a corresponding timer T expires (lines 17–18); T is activated whenever one of the current or the preceding sequence numbers was committed to a non-`dummy` payload (lines 31–32), and T is disabled when the leader *sc-broadcasts* a non-`dummy` payload (line 21). Thus, the leader sends at most two `dummy` payloads to eject a non-`dummy` payload.

Failure Detection and Switching to the Recovery Mode. There are two conditions under which the protocol switches to recovery mode: (1) when B payloads have been committed (line 35) and (2) when the leader is not functioning properly. The first condition is needed to keep the buffer *log* bounded and the second condition is needed to prevent a corrupted leader from violating liveness.

To determine if the leader of the epoch performs its job correctly, every party has access to a leader failure detector $\mathcal{F}_l$. For simplicity, Figures 3.1–3.4 do not include the pseudocode for $\mathcal{F}_l$. The protocol provides an interface *complain*(), which $\mathcal{F}_l$ can asynchronously invoke to notify the protocol about its suspicion that the leader is corrupted. Our protocol synchronously invokes an interface $update_{\mathcal{F}_l}$ of $\mathcal{F}_l$ to convey protocol-specific information (during execution of the $\mathit{update}_{\mathcal{F}_l}$ call, $\mathcal{F}_l$ has access to all variables of Protocol PABC).

An implementation of $\mathcal{F}_l$ can check whether the leader is making progress based on a timeout and protocol information as follows. Recall that every party maintains a queue $\mathcal{I}$ of initiated but not yet *a-delivered* payloads. When P_i has initiated some m, it calls $\mathit{update}_{\mathcal{F}_l}(\texttt{initiate}, m)$ (line 38); this starts a timer $T_{\mathcal{F}_l}$ unless it is already activated. When a payload is *a-delivered* during the parsimonious mode, the call to $\mathit{update}_{\mathcal{F}_l}(\texttt{deliver}, m)$ (line 29) checks whether the *a-delivered* payload is the first undelivered payload in $\mathcal{I}$, and if it is, disables $T_{\mathcal{F}_l}$. When $T_{\mathcal{F}_l}$ expires, $\mathcal{F}_l$ invokes *complain*().

When P_i executes *complain*(), it sends a `complain` message to all parties (line 43); it also sets the *complained* flag (line 44) and stops participating in the *sc-broadcasts* by not initializing the next instance. When a correct party receives $2t + 1$ `complain` messages, it enters the recovery mode. There is a complaint "amplification" mechanism by which a correct party that has received $t + 1$ `complain` messages and has not yet complained itself joins the complaining parties by sending its own `complain` message. Complaint amplification ensures that when some correct party enters the recovery mode, all other correct parties eventually enter it as well.

Recovery Mode

The recovery mode consists of three parts: (1) determining a watermark sequence number, (2) synchronizing all parties up to the watermark, and (3) delivering some payloads before entering the next epoch.

Part 1: Agree on Watermark The first part of the recovery mode determines a *watermark* sequence number w with the properties that (a) at least $t + 1$ correct parties have committed all sequence numbers less than or equal to w in epoch e, and (b) no sequence number higher than $w + 2$ has been committed by a correct party in epoch e.

Upon entering the recovery mode of epoch e, a party sends out a signed `committed` message containing $s - 1$, the highest sequence number that it has committed in this epoch.

It justifies $s-1$ by adding the bit string C that completes the *sc-broadcast* instance with tag e and $s-1$ (lines 50–51). Then, a party receives $n-t$ such `committed` messages with valid signatures and valid completion bit strings. It collects the received `committed` messages in a *watermark proposal vector* W and proposes W for MVBA. Once the agreement protocol decides on a *watermark decision vector* $\bar{W}$ (lines 52–56), the watermark w is set to the maximum of the sequence numbers in $\bar{W}$ minus 1 (line 57).

Consider the maximal sequence number $\bar{s}_j$ in $\bar{W}$ and the corresponding $\bar{C}_j$. It may be that P_j is corrupted or that P_j is the only correct party that ever committed $\bar{s}_j$ in epoch e. But the values contain enough evidence to conclude that at least $n-2t \geq t+1$ correct parties contributed to this instance of strong consistent broadcast. Hence, these parties have previously committed $\bar{s}_j - 1$. This ensures the first property of the watermark above (see also Lemma 2).

Although one or more correct parties may have committed $w+1$ and $w+2$, none of them has already *a-delivered* the corresponding payloads, because this would contradict the definition of w. Hence, these sequence numbers can safely be discarded. The discarding also ensures the second property of the watermark above (see Lemma 5). It is precisely for this reason that we delay the *a-delivery* of a payload to which sequence number s was committed until $s+2$ has been committed. Without it, the protocol could end up in a situation where up to t correct parties *a-delivered* a payload with sequence number $w+1$ or $w+2$, but it would be impossible for all correct parties to learn about this fact and to learn the *a-delivered* payload.

Part 2: Synchronize up to Watermark The second part of the recovery mode (lines 58–72) ensures that all parties *a-deliver* the payloads with sequence numbers less than or equal to w. It does so in a straightforward way using the *transferability* property of strong consistent broadcast.

In particular, every correct party P_i that has committed sequence number w (there must be at least $t+1$ such correct parties by the definition of w) computes *completing strings*

M_s for $s = 0, \ldots, w$ that complete the *sc-broadcast* instance with sequence number s. It can do so using the information stored in *log*. Potentially, P_i has to send $M_0, \ldots, M_w$ to *all* parties, but one can apply the following optimization to reduce the communication. Note that P_i knows from at least $n - t$ parties P_j their highest committed sequence number s_j (either directly from a `committed` message or from the watermark decision vector); if P_i knows nothing from some P_j, it has to assume $s_j = 0$. Then P_i simply sends a `complete` message with $M_{s_j+1}, \ldots, M_w$ to P_j for $j = 1, \ldots, n$. Every party receives these completing strings until it is able to *a-deliver* all payloads committed to the sequence numbers up to w.

Part 3: Deliver Some Messages Part 3 of the recovery mode (lines 73–84) ensures that the protocol makes progress by *a-delivering* some messages before the next epoch starts. In an asynchronous network, implementing this property must rely on randomized agreement or on a failure detector [FLP85]. This part uses one round of MVBA and is derived from the atomic broadcast protocol of Cachin et al. [CKPS01].

Every party P_i sends a signed `queue` message with all undelivered payloads in its initiation queue to all others (lines 73–74), collects a vector Q of $n - t$ such messages with valid signatures (lines 75–77), and proposes Q for MVBA. Once the agreement protocol has decided on a vector $\bar{Q}$ (lines 78–79), party P_i delivers the payloads in $\bar{Q}$ according to some deterministic order (lines 80–81).

Then P_i increments the epoch number and starts the next epoch by re-sending `initiate` messages for all remaining payloads in its initiation queue to the new leader (lines 82–84).

3.3.3 Optimizations

Both the BFT and KS protocols process multiple sequence numbers in parallel using a sliding window mechanism. For simplicity, our protocol description does not include this optimization and processes only the highest sequence number during every iteration of the loop in the parsimonious mode. However, Protocol PABC can easily be adapted to process

Ω payloads concurrently. In that case, up to Ω *sc-broadcast* instances are active in parallel, and the delay of two sequence numbers between *sc-delivery* and *a-delivery* of a payload is set to 2Ω. In part 1 of the recovery mode, the watermark is set to the maximum of the sequence numbers in the watermark decision vector minus Ω, instead of the maximum minus 1.

In our protocol description, the leader *sc-broadcasts* one initiated payload at a time. However, Protocol PABC can be modified to process a *batch* of payload messages at a time by committing sequence numbers to batches of payloads, as opposed to single payloads. The leader *sc-broadcasts* a batch of payloads in one instance, and all payloads in an *sc-delivered* batch are *a-delivered* in some deterministic order. This optimization has been shown to increase the throughput of the BFT protocol considerably [CL02].

Although the leader failure detector described in Section 3.3.2 is sufficient to ensure liveness, it is possible to enhance it using protocol information as follows. The leader in the parsimonious mode will never have to *sc-broadcast* more than two `dummy` messages consecutively to evict non-`dummy` payloads from the buffer. The failure detector oracle can maintain a counter to keep track of and restrict the number of successive `dummy` payloads *sc-broadcast* by the leader. If m is a non-`dummy` payload, the call to $\mathit{update}_{\mathcal{F}_l}(\texttt{deliver}, m)$ upon *a-delivery* of payload m resets the counter; otherwise, the counter is incremented. If the counter ever exceeds 2, then $\mathcal{F}_l$ invokes the *complain*() function.

3.3.4 Protocol Complexity

In this section, we examine the message and communication complexities of our protocol. We assume that strong consistent broadcast is implemented by the echo broadcast protocol using threshold signatures, and that MVBA is implemented by the protocol of Cachin et al. [CKPS01], as described in Section 3.2.1.

For a payload m that is *a-delivered* in the parsimonious mode, the message complexity is $\mathcal{O}(n)$, and the communication complexity is $\mathcal{O}\big(n(|m|+K)\big)$, where the length of a threshold signature and a signature share are at most K bits.

The recovery mode incurs higher message and communication complexities because it involves Byzantine agreement. The MVBA protocol of Cachin et al. [CKPS01] has an expected message complexity of $\mathcal{O}(n^2)$. Hence, determining the watermark in part 1 of the recovery involves expected $\mathcal{O}(n^2)$ messages. The corresponding expected communication complexity is $\mathcal{O}\big(n^3(|m| + K)\big)$ since the proposal values contain $\mathcal{O}(n)$ 3-tuples of the form (s_j, C_j, σ_j), each of length $\mathcal{O}(|m| + K)$. Here, m denotes the longest payload contained in the proposal.

In part 2 of the recovery mode, up to $\mathcal{O}(n^2)$ `complete` messages are exchanged. Recall that a `complete` message may encompass all payload messages that were previously *a-delivered* in the parsimonious mode of the epoch. Each of the $w \leq B$ completing strings in a `complete` message may be $\mathcal{O}(|m| + K)$ bits long, where m denotes the longest *a-delivered* payload. Hence, the communication complexity of part 2 of the recovery mode is $\mathcal{O}\big(n^2 B(|m| + K)\big)$.

Part 3 of the recovery mode is again dominated by the cost of the MVBA protocol. Hence, the expected message complexity of part 3 is $\mathcal{O}(n^2)$ and the expected communication complexity is $\mathcal{O}(n^3|m|)$ since the proposal values in MVBA are of length $\mathcal{O}(n|m|)$.

To summarize, for a payload that is *a-delivered* in the recovery mode, the cost is dominated by the MVBA protocol, resulting in an expected message complexity of $\mathcal{O}(n^2)$ and an expected communication complexity of $\mathcal{O}\big(n^2(n+B)(|m|+K)\big)$. Assuming that the protocol stays in the parsimonious mode as long as possible and *a-delivers* B payloads before executing recovery, the *amortized* expected complexities per payload over an epoch are $\mathcal{O}(n + \frac{n^2}{B})$ messages and $\mathcal{O}\big(\frac{n^3}{B}(|m| + K)\big)$ bits. It is reasonable to set $B \gg n$, so that we achieve amortized expected message complexity $\mathcal{O}(n)$ as claimed.

3.4 Analysis

In this section, we prove the following theorem.

Theorem 1 *Given a digital signature scheme, a protocol for strong consistent broadcast, and a protocol for multi-valued Byzantine agreement, Protocol* PABC *provides atomic broadcast for* $n > 3t$*.*

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

Lemma 2 *At the point in time when the first correct party has determined the watermark* w *during the recovery mode of epoch* e*, at least* $t+1$ *correct parties have committed sequence number* w *in epoch* e*.*

Proof 1 *First note that the lemma holds trivially if* $w = -2$*, and we may assume* $w \geq -1$ *in the rest of the proof. Let* j^* *denote the index of the largest sequence number* $\bar{s}_1, \ldots, \bar{s}_n$ *contained in the decision vector* $\bar{W}$ *of the agreement with tag* $ID|\texttt{watermark}.e$*. Note that* $w = \bar{s}_{j^*} - 1$ *according to the protocol. By the predicate* $Q_{ID|\texttt{watermark}.e}$*, the string* $\bar{C}_{j^*}$ *in* $\bar{W}$ *completes the strong consistent broadcast with tag* $ID|\texttt{bind}.e.j^*$*. According to the strong unforgeability property of strong consistent broadcast,* $\bar{C}_{j^*}$ *contains evidence that at least* $n-2t$ *distinct correct parties have participated in the* sc-broadcast *instance with sequence number* j^**. According to the logic of the parsimonious mode, a correct party initializes an instance of strong consistent broadcast with tag* $ID|\texttt{bind}.e.s$ *only after committing sequence number* $s-1$*. Hence, these* $n-2t \geq t+1$ *correct parties have also committed sequence number* $\bar{s}_j^* - 1 = w$*.*

Lemma 3 *If some correct party has entered the recovery mode of epoch* e*, then all correct parties eventually enter epoch* $e+1$*.*

Proof 2 *To establish the above lemma, we prove the following two claims.*

> Claim 1: *If some correct party has entered the recovery phase of epoch* e*, then all correct parties eventually enter the recovery mode of epoch* e*.*
>
> Claim 2: *If all correct parties have entered the recovery mode of epoch* e*, then all correct parties eventually enter epoch* $e+1$*.*

By the transitive application of the two claims, the lemma follows.

We first prove Claim 1. Suppose that a correct party P_i enters the recovery mode of epoch e. P_i does so only after receiving `complain` *messages from $2t+1$ distinct parties. At least $t+1$ of these messages must have been from correct parties. Hence, every correct party eventually receives $t+1$* `complain` *messages and sends its own* `complain` *message. Thus, every correct party eventually receives $n-t \geq 2t+1$* `complain` *messages and transitions to the recovery mode of epoch e.*

To prove Claim 2, one has to show that a correct party that enters the recovery mode of epoch e eventually completes all three parts of the recovery and moves to epoch $e+1$.

A correct party completes part 1 of the recovery because it eventually receives $n-t$ valid `committed` *messages from all correct parties and because all correct parties eventually decide in the MVBA protocol, according to its termination property.*

Part 2 of the recovery mode is concerned with ensuring that all correct parties a-deliver *the set of non-*`dummy` *payloads to which sequence numbers less than or equal to w were committed. Completion of part 2 by a correct party is guaranteed by the transferability property of strong consistent broadcast as follows. A correct party P_i that has committed sequence number w first* a-delivers *non-*`dummy` *payloads committed to sequence numbers $w-1$ and w (if it has not already done so). Then it sends a message with a set of completing strings $\{M_s | \ 0 \leq s \leq w\}$ to all other parties and moves to part 3 of the recovery mode. Here, M_s is the string that completes the* sc-broadcast *instance with sequence number s, which can be computed from the information stored in log. A correct party P_j that has not committed all sequence numbers less than w waits to receive the corresponding completing strings; P_j is guaranteed to receive them eventually, since, by Lemma 2, there are at least $t+1$ correct parties that have committed sequence number w. P_j then* a-delivers *all non-*`dummy` *payloads with sequence numbers up to w and moves to part 3 of the recovery mode.*

Analogous to part 1, completion of part 3 of the recovery is guaranteed by the fact that $n-t$ `queue` *messages will eventually be received and by the termination property of MVBA.*

Lemma 4 *Suppose e^* is the largest epoch number at any correct party at the point in time when $t+1$ correct parties have* a-broadcast *some payload m, and assume that some correct party did not* a-deliver *m before entering epoch e^*. Then some correct party P_i* a-delivers *m before entering epoch e^*+1.*

Proof 3 *The lemma is trivially satisfied if P_i* a-delivers *m during the parsimonious mode of epoch e^*. Otherwise, m is still present in the initiation queue $\mathcal{I}$ of at least $t+1$ correct parties. Since the initiation queues of $n-t$ parties are included in the decision vector of MVBA in part 3 of the recovery phase, at least one of these queues also contains m, and the lemma follows.*

Lemma 5 *Let w be the watermark of epoch e. No correct party commits a sequence number larger than $w+2$ in epoch e, and no correct party* a-delivers *a payload to which a sequence number larger than w has been committed in epoch e before reaching part 3 of the recovery mode.*

Proof 4 *The proof is by contradiction. Suppose that some correct party P_i has committed sequence number $w' = w+3$. Then, P_i has previously* sc-delivered *some m with tag $ID|\texttt{bind}.e.w'$, and the strong unforgeability property of strong consistent broadcast implies that at least $n-2t \geq t+1$ correct parties have participated in this* sc-broadcast *instance. Since correct parties initialize the* sc-broadcast *instance with tag $ID|\texttt{bind}.e.w'$ only after committing the previous sequence number, they have also committed sequence number $w'-1$.*

Therefore, these $t+1$ correct parties have also sent a signed `committed` *message containing sequence number $w'-1$ during recovery. Hence, the decision vector $\bar{W}$ with $n-t$ entries signed by distinct parties contained a triple $(\bar{s}_{j^*}, \bar{C}_{j^*}, \bar{\sigma}_{j^*})$ signed by one of those $t+1$ correct parties with $\bar{s}_{j^*} = w'-1$. By the agreement property of MVBA, every correct party must have computed the same $\bar{W}$ and set w to the maximum among the $\bar{s}_j$ values contained in $\bar{W}$ minus 1, i.e. $w = \bar{s}_{j^*} - 1 = w' - 2$. But this contradicts our assumption that $w' = w+3$.*

To prove the second part of the lemma, recall that the a-delivery *of the payload to which sequence number $s-2$ has been committed is delayed until after sequence number s has been* *committed. But since no correct party commits a sequence number larger than $w + 2$, as shown in the first part of the lemma, no correct party* a-delivers *any payload to which a sequence number larger than w has been committed in the parsimonious mode of epoch e. After the watermark in part 1 of the recovery mode has been determined, as can be seen from the checks in lines 59 and 68, part 2 ensures that payloads are* a-delivered *only up to the watermark; sequence numbers $w + 1$ and $w + 2$ are simply discarded.*

Lemma 6 *Suppose the watermark of epoch e satisfies $w \geq -1$. Then all correct parties eventually* a-deliver *all non-*`dummy` *payloads to which any correct party has committed a sequence number less than or equal to w in epoch e.*

Proof 5 *By the agreement and termination properties of MVBA, a correct party P_i eventually determines the watermark w of epoch e. During the parsimonious mode, it has* a-delivered *all non-*`dummy` *payloads with sequence numbers less than $s - 2$.*

When P_i moves to part 2 of the recovery mode, the code in lines 59–62 ensures that P_i also a-delivers *those non-*`dummy` *payloads to which sequence numbers $s - 2$ and $s - 1$ have been committed.*

Note that w may be smaller or larger than $s-1$, the highest committed sequence number. If $w < s$ and P_i has already committed w, then by the logic of the parsimonious mode and the loop in lines 59–62, P_i eventually a-delivers *all non-*`dummy` *payloads to which a sequence number less than or equal to w has been committed.*

On the other hand, if $w \geq s$ and P_i has not yet committed w, it waits to receive a string $\{M_{s'}\}$ that completes the sc-broadcast *instance with sequence number s' for $s' \leq w$. Party P_i is guaranteed to receive all of them eventually, since there are at least $t+1$ correct parties that have committed all sequence numbers up to w by Lemma 2. P_j then* a-delivers *all non-*`dummy` *payloads to which sequence numbers between s and w have been committed in epoch e.*

Lemma 7 *In every epoch e, there exists a sequence S of payloads such that any correct party* a-delivers *all payloads in epoch e in the order of S.*

Proof 6 *We first define a sequence S'_i for every correct P_i and show that the sequences computed by distinct correct parties are equal.*

S'_i is defined as follows. During the parsimonious mode, all payloads that P_i sc-delivers *are appended to S'_i in the order of their delivery. Suppose P_i has entered the recovery phase and has computed the watermark w; note that S'_i contains s elements. If $s > w + 1$, then S'_i is truncated to the first $w+1$ elements; if $s \leq w$, then S'_i is extended to $w+1$ elements by the payloads that are* sc-delivered *with tags $ID|\texttt{bind}.e.v$ for $v = s, \ldots, w$ through the* `complete` *messages in lines 68–72. During part 3 of the recovery mode, all payloads in $\cup_{j=1}^{n} \bar{\mathcal{I}}_j \setminus \mathcal{D}$ are appended to S'_i according to the given deterministic order, according to line 80.*

It is easy to see that, except with negligible probability, $S'_i = S'_j$ for any two correct parties P_i and P_j from the consistency property of strong consistent broadcast and from the agreement property of MVBA. Hence, one may think of a global sequence $S' = S'_i$ for all P_i.

Note that every party a-delivers *all non-*`dummy` *payloads in S'_i during epoch e. Hence, the sequence S is equal to S' with all* `dummy` *payloads removed.*

Proof 7 (Proof of Theorem 8.) *We have to show that Protocol* PABC *satisfies the validity, agreement, total order, and integrity properties of atomic broadcast.*

Validity *follows directly from Lemma 4. In fact, Lemma 4 proves a stronger version of the validity property stated in Section 3.2.2. The reason is that while the validity property specifies only an "eventual* a-delivery*" for a payload m that has been* a-broadcast *by $t + 1$ correct parties, Lemma 4 shows that m will be delivered relatively quickly.*

To show Agreement*, suppose that a correct party P_i has* a-delivered *some m in epoch e. We have to show that eventually all correct parties* a-deliver *m.*

We first distinguish two cases. In the first case, suppose P_i has a-delivered *m before entering part 3 of the recovery mode in epoch e. Then, Lemma 5 proves that a sequence number less than or equal to the watermark w of epoch e has been committed to m. Lemma 6 shows that all correct parties eventually* a-deliver *all non-*`dummy` *payloads to which a sequence number less than or equal to w has been committed, including m. This proves that the* *agreement property holds for the first case.*

In the second case, m was a-delivered *during part 3 of the recovery mode, after P_i had terminated the MVBA protocol. Then the agreement and termination properties of MVBA guarantee that all correct parties eventually terminate the MVBA protocol and* a-deliver *the same sequence of payloads.*

Hence, we have proved that if P_i a-delivers *m in epoch e, all correct parties in epoch e eventually* a-deliver *m. Extending the proof to the definition of agreement now only requires us to show that all correct parties eventually reach epoch e. Lemma 3 implies exactly that. Hence, by induction on the epoch, it can be easily seen that Protocol* PABC *satisfies agreement.*

The total order *property for a particular epoch e is proved by Lemma 7. Hence, by induction on the epoch number, it can be easily seen that Protocol* PABC *satisfies total order.*

For integrity*, we first show that a payload m is* a-delivered *at most once by a correct party P_i. Suppose that P_i* a-delivers *m in epoch e. Then, there are two possibilities, depending on whether the* a-delivery *happened before or after part 3 of epoch e's recovery mode was entered. In the former case (*a-delivery *before part 3 is entered), some sequence number less than or equal to w must have been committed to m. The check in line 23 ensures that a sequence number is committed to payload m only if $m \notin \mathcal{D}$. In the latter case (*a-delivery *in part 3 of the recovery mode), payload m must have been a part of the decision vector $\bar{Q}$. The check in line 80 ensures that only payloads in $\bar{Q}$ that are not in $\mathcal{D}$ are* a-delivered *in a deterministic order. Hence, it is clear that a payload m is* a-delivered *at most once by P_i. Even if corrupted parties* a-broadcast *payloads that have already been* a-delivered*, they are not* a-delivered *again.*

The second part of the integrity property, i.e., that our protocol only a-delivers *payloads that were previously* a-broadcast *by some party if all parties are correct, is trivially satisfied by the protocol.*

Table 3.1: Comparison of Efficient Byzantine-Fault-Tolerant Atomic Broadcast Protocols

Protocol	Synchrony for Safety?	Synchrony for Liveness?	Public-key Operations?	Message Complexity	
				Normal Cond.	Worst-case
Rampart [Rei95]	yes	yes	yes	$\mathcal{O}(n)$	unbounded
SecureRing [KMMS01]	yes	yes	yes	$\mathcal{O}(n)$	unbounded
ITUA [RPL$^+$02]	yes	yes	yes	$\mathcal{O}(n)$	unbounded
Cachin et al. [CKPS01]	no	no	yes	expected $\mathcal{O}(n^2)$	expected $\mathcal{O}(n^2)$
BFT [CL02]	no	yes	no	$\mathcal{O}(n^2)$	unbounded
KS [KS05]	no	no	yes	$\mathcal{O}(n^2)$	expected $\mathcal{O}(n^2)$
Protocol PABC	no	no	yes	$\mathcal{O}(n)$	expected $\mathcal{O}(n^2)$

3.5 Discussion

In this section, we discuss the practical significance of our parsimonious protocol and compare it with other efficient atomic broadcast protocols.

3.5.1 Practical Significance

In our formal system model, the adversary controls the scheduling of messages and hence the timeouts; thus, the adversary can cause parties to complain about a correctly functioning leader resulting in unnecessary transitions from the parsimonious mode to the recovery mode.

Unlike the adversary in our formal model, the network in a real-world setting will not always behave in the worst possible manner. The motivation for Protocol PABC — or any optimistic protocol such as the BFT and KS protocols for that matter — is the hope that timing assumptions based on stable network conditions have a high likelihood of being accurate. During periods of stability and when no new intrusions are detected, the optimistic assumption will be satisfied and our protocol will make fast progress in the parsimonious mode. However, both safety and liveness are still guaranteed even if the network is unstable, as long as no more than $t < n/3$ parties are actively misbehaving.

3.5.2 Comparison

Table 3.1 compares the synchrony assumptions, cryptographic requirements, and message complexity of Protocol PABC with the other recent Byzantine-fault-tolerant atomic broadcast protocols mentioned in the introduction. We devote the rest of this section to a more elaborate comparison with the two protocols closest to ours, namely the BFT protocol and the KS protocol.

Under stable network conditions and with a correct leader, all three protocols operate in their optimistic modes. These conditions are likely to apply during most of the running time of the system. In this case, the linear message and communication complexities of Protocol PABC compare favorably with the quadratic complexities of the BFT and KS protocols.

Under unstable network conditions, the deterministic BFT protocol can generate a potentially unbounded number of protocol messages by repeatedly switching from one epoch to another without making progress. This represents a violation of liveness and is prevented in the KS protocol and in Protocol PABC, since their recovery modes rely on randomized agreement and *a-deliver* some payloads. Naturally, using Byzantine agreement makes our recovery mode more expensive than the one of the BFT protocol.

The recovery mode of Protocol PABC is slightly more efficient than that of the KS protocol. The KS protocol requires four iterations of Byzantine agreement in addition to one iteration for each concurrently handled reliable broadcast instance. The recovery mode of our protocol uses only two iterations of Byzantine agreement, irrespective of the number of strong consistent broadcast instances that are concurrently handled.

3.5.3 Practical Issues

Keeping the Initiation Queue and Delivery Set Bounded

In the above description, Protocol PABC assumes an unbounded initiation queue $\mathcal{I}$ and delivery set $\mathcal{D}$. However, in the broader context, when the protocol is implemented as part of the agreement phase in a replica, that assumption is not required due to the following reasons. First, the only payloads that a correct replica initiates are requests from a finite set of authorized clients.

Second, the number of requests from a given client both in the initiation queue and delivery set can be limited. Client requests can be required to carry timestamps; then, the agreement phase participant keeps track of the timestamp of the last initiated request and the last *a-delivered* request from a given client. If a payload containing the request from client C with timestamp t has been initiated or *a-delivered* by a correct agreement phase participant, then from that point on, the replica will not accept payloads containing requests from C with lower timestamps for initiation. Hence, a correct agreement phase participant has at most one[1] outstanding request from a given client in its initiation queue. An additional check can be included in the *deliver*(m) function to check whether the payload m representing some request from a client C carries a timestamp greater than that of the last *a-delivered* request from C; if so, then m is discarded (just like `dummy` payloads) and not *a-delivered*.

Third, we introduce an acknowledgment mechanism for the *a-delivered* payloads. The agreement phase (Protocol PABC) and the execution phase (Protocol APE) have a producer-consumer relationship, in which the delivery set at Protocol PABC serves as the input for the execution phase. After *a-delivering* a payload containing a client request, Protocol PABC blocks until the it has obtained a reply certificate for the request.

[1]It is straightforward to generalize to some finite number, say K, of outstanding requests.

3.6 A Latency-Efficient Variant

In this section, we present the latency-efficient variant of Protocol PABC whose parsimonious phase uses a variant of strong consistent broadcast called *short strong consistent broadcast*. Short strong consistent broadcast provides the same properties as strong consistent broadcast; the only difference is in the implementation. The strong consistent broadcast implementation described in Section 3.2.1 and shown in Figure 3.5(a) uses a centralized approach (based on the designated sender) first to collect the quorum of $n-t$ signed echoes and then to disseminate the collected echoes to other parties. Such an implementation involves 3 communication steps: one for the designated sender to convey the payload, one to collect the echoes, and one to disseminate the echoes.

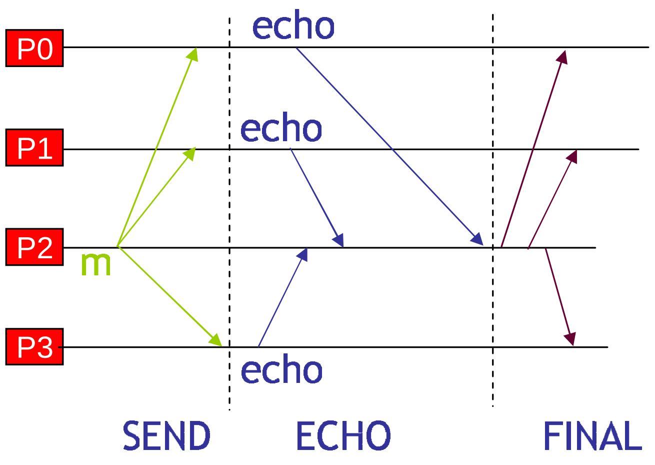
Short strong consistent broadcast is a simple variant that instead uses a decentralized approach to collect the echoes. As Figure 3.5(b) shows, there are only two communication steps: one for the designated sender to convey the payload and another for the parties to exchange their signed echoes with each other.

If the designated sender is correct, then in both implementations, the parties obtain the quorum of $n-t$ echoes for the payload. On the other hand, if the designated sender is faulty, it is possible in both implementations that some correct party obtains the quorum of $n-t$ echoes while other correct parties do not.

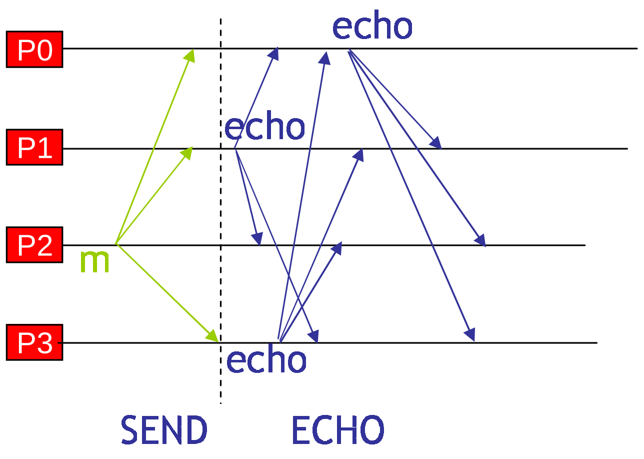
A consequence of using a decentralized approach to collect echoes is that the communication pattern in the second communication step is no longer many-to-one; instead, it is many-to-many. Hence, short strong consistent broadcast trades communication steps for message complexity.

3.7 Summary

We described an optimally resilient protocol that, for the first time, achieves asynchronous atomic broadcast with $\mathcal{O}(n)$ amortized expected messages per payload message. The pre-



(a) Strong Consistent Broadcast



(b) Short Strong Consistent Broadcast

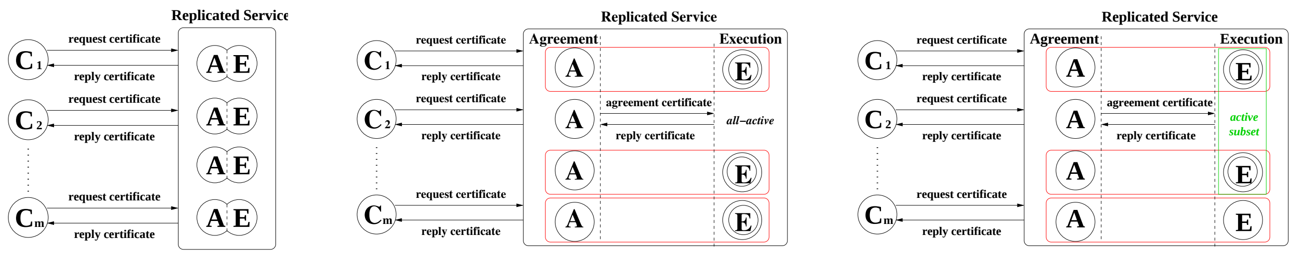
Figure 3.5: Strong Consistent Broadcast versus Short Strong Consistent Broadcast

vious best solutions used $\Theta(n^2)$ messages. We also described a variant protocol that, for the first time, achieves asynchronous atomic broadcast with latency degree 2 per payload message while providing optimal fault resilience. The previous best optimally resilient solutions had latency degree 3. Despite intrusions and instability, our protocol guarantees both safety and liveness as long as no more than $t < n/3$ parties are corrupted by the adversary. Our use of strong consistent broadcast, instead of reliable broadcast as in the BFT and KS protocols, introduces an additional digital signature computation at each party for every delivered payload. However, the intended deployment environments for our protocol are WANs, where message latency typically exceeds the time to perform digital signature computations; hence, we expect our protocol to be significantly more efficient than previous protocols in WANs.

Chapter 4

Parsimonious Execution

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.



(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

Figure 4.1: Successive Steps for Obtaining Efficient Execution in BFT Replication

4.1 Introduction

We propose a resource-efficient way to execute requests in BFT replication that is particularly well-suited for services in which request execution is resource-intensive (e.g., computation-intensive). The previous best way was the one proposed by Yin et al. that used $2t+1$ execution replicas. Previous work followed an *all-active* approach (Figures 4.1(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* [Kur02] 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.

The rest of this chapter is organized as follows. Section 4.2 describes an abstraction of the agreement phase that simplifies our protocol presentation. The protocol is presented in Section 4.3 and analyzed in Section 4.4. Section 4.5 discusses the practical significance of our protocol, lists applications for it, and compares it with related work. Finally, Section 4.6 summarizes the chapter.

4.2 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 relating to the ordering of client requests. We use $\mathcal{AS}$ to denote that service. Abstracting the agreement phase participants as one logical entity allows us to keep the focus on the execution replicas with whose behavior the parsimonious execution protocol is concerned. Later, in Section 4.5.3, we remove this assumption of a single-entity trusted agreement service. The functionality provided by $\mathcal{AS}$ 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. $\mathcal{AS}$ 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, $\mathcal{AS}$ sends an `agree` message with the *flag* value `false`. If $\mathcal{AS}$ 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`.

$\mathcal{AS}$ 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 r to $\mathcal{AS}$, then all correct execution replicas eventually receive an agreement certificate that binds some sequence number s to rc.

The above properties of $\mathcal{AS}$ allow the BFT-replicated service to tolerate an arbitrary number of corrupted clients; even if corrupted clients' requests are executed, those clients cannot cause the service states of correct execution replicas to become inconsistent. Client access control and request-filtering policies [CL02, YMV$^+$03] can be enforced in the implementation of $\mathcal{AS}$; the policies can effectively limit the number and scope of requests from corrupted clients.

4.3 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.

4.3.1 Protocol Properties

A protocol for the execution phase satisfies the following properties:

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

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

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

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

4.3.2 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*. Figure 4.2 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 (Figure 4.1(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

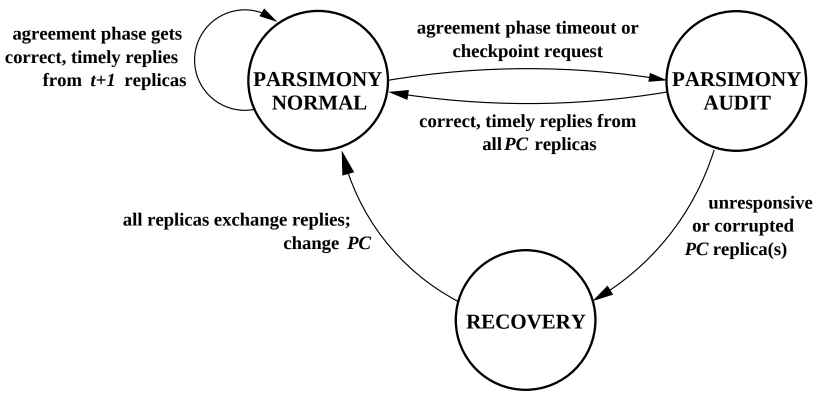

agreement phase gets correct, timely replies from t+1 replicas
PARSIMONY NORMAL
agreement phase timeout or checkpoint request
PARSIMONY AUDIT
correct, timely replies from all PC replicas
all replicas exchange replies; change PC
unresponsive or corrupted PC replica(s)
RECOVERY

Figure 4.2: The Three Modes of Protocol Operation

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.

4.3.3 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 Figures 4.3–4.7.

In the following description, we use $P_1, P_2, \ldots, P_{n_e}$ to denote the execution replicas, where

```
Protocol APE for execution replica $P_i$ with input parameter $\delta$ (checkpoint interval)
initialization:
  1: $\mathcal{PC} \leftarrow \{P_1, P_2, \ldots, P_{t+1}\}$   {current primary committee}
  2: $corrupt \leftarrow \emptyset$   {set of replicas for which $P_i$ has sent convict messages}
  3: $slow \leftarrow \emptyset$   {set of replicas for which $P_i$ 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 $P_i$ has reply certificates}
  6: $\mathcal{R} \leftarrow []$   {array of size $(\delta - 1)$ containing operations requested by $\mathcal{AS}$}
  7: $s \leftarrow 0$   {sequence number in the last agree message received from $\mathcal{AS}$}
  8: $u \leftarrow 0$   {sequence number up to which $P_i$'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 $P_i$ must execute even if
     $P_i \notin \mathcal{PC}$}
```

Figure 4.3: Protocol APE for Byzantine-Fault-Tolerant Request Execution (Part I)

$n_e \geq 2t+1$ for a desired fault resilience t. The *rank* of replica P_i is i. $\langle m \rangle_{\sigma_i}$ is used to denote a message m signed by replica P_i. P_i maintains a local sequence number variable s, where $s - 1$ indicates the highest sequence number for which P_i is sure that $\mathcal{AS}$ has obtained a reply certificate.

P_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 $\{P_1, P_2, \ldots, P_{t+1}\}$. 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 $\{P_1, P_2, P_3, P_4\}$. If replica P_2 was later added to replica P_4's *slow* or *corrupt* set, then the primary committee at replica P_4 would become $\{P_1, P_3, P_4, P_5\}$.

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

```
forever:
    12: wait for receipt of first-time agree($s+1$) or retransmit agree($s$) message from $\mathcal{AS}$
    {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 $P_i \in \mathcal{PC}$ then
    17:         $r \leftarrow obtain\_reply()$
    18:         $send\_reply(r,$ 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 $(P_i \in \mathcal{PC})$ then
    25:             $r \leftarrow obtain\_reply()$
    26:             $send\_reply(r,$ audit$)$
    27:         $update_{\mathcal{FD}}($start-monitor$, s)$
    28:         wait for $s \in certified$ or $s \in must\_do$ or $oldpc \neq \mathcal{PC}$
    29:         $update_{\mathcal{FD}}($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 $\mathcal{AS}$
    {Mode 3: Recovery Mode}
    33:     else $\{s \in must\_do\}$
    34:         $r \leftarrow obtain\_reply()$
    35:         $send\_reply(r,$ recover$)$
```

Figure 4.4: Protocol APE for Byzantine-Fault-Tolerant Request Execution (Part II)

```
function $send\_reply(r, mode)$:
36: if $(mode = \texttt{normal})$ then
37:     send the message $(\texttt{reply}, i, s, r)$ to $\mathcal{AS}$
38: else
39:     send the message $\langle \texttt{reply}, i, s, r \rangle_{\sigma_i}$ to all replicas

function $obtain\_reply()$:
40: if $\exists\ r$: ($replies$ contains $\langle \texttt{reply}, k, s, r \rangle_{\sigma_k}$ messages from $t+1$ distinct $P_k$ 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
        $P_k$)
56:     $certifiers \leftarrow$ set of $t+1$ distinct $P_k$ whose reply messages form reply certificate for
        $stable$
57:     send the message $(\texttt{state-request}, stable)$ to all $P_k \in certifiers$
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:     $execute(o)$

upon receiving message $\langle \texttt{reply}, j, s_j, r_j \rangle_{\sigma_j}$ from $P_j \notin corrupt$ for the first time:

62: if $i \neq j$ then
63:     $store\_reply(m, j)$
```

Figure 4.5: Protocol APE for Byzantine-Fault-Tolerant Request Execution (Part III)

```
function $store\_reply(m, j)$:        where $m = \begin{cases} \langle \texttt{reply}, j, s_j, r_j \rangle_{\sigma_j} & \text{if } j \neq i \\ (\texttt{reply}, i, s_i, r_i) & \text{otherwise} \end{cases}$
    64: $replies \leftarrow replies \cup \{m\}$
    65: if $j \neq i$ then
    66:     for replicas $P_k$ s.t. $(\langle \texttt{suspect}, k, j, s_j, c \rangle_{\sigma_k} \in suspects)$ do
    67:         forward $m$ to $P_k$
    68: if $\exists\ r$: $replies$ contains $\langle \texttt{reply}, k, s_j, r \rangle_{\sigma_k}$ messages from $t+1$ distinct $P_k$ then
    69:     $certificate \leftarrow$ set of $\langle \texttt{reply}, k, s_j, r \rangle_{\sigma_k}$ messages from $t+1$ distinct $P_k$
    70:     if $(s_j \notin certified)$ then
    71:         $certified \leftarrow certified \cup \{s_j\}$
    72:         if $(s_j \in must\_do)$ then
    73:             send $certificate$ to $\mathcal{AS}$
    74: $update_{\mathcal{FD}}(\texttt{got-reply}, j, s_j)$

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 $corrupt$

function $fault\_report(k, type, s_k)$:
    80: if $type = \texttt{mute-suspect}$ then
    81:     send $\langle \texttt{suspect}, i, k, s_k, c \rangle_{\sigma_i}$ to all replicas
    82: else if $type = \texttt{implicate}$ then
    83:     find $P_j$ : $(P_j \notin corrupt$ 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 $\{type = \texttt{convict}\}$
    88:     $certificate \leftarrow$ set of $\langle \texttt{reply}, j, s_k, r \rangle_{\sigma_j}$ messages from $t+1$ distinct $P_j$
    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:     $corrupt \leftarrow corrupt \cup \{P_k\}$
    92:     $must\_do \leftarrow must\_do \cup \{s_k\}$
    93:     $pc\_refresh()$
```

Figure 4.6: Protocol APE for Byzantine-Fault-Tolerant Request Execution (Part IV)

```
upon receiving message $\langle \texttt{suspect}, j, k, s_j, c\rangle_{\sigma_j}$ from $P_j \notin \mathit{corrupt}$ for the first time:

    94: if $P_k$'s reply message for $\texttt{agree}(s_j)$ is in $\mathit{replies}$ then
    95:     forward $P_k$'s reply message for $\texttt{agree}(s_j)$ to replica $P_j$
    96: $\mathit{suspects} \leftarrow \mathit{suspects} \cup \{\langle \texttt{suspect}, j, k, s_j, c\rangle_{\sigma_j}\}$
    97: if $P_k \notin \mathit{slow}$ and $\langle \texttt{suspect}, h, k, s_j, c\rangle_{\sigma_h}$ messages from $n-t$ distinct $P_h$ are in $\mathit{suspects}$
        then
    98:     $\mathit{not\_responsive}(s_j, k)$

upon receiving message $(\texttt{indict}, k, s_j, c, \mathit{proof})$ from $P_j \notin \mathit{corrupt}$ for the first time:

    99: if $(s_j \notin \mathit{must\_do})$ or $(P_k \notin \mathit{corrupt} \cup \mathit{slow})$ then
    100:    if $(\mathit{proof}$ contains $\langle \texttt{suspect}, h, k, s_j, c\rangle_{\sigma_h}$ messages from $n-t$ distinct $P_h)$ then
    101:        $\mathit{suspects} \leftarrow \mathit{suspects} \cup \mathit{proof}$
    102:        $\mathit{not\_responsive}(s_j, k)$

function $\mathit{not\_responsive}(s', k)$:
    103: send $(\texttt{indict}, k, s', c, \mathit{proof})$ to all replicas, where $\mathit{proof}$ is set of $\langle \texttt{suspect}, j, k, s', c\rangle_{\sigma_j}$
             messages from $n-t$ distinct $P_j$
    104: $\mathit{slow} \leftarrow \mathit{slow} \cup \{P_k\}$
    105: $\mathit{must\_do} \leftarrow \mathit{must\_do} \cup \{s'\}$
    106: $\mathit{pc\_refresh}()$

upon receiving message $(\texttt{implicate}, s_j, \mathit{proof})$ from $P_j \notin \mathit{corrupt}$ for the first time:

    107: if $\exists\, P_k, P_h : \langle \texttt{reply}, k, s_j, r_k\rangle_{\sigma_k} \in \mathit{proof}$ and $\langle \texttt{reply}, h, s_j, r_h\rangle_{\sigma_h} \in \mathit{proof}$ and $r_h \neq r_k$
         then
    108:     for $m \in \mathit{proof} \setminus \mathit{replies}$ do
    109:         $\mathit{store\_reply}(m, \text{sender}(m))$

upon receiving message $(\texttt{convict}, k, s_j, \mathit{proof})$ from $P_j \notin \mathit{corrupt}$ for the first time:

    110: if $\mathit{proof}$ contains $\langle \texttt{reply}, h, s_j, r\rangle_{\sigma_h}$ messages from $t+1$ distinct $P_h$ and
           $\langle \texttt{reply}, k, s_j, r_k\rangle_{\sigma_k} \in \mathit{proof}$ and $r \neq r_k$ then
    111:     for $m \in \mathit{proof} \setminus \mathit{replies}$ do
    112:         $\mathit{store\_reply}(m, \text{sender}(m))$
```

Figure 4.7: Protocol APE for Byzantine-Fault-Tolerant Request Execution (Part V)

Parsimonious Normal Mode

When P_i receives a *first-time* $\texttt{agree}(s + 1)$ message, the protocol at P_i moves to the parsimonious normal mode (lines 19–24). P_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 $\mathcal{AS}$ has at most one outstanding request, P_i can be sure that $\mathcal{AS}$ has obtained a reply certificate for $\texttt{agree}(s)$ when it receives the *first-time* $\texttt{agree}(s+1)$ message. Hence, P_i increments its sequence number variable s.

If $P_i \notin \mathcal{PC}$, then it does nothing more in this mode. On the other hand, if $P_i \in \mathcal{PC}$, then it executes the service operation indicated in the `agree` message, and sends the result r of the execution to $\mathcal{AS}$ in a `reply` message of the form $(\texttt{reply}, i, s, r)$. P_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 $\mathcal{AS}$ to obtain a reply certificate, which it then forwards to the respective client. $\mathcal{AS}$ can then issue the `agree` message with the next sequence number $s + 1$.

Transition from Normal to Audit Mode

The protocol at P_i transitions to the parsimonious audit mode from the parsimonious normal mode when

- P_i receives a *retransmit* $\texttt{agree}(s)$ message from $\mathcal{AS}$ and thereby learns that $\mathcal{AS}$ did not get a reply certificate for $\texttt{agree}(s)$ in a timely manner, or
- P_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, $\mathcal{AS}$ generates a special `agree` message in which the requested operation o is a *checkpoint* operation[1]. When P_i receives the checkpoint request, P_i knows that $\mathcal{AS}$ 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 P_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 P_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.

Parsimonious Audit Mode

In this mode (lines 25–38), 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 P_i starts a timer and expects to obtain a reply certificate before the timer expiry. If progress is not being made, the replicas collectively switch the protocol to the recovery mode, in which all correct replicas generate their own reply messages (if they hadn't done so previously) and ensure that $\mathcal{AS}$ 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 $\mathcal{AS}$, and the protocol will switch back to the parsimonious normal mode.

To enable the monitoring of progress, the $\mathcal{PC}$ replicas are required to send signed `reply` messages to all execution replicas. A $\mathcal{PC}$ replica P_j retrieves the result value of the `reply` message from the *replies* data structure if it had previously sent a `reply` message to $\mathcal{AS}$ in the parsimonious normal mode. Otherwise, P_j obtains the result value by executing the

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

operation specified in the corresponding `agree` message from $\mathcal{AS}$.

Failure Detection and Transition from Audit to Recovery Mode

An execution replica P_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 message.
- *Wrong Replies:* A $\mathcal{PC}$ replica did send its reply message, but with the wrong result value.

To determine if the $\mathcal{PC}$ is performing its job correctly, every replica P_i has access to a failure detector oracle $\mathcal{FD}$. For simplicity, Figs. 4.3–4.7 do not include the pseudocode for $\mathcal{FD}$. 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 $\mathit{update}_{\mathcal{FD}}$ that Protocol APE synchronously invokes to convey protocol-specific information. During the execution of the $\mathit{update}_{\mathcal{FD}}$ function, $\mathcal{FD}$ can read the global data structures and variables of Protocol APE.

An implementation of $\mathcal{FD}$ at a replica P_i can check whether the $\mathcal{PC}$ replicas are functioning properly based on a local timeout and protocol information as follows. When P_i is in the audit mode, the call to $\mathit{update}_{\mathcal{FD}}(\texttt{start-monitor}, s)$ starts a timer $T_{\mathcal{FD}}$ (line 33). If the $\mathcal{PC}$ changes, or if P_i obtains a reply certificate for $\texttt{agree}(s)$, or if the protocol switches to the recovery mode, the call to $\mathit{update}_{\mathcal{FD}}(\texttt{stop-monitor}, s)$ disables the timer $T_{\mathcal{FD}}$ (line 35). When $T_{\mathcal{FD}}$ expires, $\mathcal{FD}$ notifies Protocol APE by invoking the $\mathit{fault\_report}(k, \texttt{mute-suspect}, s)$ for each $\mathcal{PC}$ replica P_k whose reply message for $\texttt{agree}(s)$ has not yet been received. P_i sends a signed `suspect` message for P_k to all execution replicas (line 104). The `suspect` message for P_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 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. We discuss this in detail in section 4.3.4.

Consider the point in time when P_i's `suspect` message for P_k is received at a correct replica P_j. If P_j (has received or) later receives P_k's `reply` message with sequence number s, then P_j simply forwards P_k's `reply` message to P_i upon receiving P_i's `suspect` message. This `reply` forwarding (lines 89–90 and lines 117–118) ensures that if at least one correct replica has received P_k's `reply` message for `agree`(s), then all correct replicas will eventually receive P_k's message. On the other hand, if no correct replica has received P_k's `reply` message for `agree`(s) in a timely fashion (determined by the replicas' respective local timers), then each of the $n - t$ correct replicas will generate a `suspect` message for P_k. A correct replica P_i keeps track of all the `suspect` messages it receives by storing them in a data structure, *suspects* (line 119).

After receiving $\langle \texttt{suspect}, j, k, s, c\rangle_{\sigma_j}$ messages from $n - t$ distinct P_js (lines 120–121), replica P_i adds P_k to its *slow* set and sends an `indict`[2] message of the form $(\texttt{indict}, k, s, c, \mathit{proof})$ to all replicas, where *proof* contains the signed `suspect` messages. P_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 `reply` messages for those requests. Having added the $\mathcal{PC}$ replica P_k to its *slow* set, P_i updates its $\mathcal{PC}$ accordingly. Any replica P_j at which $P_k \notin \mathit{slow} \cup \mathit{corrupt}$ that receives P_i's `indict` message will add P_k to its *slow* set, add s to the *must_do* set, send its own similar `indict` message for P_k to all replicas, and update its $\mathcal{PC}$ (lines 122–125).

To identify wrong replies, Protocol APE invokes the $\mathit{update}_{\mathcal{FD}}(\texttt{got-reply}, j, s_j)$ function (line 97) when the protocol receives P_j's `reply` message for `agree`(s_j). $\mathcal{FD}$ compares the result value in the `reply` message with the values in the `reply` messages received from other

[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 $n-t$ replicas, having not received replica P_k's reply message for `agree`(s) in a timely manner, accuses replica P_k of being slow.

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 `reply` messages from two replicas indicates that at least one of them is corrupted. However, to be able to pinpoint in a provable manner which of those two replicas is corrupt, a reply certificate is needed; any replica whose `reply` message contains a result value different from that in a reply certificate is corrupt. When P_j's result value differs from that in a previously received `reply` (say from P_k) but a reply certificate for $\texttt{agree}(s_j)$ has not yet been obtained, $\mathcal{FD}$ notifies Protocol APE by invoking the $\mathit{fault\_report}(k, \texttt{implicate}, s_k)$ function. Replica P_i then sends an `implicate`[3] message of the form $(\texttt{implicate}, s, \mathit{proof})$ to all replicas, where *proof* contains P_j and P_k's `reply` messages (lines 106–108). A recipient P_k of P_i's `implicate` message will not know which of the implicated replicas is actually corrupt, but will be convinced of the need to switch to the recovery mode and add s to the *must_do* set.

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 $\mathcal{AS}$, 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 $\mathcal{PC}$.

Protocol APE addresses the above problem as follows. If the fraction of requests that resulted in a transition from the normal to audit mode exceeds a fixed threshold, the protocol operates semi-permanently in the audit mode until the next transition to the recovery mode. After the $\mathcal{PC}$ is changed in the recovery mode, the protocol reverts back to the normal mode.

[3]The legal term "implicate" means "to bring into incriminating connection." In Protocol APE, the `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).

Recovery Mode

Only the $\mathcal{PC}$ replicas send `reply` messages in the normal and audit modes. In the recovery mode (lines 39–41), 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 $\mathcal{AS}$.

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. Before executing the operation specified in the `agree`(s) message, however, a backup P_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 P_i execute the operation specified in the `agree`(s) message. Bringing the state up to date may involve two steps (lines 76–84):

- If $u < stable$ at P_i, where *stable* is the sequence number of P_i's last stable checkpoint, then P_i first obtains the state corresponding to the execution of all requests with sequence numbers up to *stable* (lines 77–81). P_i determines the $t+1$ replicas whose `reply` messages form the reply certificate for `agree`(*stable*). P_i then requests the state corresponding to that checkpoint by sending a message of the form (`state-request`, *stable*) to those $t+1$ replicas. Since at least one of the replicas is correct, P_i is guaranteed eventually to obtain the state corresponding to that checkpoint. P_i can easily verify whether the state transferred is correct; P_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. P_i then changes the value of u to be equal to *stable*.

- P_i updates its state to reflect the execution of requests with sequence numbers from

$u+1$ to $s-1$ (lines 82–84). To perform the update, P_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 [CL02].

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 $\mathit{update}_{\mathcal{FD}}(\texttt{got-reply}, k, s_k)$ function detects that P_k's result value for $\texttt{agree}(s_k)$ differs from that in a reply certificate, then $\mathcal{FD}$ notifies Protocol APE that P_k is corrupted by invoking the $\mathit{fault\_report}(k, \texttt{convict}, s_k)$ function. Replica P_i adds P_k to its local *corrupt* set, updates its $\mathcal{PC}$ accordingly, and shares this information about P_k with other replicas by sending a `convict` message to all replicas (lines 110–116). The `convict`[4] message has the form $(\texttt{convict}, k, s, \mathit{proof})$, where *proof* contains the reply certificate and replica P_k's `reply` message for $\texttt{agree}(s)$. Once a correct replica has added P_k to its *corrupt* set, it discards any further protocol messages received directly from P_k.

Primary Committee Changes

At a correct execution replica, any $\mathcal{PC}$ change (line 102) 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 differ in their perspectives of the primary committee, their perspectives will eventually concur.

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

4.3.4 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 98). 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 100). A reset counter c is used to keep track of the number of resets (line 99). 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 101).

Since a correct replica P_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 P_i encounters a situation in which $|slow \cup corrupt| > t$, then any correct replica P_j will also eventually encounter a situation $|slow \cup corrupt| > t$. Thus, if the reset-counter c at replica P_i is incremented, then eventually all correct replicas will also increment their respective reset-counters to $c+1$.

4.4 Analysis

In this section, we prove the following theorem.

Theorem 8 *Given an agreement phase abstraction, Protocol* APE *provides BFT replication for $n > 2t$.*

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

Lemma 9 (Total Order) *At any two correct execution replicas P_i and P_j, their internal states just after executing the request indicated by* `agree`*(s) are the same.*

Proof 8 *The replicas implement deterministic state machines and are initialized to the same internal state. Recall that $\mathcal{AS}$ satisfies the agreement property specified in Section 4.2. These facts directly imply that the lemma is trivially satisfied for $s = 1$. For $s > 1$, P_i and P_j will execute the request indicated by* `agree`*(s) (line 70) 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 $s - 1$.*

As can be seen in the state_update() function (lines 76–84), 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 81). When $u = stable$, the requests corresponding to sequence numbers from $stable + 1$ to $s - 1$ are executed in sequence number order (lines 82–84). Thus, when $u = s - 1$ at two correct execution replicas P_i and P_j, their internal states will be identical. Then, it follows from the determinism of the state machines and the agreement property of $\mathcal{AS}$ that P_i and P_j's internal states will be the same after they have executed the request indicated by* `agree`*(s).*

Lemma 10 (Update Integrity) *Any correct execution replica updates its internal state in response to the request indicated by* `agree`*(s) at most once, and only if $\mathcal{AS}$ actually sent that message.*

Proof 9 *We first show that the request indicated by the* `agree`*(s) message is executed at most once by a correct execution replica P_i. The only time when the request indicated by the* `agree`*(s) message is executed by P_i is during the obtain_reply() function at line 70. After obtaining the result value r, P_i stores its own* `reply` *message for* `agree`*(s) in the replies data structure (line 74). Any subsequent invocations of the obtain_reply() function (lines 23, 31, and 40) will not execute the request again, since the function will retrieve the reply from the replies data structure (lines 63–64). Hence, P_i executes the request at most once.*

The second part of the update integrity property which states that P_i only updates its *internal state in response to the requests actually sent by $\mathcal{AS}$ is trivially satisfied by the protocol. The reason is that the execution of the request indicated by* `agree`*(s) (line 70 in the obtain_reply() function) is always preceded by receipt of the* `agree`*(s) message directly from $\mathcal{AS}$ (line 12). Hence, Protocol* APE *satisfies update integrity.*

Lemma 11 (Result Integrity) *If r is the result value in the reply certificate received by $\mathcal{AS}$ for* `agree`*(s), then at least one correct execution replica sent a* `reply` *message for* `agree`*(s) with result value r.*

Proof 10 *Result integrity is trivially satisfied by Protocol* APE *since, by definition, the reply certificate consists of* `reply` *messages from $t+1$ distinct replicas, out of which at most t are corrupt.*

Lemma 12 (Liveness) *If $\mathcal{AS}$ sends* `agree`*(s) to the execution replicas, it eventually receives a reply certificate for* `agree`*(s).*

Proof 11 *Once the* first-time `agree`*(s) message has been received from $\mathcal{AS}$, the next message a correct replica P_i waits to receive from $\mathcal{AS}$ is a* first-time `agree`*$(s+1)$ or a* retransmit `agree`*(s) message from $\mathcal{AS}$ (line 12). If P_i receives a* first-time `agree`*$(s+1)$ message, then by our assumption that $\mathcal{AS}$ has only one outstanding request, it is clear that $\mathcal{AS}$ has obtained a reply certificate for* `agree`*(s); hence, liveness is trivially satisfied.*

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

It is clear that if s is added to P_i's must_do set when P_i is in the parsimonious audit mode, then P_i switches to the recovery mode (lines 39–41), in which, irrespective of whether P_i is a $\mathcal{PC}$ replica or not, P_i's `reply` *message will be received at $\mathcal{AS}$. At replica P_i, the addition of s to its must_do set (lines 109, 115, and 128) is always preceded by P_i sending an* `implicate`*,* `convict`*, or* `indict` *message with valid proof that will convince any correct* *recipient replica P_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, $\mathcal{AS}$ will eventually obtain a reply certificate for* `agree`*(s).*

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

Then, the key to showing that Protocol APE *satisfies liveness is to show that the clause ($s \in$ certified* ***or*** *$s \in$ must_do) eventually holds at a replica P_i that is in the parsimonious audit mode. We show exactly that by proving Claim 1 below. Hence, Protocol* APE *satisfies liveness.*

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

Proof 12 *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 P_i never comes out of the* ***repeat until*** *loop (lines 28–36). That is possible only in two cases: (1) the* ***repeat until*** *loop iterates infinitely and the* ***wait for*** *clause in line 34 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 34. We now show that both cases result in contradictions.*

Case (1): In this case, the ***repeat until*** *loop iterates infinitely and at each iteration of the loop, there is a $\mathcal{PC}$ change. Every $\mathcal{PC}$ change at P_i is the result of adding some new replica to the set slow $\cup$ corrupt. Recall that the next $\mathcal{PC}$ is chosen to consist of the $t+1$ lowest-ranked replicas that are not in slow $\cup$ corrupt. Hence, after $n-(t+1)$ $\mathcal{PC}$ 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 $n-(t+1)$ $\mathcal{PC}$ 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* $s \in$ *certified eventually, contradicting our assumption.*

Case (2): The control can get stuck at the ***wait for*** *clause in line 34 only if the clause* $(oldPC \neq \mathcal{PC})$ *never became true at* P_i*, i.e., the* $\mathcal{PC}$ *set at* P_i *never changed. The* $\mathcal{PC}$ *set at* P_i *would never change only if there had been no further additions to* P_i*'s slow or corrupt sets. That would be possible only if the slow and corrupt sets at any other correct replica were proper subsets of* P_i*'s slow and corrupt sets respectively.*

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

To recap, we have so far shown that if the control at P_i *gets stuck at the* ***wait for*** *clause in line 34, then the* $\mathcal{PC}$ *set at all correct replicas will eventually be the same as* P_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* P_i *never changes. Recall that the* `reply` *forwarding done in lines 89–90 and lines 117–118 ensures that if at least one correct replica has received a* $\mathcal{PC}$ *replica* P_k*'s* `reply` *message for* `agree`*(s), then all correct replicas will eventually receive* P_k*'s message. This reply-forwarding logic coupled with our initial assumption that s is never added to the certified set implies that there must be some* $\mathcal{PC}$ *replica* P_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* P_k *with sequence number s. When those* `suspect` *messages are eventually received at* P_i*,* P_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 8: Lemmas 9, 10, 11, and 12 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 $n > 2t$.

4.5 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.

4.5.1 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 $\mathcal{AS}$ for every request, since all replicas execute all requests and send the results to $\mathcal{AS}$. 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.

In our formal system model, the adversary controls the scheduling of messages and hence the timeouts of the failure detector; thus, the adversary can cause a correctly functioning $\mathcal{PC}$ replica to be added to the *slow* sets of correct replicas. Such inaccurate failure detection will result in Protocol APE unnecessarily switching from the parsimonious modes to the recovery mode.

Unlike the adversary in our formal model, the network in a real-world setting will not always behave in the worst possible manner. The motivation for an optimistic protocol such as ours is the hope that timer values that are set based on stable network conditions have a high likelihood of being accurate. During periods of stability and when the $\mathcal{PC}$ replicas do not actively misbehave, the optimistic hope will be satisfied, and our protocol will provide resource-efficient request execution with roughly the same latency as the all-active approach.

Even if the optimistic hope is not satisfied, our protocol guarantees safety and liveness. Safety mainly relates to replica state consistency. Since replicas always execute a request bound to sequence number s only after a state update that reflects the execution of all lower-sequence-numbered requests, safety is never violated. Liveness, which is the ability to obtain a reply certificate eventually, is also guaranteed; inaccurate failure detection can, at worst, cause correct $\mathcal{PC}$ 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.

4.5.2 Practical Applications

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

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

2. In the computational Grid, many services are computation-intensive. BFT replication

can be used to obtain a trustworthy system from untrusted participating Grid nodes. Since Grid nodes may be shared among several Grid services, our protocol can help significantly reduce the performance impact (compared to the all-active execution approach) that one service has on other services running in the same Grid node.

4.5.3 From a Trusted Agreement Service to Agreement Replicas

For the sake of simplicity, the description of Protocol APE given in this chapter assumes a single-entity trusted agreement service. There are two advantages of the parsimonious normal mode over the parsimonious audit mode. The first advantage is that there is no inter-replica communication; the primary committee replicas send their reply messages directly to the agreement service. The second advantage is that the reply messages need not be digitally signed; MAC authentication is sufficient.

In the system model described in Section 2.2, the trusted agreement service is implemented by a set of at least $3t + 1$ agreement replicas to achieve a fault resilience t. Although a logical separation between the agreement phase and execution phase is maintained, Protocol APE runs in the same replica as Protocol PABC. In such a model, the first advantage (concerning inter-replica communication) is lost, since sending the reply message to the agreement service is equivalent to sending the reply message to all agreement phase participants (i.e., to all replicas). Hence, Protocol APE should be run only in two modes, the parsimonious audit mode and the recovery mode. Although such an optimization will result in the loss of the second advantage (concerning digital signatures), that loss is not a significant one in the context of our target applications. Recall that the target applications for our parsimonious execution protocol involve resource-intensive request processing; one can expect the request-processing overhead in such applications to be significantly more expensive than digital signature computations.

The above optimization of running Protocol APE in two (as opposed to three) modes has two additional benefits. First, it obviates the need for checkpointing and state transfer.

The reason is that in the parsimonious audit mode, a backup replica obtains reply messages, and hence the reply certificate, from all primary committee members. If information about the update to the internal state caused by the request execution is piggybacked on the reply message, then a reply certificate for a request indicates the correct value of the state update corresponding to that request. Using that information, a backup replica can update its internal state after obtaining a reply certificate. Hence, backup replicas can keep their states up-to-date for each request and there is no need for state transfer (and hence checkpointing) when switching to the recovery mode. The second benefit of the above optimization is that it avoids the situation (described in Section 4.3.3) in which a corrupted $\mathcal{PC}$ replica cleverly degrades protocol performance by causing repeated transitions from the normal to audit mode and back to normal mode, without a change in the $\mathcal{PC}$.

4.5.4 Related Work

BFT replication techniques are of two categories: quorum replication and state machine replication. Quorum replication (e.g., [MR98, MAD02a]) uses subsets of replicas (called *quorums*) to implement read/write operations on the variables of a data repository, such that any two subsets intersect in enough correct replicas. State machine replication can be used to perform arbitrary computations accessing arbitrary numbers of variables; quorum replication is less generic and cannot handle concurrent requests by clients to update the same information. Our protocol is similar to quorum systems in that it uses a subset of replicas to perform operations. However, the similarity is only superficial; our protocol is concerned with the execution phase of state machine replication, our use of a $(t+1)$-subset of replicas to execute requests is based on whether the system is stable or not (a distinction that quorum systems do not make), and (unlike quorum systems) we do not use different subset sizes for read and write operations.

Our protocol is both unique and novel. While most work on BFT replication has focused on the hard problem of Byzantine agreement (e.g., [CL02, Rei95]), 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., [BSTM93, DS02]), 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 [RCL01] and the BASE extension by Yin et al. [YMV$^+$03]. In particular, one could combine our protocol with (1) the proactive recovery and abstraction techniques of BASE to overcome the drawbacks of state machine replication in many applications (namely, the determinism requirement and the assumption that at most one-third of the replicas are corrupt), and (2) the privacy firewall architecture of [YMV$^+$03] to obtain BFT confidentiality.

In the context of parallel computing, Sarmenta [Sar02] proposed mechanisms for tolerating erroneous results submitted by malicious volunteers in the Grid, SETI@home, and other *volunteer computer systems*. The mechanisms, called *credibility-based fault-tolerance* mechanisms, estimate the credibility of a node and use these probability estimates in limiting the amount of redundant computations necessary to meet desired error rates. However, their scheme trades off correctness for performance and is not relevant to applications that are stateful or cannot tolerate any errors at all (e.g., banking or financial applications). Also, their mechanisms operate in a system and fault model that is very restrictive (e.g., it requires synchronous computations and non-collusion among malicious nodes) and less generic than our system and fault model.

4.6 Summary

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 $n > 2t$ execution replicas to execute a request. Despite failures and instability, our protocol always guarantees both safety and liveness as long as no more than $t < n/2$ 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.

Chapter 5

Group Membership and Reconfiguration

5.1 Motivation and Approach

The parsimonious protocols described in the previous chapters are based on a *static group*, i.e., the group of nodes that constitute the agreement and execution replicas are fixed at system startup and remain unchanged throughout the system lifetime. In order to subvert the service, the adversary has to compromise more than t agreement replicas or more than t execution replicas. Proactive recovery mechanisms can help reduce the time available for the adversary to accomplish this objective from practically infinity (for long-lived services) to a reasonable *window of vulnerability.* For example, Castro and Liskov [CL02] use a watchdog timer at each replica node to interrupt the node activities periodically; the node is rebooted and a copy of the service code is reloaded from a tamper-resistant read-only disk. A combination of checkpointing and state-transfer techniques is used to restore the service state in case of corruption. As another example, refreshing replica keys [CKLS02, CL02] has also been suggested as a proactive recovery technique.

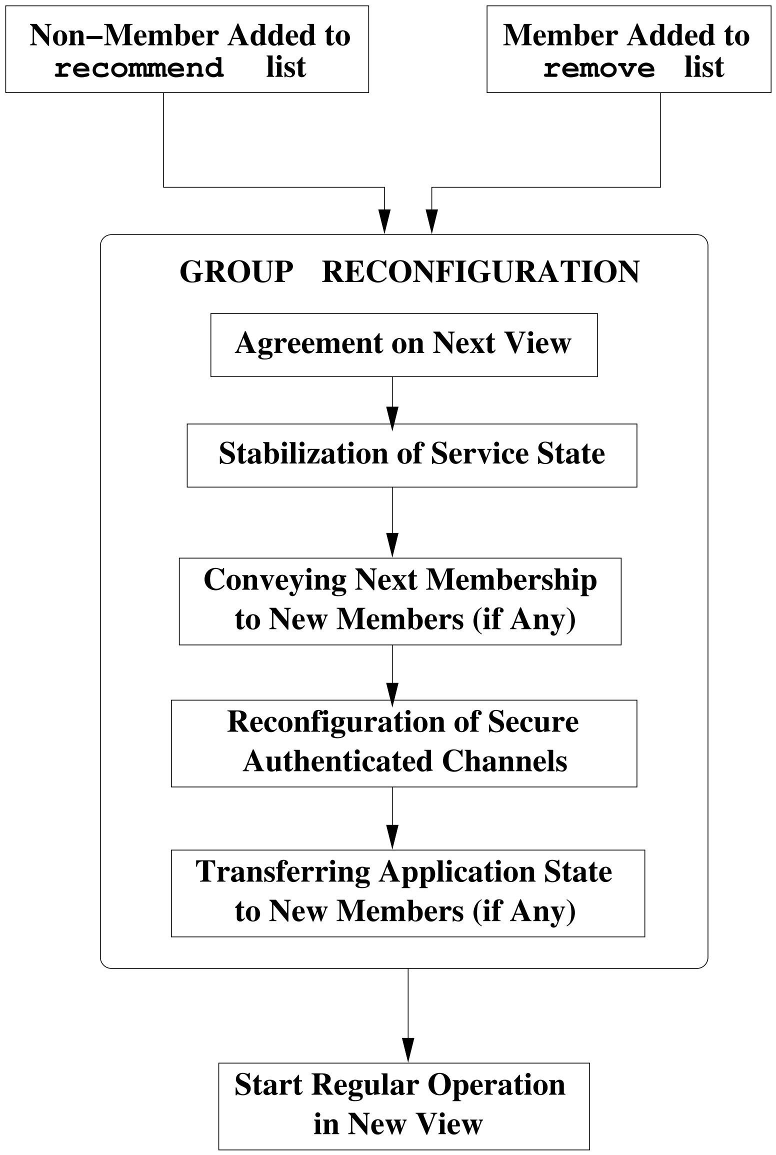
In a real-world setting, it is important to complement the implementations of our parsimonious protocols with proactive recovery and replica diversity techniques to improve the coverage of the assumption that at most t replicas are faulty. However, those techniques alone may not be sufficient. While key-refresh-only proactive recovery [CKLS02] makes cryptographic keys of corrupt nodes stolen by an attacker useless in the subsequent windows of vulnerability, it does not offer any immunization against an attacker who can replace the service code binaries running at the corrupted replicas. The periodic reboot strategy of [CL02] helps recover from binary code replacement; however, it requires special-purpose hardware, and its effectiveness is still limited for the following reason. An attacker having physical control of the corrupted nodes would be able to tamper with their recovery. Once an attacker has successfully compromised a node, it is very likely that, using the same attack strategy as before, the attacker can easily and quickly compromise the node again. Thus, with each new node corruption, the attacker has a better chance of subverting the service operation in the next window of vulnerability. Even if there are only less than t corrupted nodes and the replicated service as a whole is still operational, the corrupted nodes could act as a nuisance. For example, they could force frequent transitions from the parsimonious modes to the high-overhead recovery modes, thereby making it difficult to provide performance guarantees.

Situations such as the above render the use of proactive recovery techniques alone inadequate. That fact, coupled with the growing need to make NISs adaptive in the face of attacks, makes a strong case for augmenting BFT replication with the ability to dynamically alter the composition of the replication group.

The ability to add new nodes and remove suspected corrupt nodes has been previously explored in the context of Byzantine fault-tolerant group communication systems (e.g., [Rei96, KMMS01, RPL+02]). However, the replication protocols and even the group management protocols *required* the removal of suspected corrupt nodes in order to provide their specified properties. Such a requirement opens up an easily exploitable vulnerability for the attacker. The attacker may cause (for example, by slowing down the communication links) the misclassification of correct nodes as nodes suspected of being corrupt. Such a misclassification would result in those nodes being removed from the group, thereby resulting in violations of safety properties or even the disintegration of the group.

Our approach to obtaining reconfigurable BFT replication systems without the above pitfalls of the status quo is to design replication protocols that never *require* the removal of faulty replicas, so that reconfigurability of the replication group can be kept as a capability that is used very selectively and conservatively.

5.2 Overview



Non–Member Added to recommend list
Member Added to remove list
GROUP RECONFIGURATION
Agreement on Next View
Stabilization of Service State
Conveying Next Membership to New Members (if Any)
Reconfiguration of Secure Authenticated Channels
Transferring Application State to New Members (if Any)
Start Regular Operation in New View

Figure 5.1: Steps Involved in Dynamic Group Reconfiguration

We have developed a suite of protocols called RMan (which stands for Reconfiguration Management) that defines a group abstraction for the replicas and allows for the dynamic alteration of the composition of the replication group. RMan comprises the following modules:

1. Protocol Admission: At a party that is not a group member, Protocol Admission helps the party join the group. At a group member, the protocol regulates, based on specified admission control policies, the entry of new members into the group.

2. Algorithm FD (which stands for failure detection): At group members, Algorithm FD identifies, based on specified member removal policies, potential candidates for removal from the group.

3. Protocol GMA: At group members, Protocol GMA (which stands for *group membership agreement*) ensures that all correct group members agree on the next group membership.

4. Protocol D-PABC (which stands for dynamic PABC): Protocol D-PABC is an extension of Protocol PABC that provides interfaces for changing the set of agreement phase participants and for receiving notifications about an impending change.

5. Protocol D-APE (which stands for dynamic APE): Protocol D-APE is extension of Protocol APE that provides interfaces for changing the set of execution replicas and for receiving notifications about an impending change.

6. Algorithm Reconf: Algorithm Reconf is responsible for coordinating the transition of Protocol D-PABC and Protocol D-APE from one group membership to the next in a manner that allows the provision of view synchrony [BJ87] properties. It also notifies all other modules about when the reconfiguration procedure is complete so that they can re-initialize their local data structures based on the new membership and re-start normal operation.

The triggers for group reconfiguration and the steps involved in reconfiguration are shown at a high level in Figure 5.1. The trigger for reconfiguration comes in one of two forms:

(1) when Protocol Admission has identified a party that is currently not a group member to be worthy of inclusion in the group, or (2) when Algorithm FD has identified a group member as a potential candidate for exclusion from the group. The above triggers are fed to Protocol GMA, which then starts an MVBA instance to agree on the next group membership. Protocol GMA notifies Algorithm Reconf about the agreed-upon next membership.

From that point on, Algorithm Reconf directs the *view installation*, which is a series of steps by which a party systematically transitions from the current *view* or membership of the group to the next. Algorithm Reconf instructs Protocol D-PABC to determine a synchronization point at which current group members can suspend their operations and block until the membership change is effected. Prior to blocking, Protocol D-PABC ensures that all correct group members atomically deliver all payloads up to that synchronization point and no more. Algorithm Reconf then instructs Protocol D-APE to update the (replicated) service state to reflect the execution of all client requests corresponding to payloads that were atomically delivered up to that synchronization point. That ensures that the service states at all correct members at the end of the current membership are the same. If the agreed-upon next membership includes any new members, then Algorithm Reconf instructs Protocol Admission to convey to those new members the new membership information and a digest of the *stable* service state. That is followed by the reconfiguration of secure authenticated channels that are used for communication among group members; channels that a group member shares with a previous member that is not present in the new membership are severed, and new channels are established with new group members. If the new membership includes new members, then those members request transfer of the service state from old members. Once state transfer is completed, Algorithm Reconf instructs all other modules to re-initialize their local data structures and variables to reflect the new membership and to begin normal operation in the new membership. At that point, Protocol D-PABC and Protocol D-APE unblock and resume normal operation in the new membership.

At a party that is about to be admitted into the group, the view installation procedure begins when the party receives confirmation of its admission from "enough" members of the previous view. That confirmation is accompanied by information about the internal service state at which operations in the previous view ended. The new member establishes secure authenticated channels with other members of the new view. Then, Algorithm Reconf instructs Protocol D-APE to bring the internal service state up-to-date. Protocol D-APE requests state transfer from parties that were members of both the previous view and the new view. Once state transfer is completed, Algorithm Reconf instructs all other modules to re-initialize their local data structures and variables to reflect the new membership and to begin normal operation in the new membership.

5.3 Preliminaries

5.3.1 System Model

In this section, we extend the system model described in Section 2.2 by defining a group abstraction and allowing for the group composition to be altered dynamically. As before, we consider an asynchronous distributed system in which all participants (including the adversary) are computationally bounded and the adversary schedules messages on the network connecting the parties subject only to the condition that messages exchanged between correct parties are eventually delivered unmodified (i.e., every run of the system is *complete*).

The RMan modules are concerned with one universal set $\mathcal{U}$ of parties that wish to be in a replication group, $\mathcal{G}$, for which successive memberships are defined by *views* V_1, V_2, ... Each view is a subset of $\mathcal{U}$ and is an ordered set of identifiers of parties that are members of the view. The parties in a single view V_x have ranks from 0 to $|V_x| - 1$ and are denoted by $P_0, \ldots, P_{|V_x|-1}$. We assume that each party in $\mathcal{U}$ is uniquely identifiable, e.g., by the combination of host IP address and a public key certificate obtained from a trusted certification authority.

For the sake of simplicity, we assume only logical (as opposed to physical) separation between the agreement and execution phases. Any party $P_i \in V_x$ plays the role of an agreement phase participant in view V_x. The RMan modules satisfy their specified properties in view V_x as long as no more than $t < |V_x|/3$ parties are under the control of the adversary. In view V_x, the parties with ranks from 0 to $|V_x|-t-1$ also play the role of execution replicas. Thus, the set $\mathcal{A}$ of agreement phase participants in view V_x consists of $P_0, \ldots, P_{|V_x|-1}$, and the set $\mathcal{E}$ of execution replicas in view V_x consists of $P_0, \ldots, P_{|V_x|-t-1}$. Every pair of parties in a given view is linked by an *authenticated asynchronous channel* that provides message integrity. A group member establishes the channel with another party when the latter becomes a member of the group and severs the channel when that party is removed from the group.

There is also a trusted *dealer* that performs setup of the first view V_0. The members constituting the view V_0 and the fault resilience of the group in that view are given as input to the dealer, which then generates the state information that is used to initialize each party of view V_0. As part of the system initialization, the dealer generates the public key/private key pair for each party in the set $\mathcal{U}$ and gives every party in the view V_0 its private key and the public keys of all parties. At later points in time, the dealer may instantiate other parties in the set $\mathcal{U}$ and initialize them with information about their respective private keys, public keys of all parties in the set $\mathcal{U}$, and the latest view. Those dynamically instantiated parties use that information to attempt to join the group $\mathcal{G}$. We assume that parties are not corrupted by the adversary before they join the group $\mathcal{G}$ for the first time.

5.3.2 Specification of Properties

Protocol D-PABC provides all the properties of the Protocol PABC specified in Section 3.2.2. Similarly, Protocol D-APE provides all the properties of the Protocol APE specified in Section 4.3.1. The combination of the RMan modules provides additional properties relating to view synchrony and group membership. In this section, we specify those properties.

Suppose there is a group $\mathcal{G}$ for which successive memberships are defined by views

V_1, V_2, ... Then, the modules in RMan provide the following properties:

Group Membership Monotonicity If a party P installs view V_x after installing view V_y, then $x > y$.

Group Membership Agreement If P and Q are two correct parties, then the view V_x (if installed) at both parties will have the same membership.

Group Membership Self-Inclusion If a correct party P installs a view V_x, then V_x includes P.

Group Membership Validity If a correct party P installs a view V_x, then any correct party Q such that $Q \in V_x$ will (eventually) install view V_x.

View Synchrony If two parties P and Q both install a new view V_{x+1} in the same previous view V_x, then

(1) **View Synchronous Atomic Delivery**: the sequence of messages *a-delivered* by both P and Q in view V_x is the same.

(2) **View Synchronous State Consistency**: the internal service state just after the installation of view V_{x+1} at both P and Q is the same.

Admission Validity If party P is such that $P \in V_{x+1} - Vx$, then $P_{ID}(x, P, \vartheta) = \texttt{true}$, where P_{ID} is the predicate specifying the admission control policy and ϑ is the admission credential presented by P.

Removal Validity If party P is such that $P \in V_x - Vx + 1$, then there exists ϑ such that $F_{ID}(x, P, \vartheta) = \texttt{true}$, where F_{ID} is the set of predicates specifying the member removal policy.

Liveness If a membership change is desired by $t+1$ correct members of view V_x, then some correct party $P \in V_x$ will eventually install view V_{x+1}.

Protocol Admission **for party** P **and tag** ID **with input parameters** P_{ID} **(predicate specifying the admission control policy) and** ϑ **(admission credential)**

```
initialization:
 1: $curr\_view \leftarrow \{P_1, P_2, \ldots, P_n\}$        {current membership of target group}
 2: $vn \leftarrow 0$                                          {current view number}
 3: $recommend \leftarrow \perp$                               {candidate for admission into the group}
 4: $\mathcal{X} \leftarrow \emptyset$                          {set of new-view messages received at $P \notin curr\_view$}
 5: if $P \notin curr\_view$ then
 6:     $send\_request\_join()$

function $send\_request\_join()$:
 7: compute a signature $\sigma$ on $(ID, \texttt{request-join}, vn, id, \vartheta)$
 8: send the message $(ID, \texttt{request-join}, vn, id, \vartheta, \sigma)$ to all $P_j \in curr\_view$

function $validate_{admission}(view\_num, id, \vartheta_{id})$:
 9: if $(view\_num = vn)$ and $(id \notin curr\_view)$ and $(A_{ID}(view\_num, id, \vartheta_{id}) = \texttt{true})$ then
10:     return true
11: else
12:     return false

function $send\_new\_view_{admission}(next\_view, vn, digest)$:
13: compute a signature $\sigma$ on $(ID, \texttt{new-view}, vn, next\_view, digest)$
14: send the message $(ID, \texttt{new-view}, vn, next\_view, digest, \sigma)$ to each party in $next\_view \setminus$
    $curr\_view$

upon receiving message $(ID, \texttt{new-view}, vn+1, next\_view, digest, \sigma_j)$ from $P_j$ for the first time:

15: if $(P \notin curr\_view)$ and $(P \in next\_view)$ and $(P_j \in curr\_view)$ and
        $(\sigma_j$ is a valid signature by party $P_j$ on $(ID, \texttt{new-view}, vn+1, next\_view, digest))$
    then
16:     $\mathcal{X} \leftarrow \mathcal{X} \cup (ID, \texttt{new-view}, vn+1, next\_view, digest, \sigma_j)$
17:     if $\mathcal{X}$ has $(ID, \texttt{new-view}, vn+1, next\_view, digest, \sigma_k)$ messages from
            $\lfloor \frac{|curr\_view|-1}{3} \rfloor + 1$ distinct $P_k \in curr\_view$ then
18:         $update_{reconf}(\texttt{start-reconf}, [next\_view, vn+1, digest])$
```

Figure 5.2: Protocol Admission for Controlling and Aiding Admission of New Parties (Part I)

```
upon receiving message $(ID, \texttt{request-join}, vn, id, \vartheta_{id}, \sigma_{id})$ from party $id$ for the first time:

    19: if $(id \notin curr\_view)$ and $(recommend = \perp)$ and
            $(\sigma_{id}$ is a valid signature by party $id$ on $(ID, \texttt{request-join}, vn, id, \vartheta_{id}))$ and
                $(validate_{admission}(vn, id, \vartheta_{id}) = true)$ then
    20:     $recommend \leftarrow id$
    21:     $trigger\_view\_change_{gma}(\texttt{add}, id, \vartheta_{id})$

function $update\_view_{admission}(new\_view, view\_num)$: {called by Algorithm Reconf at
            $P \notin curr\_view$}
    22: $curr\_view \leftarrow new\_view$
    23: $vn \leftarrow view\_num$
    24: $send\_request\_join()$

function $done\_reconfigure_{admission}(new\_view, view\_num)$:
    25: $curr\_view \leftarrow new\_view$
    26: $vn \leftarrow view\_num$
    27: $\mathcal{X} \leftarrow \emptyset$
    28: $recommend \leftarrow \perp$
```

Figure 5.3: Protocol Admission for Controlling and Aiding Admission of New Parties (Part II)

5.4 Protocol Admission

We now describe Protocol Admission in detail. The line numbers refer to the detailed protocol description in Figures 5.2–5.3.

Consider a party $P \in \mathcal{U} \setminus \mathcal{G}$ that wants to join the group $\mathcal{G}$. At the time of its creation, P is in its own singleton group, and its sole aim is to become part of $\mathcal{G}$; the replication protocols, Protocol D-PABC and Protocol D-APE, do not begin regular operation until $P \in \mathcal{G}$.

At P, there is an array, *curr_view*, that is initialized with the latest membership of the group $\mathcal{G}$ by the trusted dealer; if the membership changes before P succeeds in joining the group, then the dealer re-initializes the array with the latest membership information.

Pairwise secure authenticated channels are established only among members of $\mathcal{G}$. Hence, until P succeeds in becoming a part of $\mathcal{G}$, all communication between P and the group members is authenticated using digital signatures.

After initialization, the party P invokes the function *send_request_join*() (lines 7– 8). The function sends out a signed `request-join` message containing the party's identifier *id* and its admission credential ϑ to each party $P_i \in curr\_view$. The admission credential is provided as an input parameter to Protocol Admission by the trusted dealer as part of the system startup.

In a given view V_{vn}, a group member P_i recommends at most one party as a candidate for inclusion in the next view. Upon receiving the `request-join` message from a non-member P, a group member P_i that has not yet identified a candidate verifies the signature on the message and invokes the *validate* function passing the current view number vn, P's identifier, and its admission credential as input arguments (line 19). The function (lines 9–12) verifies whether P's credential satisfies the admission control policy in view V_{vn}. The admission policy is specified by a global, polynomial-time computable predicate P_{ID}, which is provided as an input parameter to Protocol Admission by the trusted dealer as part of the system startup. A number of admission control policies are described in [KMT03]. Later, in Section 6.6.3, we describe the implementation of two example admission control policies, one for first-time admission into the group and another for parties seeking readmission into the group.

If the party P satisfies the admission control policy, then Protocol Admission recommends P as a candidate for inclusion in the next view by invoking the *trigger_view_change* interface provided by Protocol GMA and passing P's identifier and admission credential as input arguments (line 21). We guarantee that if, in a given view V_{vn}, a non-member P is recommended as the candidate for inclusion in the next view by $t + 1$ correct members of V_{vn} (where $t < \frac{|V_{vn}|}{3}$), then P will eventually become a member of view V_{vn+1}.

5.5 Algorithm FD

In a view V_x, Protocol PABC ensures safety and liveness as long as no more than $\lfloor \frac{|V_x|-1}{3} \rfloor$ participants in the agreement phase are corrupted by the adversary. Similarly, in view V_x, Protocol APE ensures safety and liveness as long as no more than $\lfloor \frac{|V_x|-1}{2} \rfloor$ participants in the execution phase are corrupted by the adversary. Neither protocol makes the removal of corrupt members from the group a *requirement* to ensure correctness. Despite that fact, there may be situations in which the capability to remove members from the group may be desirable and beneficial from the dependability or performance point of view, e.g., when a group member exhibits a fault that exposes its corruption without any doubt.

When providing the capability to remove members from the group, it is useful to ensure the following two properties. First, it must not be possible for corrupted members alone to effect the removal of a member from the group; if it were, then the corrupted parties may act in collusion to cause the disintegration of the group. Second, if, in a given view, enough correct members want a member to be removed from the group, then that member must be eventually removed.

Algorithm FD helps provide the above two guarantees by controlling the removal of members from the group. In a given view V_{vn}, the algorithm at a group member P_i recommends at most one group member as a candidate for exclusion from the next view. The algorithm ensures the first property above by requiring that if, in a given view V_{vn}, if $P_i \in V_{vn}$ recommends the removal of $P_j \in V_{vn}$, then the recommendation must be *transferable* in that view, i.e., P_i must be able to present information upon the examination of which any correct party $P_k \in V_{vn}$ will be convinced of the validity of the recommendation; we call such information *validation data*. We ensure the second property above by guaranteeing that if, in a given view V_{vn}, a member P_j is recommended as the candidate for exclusion from the next view by $t+1$ correct members of V_{vn} (where $t < \frac{|V_{vn}|}{3}$), then $P_i \notin V_{vn+1}$.

We now describe Protocol FD. The line numbers refer to the detailed protocol description in Figure 5.4. Algorithm FD provides an interface *remove* that can be invoked by other RMan modules when they want to suggest the removal of a group member. The arguments to the interface are the current view number *view_num*, identifier of the group member *id* whose removal is being suggested, and a *removal credential* ϑ. The removal credential ϑ contains three pieces of information: (1) the module that invoked the *remove* interface to suggest the removal of *id*, (2) the fault type exhibited by *id*, and (3) validation data that makes the recommendation for removing *id* from the group transferable in view V_{view_num}.

In a given view V_{vn}, a group member P_i recommends at most one party as a candidate for exclusion from the next view. When the *remove* interface is invoked by some RMan module at a group member P_i in a given view, Algorithm FD recommends *id* as a candidate for exclusion from the next view by invoking the *trigger_view_change* interface provided by Protocol GMA and passing *id* and the removal credential ϑ as input arguments (line 12).

A module that invokes the *remove* interface as described above must also provide a corresponding removal policy that is specified by a global, polynomial-time computable predicate and that can be used by Algorithm FD to verify whether an input removal credential is transferable in a given view (line 6). The set of all such predicates corresponding to various modules and various fault types is provided as an input parameter to Algorithm FD by the trusted dealer as part of the system startup.

5.6 Protocol GMA

We now describe Protocol GMA in detail. The line numbers refer to the detailed protocol description in Figures 5.5–5.6.

Protocol GMA is responsible for maintaining consistent group membership information at all correct members of the group $\mathcal{G}$. The protocol ensures that the view V_x at any correct member of V_x consists of the exact same set of replicas.

In a given view V_{vn}, the protocol at a party $P_i \in V_{vn}$ iterates exactly once through an event loop (lines 12–25) that starts with the acceptance of some valid suggestion to change the membership (line 12). The suggestion may come from the local modules, Protocol Admission or Protocol FD, through the *trigger_view_change* interface provided by Protocol GMA (lines 6–7). The suggestion may also be "adopted" from a valid `pre-proposal` message received from another party. The $verify(type, id, \vartheta, \sigma, j)$ function (lines 8–11) checks the validity of the `pre-proposal` message from party P_j carrying a signature σ (supposedly from P_j) and suggesting either the inclusion of non-member id with admission credential ϑ or the exclusion of member id with removal credential ϑ. The validity check helps ensure the admission validity and removal validity properties specified in Section 5.3.2.

Upon accepting a valid suggestion to change the membership, party P_i sends the suggestion to other members of V_{vn} through a signed `pre-proposal` message with an appropriate admission or removal credential (line 14). Then, the party waits until it receives $n - t$ such `pre-proposal` messages from distinct members of V_{vn} with valid signatures and admission/removal credentials that are valid in the current view V_{vn}. The adoption mechanism mentioned above ensures that if any correct party sends a `pre-proposal` message, it will eventually receive valid `pre-proposal` messages from $n-t$ distinct parties (including itself). The party collects the received `pre-proposal` messages in a *membership change proposal vector* C and proposes C for MVBA. Once the agreement protocol decides on *membership change decision vector* $\bar{C}$ (lines 15–18), the party obtains the next view V_{vn+1} deterministically from the current view and $\bar{C}$ (lines 19–23). By the agreement property of MVBA, any two correct members of V_{vn} that decided will decide on the same value for V_{vn+1}. Hence, the group membership agreement property specified in Section 5.3.2 is satisfied. By the liveness property of MVBA, all correct members of V_{vn} will eventually decide on that value. That helps ensure the group membership validity property for all parties in $V_{vn} \cap V_{vn+1}$.

Suppose a particular membership change is suggested by $t + 1$ correct members of view V_{vn}. Since the `pre-proposal` messages of $n-t$ parties are included in the decision vector of MVBA, at least one of these messages contains that membership change suggestion. Hence, view V_{vn+1} will incorporate that suggestion.

After determining the next view, Protocol GMA notifies Algorithm Reconf to start the sequence of steps that will transition the party from the current view to the new view (line 24). After the notification, Protocol GMA blocks until the transition is completed (line 25).

5.7 Algorithm Reconf and the View Installation Procedure

The notification from Protocol GMA to Algorithm Reconf after the determination of the next view results in the latter coordinating a series of steps that transition the RMan modules from the current view to the next. We call that series of steps the *view installation procedure*. The detailed algorithm description for Algorithm Reconf is given in Figures 5.7–5.8.

The first step in the view installation procedure is to ensure that the Protocol D-PABC and Protocol D-APE at all correct replicas *stabilize*, i.e., suspend their operation in the current view at the same protocol state. That means that, at all correct replicas, Protocol D-PABC must atomically deliver the same set of messages in the same order in a given view. Similarly, at all correct execution replicas, the service state must reflect the execution of all and only those requests specified in the payloads that were atomically delivered in the current and previous views, where the execution order is specified by order of atomic delivery.

5.7.1 Stabilization of Protocol D-PABC

Protocol D-PABC is an extension to the original atomic broadcast protocol presented in Chapter 3, in which the set of communicating parties may be dynamically changed. In addition to the properties provided by Protocol PABC, Protocol D-PABC provides the view synchronous atomic broadcast property specified in Section 5.3.2.

Protocol PABC guarantees *epoch synchrony*, i.e., in a given epoch, all correct parties *a-deliver* the same sequence of payloads (Lemma 7). The protocol also guarantees that if some correct party enters epoch e, then all correct parties eventually enter epoch e (Lemma 3).

Given these properties of Protocol PABC, view synchronous atomic broadcast can be obtained simply by ensuring that all correct parties terminate a given view in the same epoch.

The line numbers in this section refer to the detailed protocol description for Protocol D-PABC given in Figures 5.9–5.13. The figures use the club suite (♣) symbol to indicate the lines or functions that are new or modified compared to the original protocol.

The stabilization of Protocol D-PABC in the current view V_{vn} is triggered when Algorithm Reconf invokes the *stabilize*() interface at Protocol D-PABC. At a group member $P_i \in V_{vn}$, the *stabilize*() method (lines 74–75) `a-broadcast`s a signed `close` payload containing the current view number vn.

The *deliver* function keeps track of the number of `close` payloads from distinct group members that have been passed as arguments to the function in the current view (lines 50–55). When `close` payloads from $t + 1$ distinct group members have been obtained, the function marks the current epoch as the last epoch for the current view V_{vn} by setting the Boolean variable *block* to `true` (line 54). If the protocol is in the parsimonious mode, then once the *deliver* function returns, the protocol will switch to the recovery mode of the current epoch. If the protocol is already in the recovery mode, then it will complete the recovery mode. When part 3 of the recovery mode completes, instead of starting the next epoch (as in the case of the original Protocol PABC), the Protocol D-PABC notifies Algorithm Reconf that stabilization is complete (line 86) and also sends an indication of the sequence number of the last *a-delivery* in the current view; then, Protocol D-PABC blocks (line 87) until Algorithm Reconf notifies it about the completion of the view installation procedure by invoking the *done_reconfigure*() function (lines 66–73). Due to the epoch synchrony property mentioned above, all correct parties will *a-deliver* the $t + 1^{\text{st}}$ `close` payload in the same epoch. Hence, all correct parties will terminate a given view in the same epoch, thereby ensuring view synchrony.

5.7.2 Stabilization of Protocol D-APE and Service State

Protocol D-APE is an extension to the original parsimonious execution protocol presented in Chapter 4, in which the set of execution replicas may be dynamically changed. In addition to the properties provided by Protocol APE, Protocol D-APE provides the view synchronous state consistency property specified in Section 5.3.2.

Recall that Protocol APE performs updates to the internal service state strictly in the atomic delivery order defined by Protocol PABC. In Section 5.7.1, we described how Protocol D-PABC provides view synchronous atomic broadcast. Then, view synchronous state consistency can be obtained simply by ensuring that all correct parties terminate a given view only after bringing their states up to date to reflect the execution of all requests up to the last atomically delivered request for that view.

The line numbers in this section refer to the detailed protocol description for Protocol D-APE given in Figures 5.14–5.19. The figures use the club suite (♣) symbol to indicate the lines or functions that are new or modified compared to the original protocol.

The stabilization of Protocol D-APE in the current view V_{vn} is triggered when Algorithm Reconf invokes the *stabilize*() interface at Protocol D-APE. The input argument to the method invocation indicates the sequence number of the last *a-delivery* in the current view. At a group member $P_i \in V_{vn}$, the *stabilize*() method (line 47) stores that sequence number in the local variable *last*. Thereafter, upon receiving an agreement certificate containing the sequence number *last* or if such a certificate has already been received, the protocol invokes the *finish_view*() function (lines 16–18).

The *finish_view* function (lines 42–46) first ensures that the service state reflects the execution of all requests up to sequence number *last* by invoking the *obtain_reply* function. That is followed by the taking of a checkpoint of the service state. Then, Protocol D-APE sends Algorithm Reconf a notification that stabilization is complete (line 45) along with an indication of the digest of the last checkpoint in the current view. Protocol D-APE then blocks (line 46) until Algorithm Reconf notifies it about the completion of the view installation procedure by invoking the *done_reconfigure*() function (lines 48–55).

5.7.3 Conveying the Next Membership to the Joining Replica

If the next view V_{vn+1} includes new member(s), then upon receiving the notification that Protocol D-APE has stabilized, Algorithm Reconf instructs the Protocol Admission to send the new view and service state information to the new member(s) (Figure 5.7, lines 11–13).

The instruction comes in the form of the *send_new_view* function invocation at Protocol Admission. The function (Figure 5.2, lines 13–14) sends a signed `new-view` message to each new member of V_{vn+1}. The message contains the new view V_{vn+1} and the digest of the service state in which the new member should begin request processing upon joining the group.

Protocol Admission at a party $P \in V_{vn+1} \setminus V_{vn}$ keeps track of `new-view` messages received from members of view V_{vn} (Figure 5.2, lines 15–18). Once P has obtained `new-view` messages from $\frac{|V_{vn}|}{3}+1$ distinct members of V_{vn} with identical information about the next view and the service state digest, party P considers itself a part of group $\mathcal{G}$ from view V_{vn}. Then, the view installation procedure is triggered at P by Protocol Admission notifying Algorithm Reconf (Figure 5.2, line 18). At the new member P, the view installation procedure consists of two steps: (1) establishment of authenticated channels with all other members of V_{vn+1} and (2) obtaining of transfer of the service state.

5.7.4 Reconfiguration of Pairwise Authenticated Channels

As part of the view installation procedure, a new member (i.e., a party $P \in V_{vn+1} \setminus V_{vn}$) establishes pairwise authenticated channels with other members of V_{vn}. A party $P_i \in V_{vn+1} \cap V_{vn}$ severs previously established channels with members excluded from the new view (i.e., with any party $P' \in V_{vn} \setminus V_{vn+1}$). Such reconfiguration of communication channels is initiated when Algorithm Reconf invokes the *reconfigure* function at the underlying network layer passing the new view V_{vn+1} as input argument. At a party $P_i \in V_{vn+1} \cap V_{vn}$, the invocation is made (Figure 5.7, line 15) after Protocol Admission at P_i has sent a `new-view` message to all new members of V_{vn+1}. At a new member, the invocation is made (Figure 5.7, line 17) as the first step of the view installation procedure. Later, in Sections 6.6.2 and 6.6.3, we describe one way of implementing the establishment and reconfiguration of authenticated channels using message authentication codes (MACs).

5.7.5 Obtaining Transfer of Service State at a Joining Replica

After establishing authenticated channels with other members of the new view V_{vn+1}, the next step in the view installation procedure at a party $P \in V_{vn+1} \setminus V_{vn}$ is to obtain transfer of service state. That is the responsibility of the $state\_update_{ape}$ function at Protocol APE. The function (Figure 4.5, lines 56–58) sends out a `state-request` message to all parties in the view V_{vn} and waits for state transfer. Party P can easily verify whether the state transferred is correct; P computes the digest of a copy of the state obtained through state transfer, and then compares the digest with the one that was vouched for by $\frac{|V_{vn}|}{3}$ members of V_{vn} through their `new-view` messages. If the two digests are equal, then the state transferred is correct. At that point, Protocol D-APE notifies Algorithm Reconf that the transfer of service state is complete (Figure 4.5, line 58).

5.7.6 Beginning Regular Operation in the New Membership

The last step in the view installation procedure is to re-initialize the local variables and data structures at the various RMan modules to reflect the new group membership V_{vn+1}. That step, represented by function *begin_regular_operation* at Algorithm Reconf (Figure 5.8, lines 30–36), is taken at a party $P_i \in V_{vn+1} \cap V_{vn}$ upon the reconfiguration of the pairwise authenticated channels (Figure 5.7, line 19). At a party $P \in V_{vn+1} \setminus V_{vn}$, that step is taken after the correct transfer of service state has been obtained (Figure 5.7, line 29).

5.8 Summary

We described a suite of protocols called RMan that defines a group abstraction for the replicas and allows for the dynamic alteration of the composition of the replication group. The suite includes a protocol for controlling and aiding the addition of new members into the group, a protocol for controlling the removal of members from the group, a protocol for ensuring consistency of group membership information at all correct group members, and an algorithm that coordinates the transition from one membership of the group to the next in a way that ensures view synchrony properties. Since our replication protocols were designed never to *require* the removal of faulty replicas to make progress, the ability provided by RMan to dynamically reconfigure the replication group can be used very selectively and conservatively.

Protocol FD for party P_i and tag ID with input parameters F_{ID} (set of predicates specifying the member removal policy)

initialization:

```
 1: $curr\_view \leftarrow \{P_1, P_2, \ldots, P_n\}$                {current group membership}
 2: $vn \leftarrow 0$                                                  {current view number}
 3: $remove \leftarrow \perp$                        {candidate for removal from the group}
```

function $validate_{failure\_detect}(view\_num, id, \vartheta)$**:**

```
 4: $ft \leftarrow \vartheta.failure\_type$
 5: $mt \leftarrow \vartheta.module\_type$
 6: if $(id \in curr\_view)$ and $(view\_num = vn)$ and
        $(\exists F_{ID.ft.mt} \in F_{ID}$ such that $F_{ID.ft.mt}(view\_num, id, \vartheta.proof) = \texttt{true})$ then
 7:     return true
 8: else
 9:     return false
```

function $remove_{failure\_detect}(id, \vartheta)$**:** {called by some module at P_i to suggest removal of id from the group}

```
10: if $remove = \perp$ then
11:     $remove \leftarrow id$
12:     $trigger\_view\_change_{gma}(\texttt{remove}, id, \vartheta)$
```

function $done\_reconfigure_{admission}(new\_view, view\_num)$**:**

```
13: $curr\_view \leftarrow new\_view$
14: $vn \leftarrow view\_num$
15: $remove \leftarrow \perp$
```

Figure 5.4: Algorithm FD for Aiding Member Removal

```
Protocol GMA for party $P_i$ and tag $ID$

initialization:
 1: $curr\_view \leftarrow \{P_1, P_2, \ldots, P_n\}$          {current group membership}
 2: $vn \leftarrow 0$                                  {current view number}
 3: $init\_view()$

function $init\_view()$:
 4: $p \leftarrow 0$                 {number of pre-proposal messages received}
 5: $(type_j, id_j, \vartheta_j, \sigma_j) \leftarrow (\perp, \perp, \perp, \perp)$    $(1 \leq j \leq n)$

function $trigger\_view\_change_{gma}(type, id, \vartheta)$:
 6: if $(type_i, id_i, \vartheta_i) = (\perp, \perp, \perp)$ then
 7:    $(type_i, id_i, \vartheta_i) \leftarrow (type, id, \vartheta)$

function $verify(type, id, \vartheta, \sigma, j)$:
 8: if $\sigma$ is a valid signature by $P_j$ on $(ID,$ pre-proposal$, vn, type, id, \vartheta)$ and
       $\big((type =$ add and $validate_{admission}(vn, id, \vartheta) =$ true$)$ or
           $(type =$ remove and $validate_{failure\_detect}(vn, id, \vartheta) =$ true$)\big)$ then
 9:    return true
10: else
11:    return false

forever:
12: wait for $(type_i, id_i, \vartheta_i) \neq (\perp, \perp, \perp)$
13: compute a signature $\sigma$ on $(ID,$ pre-proposal$, vn, type_i, id_i, \vartheta_i)$
14: send the message $(ID,$ pre-proposal$, vn, type_i, id_i, \vartheta_i, \sigma)$ to all parties
15: wait for $p = n - t$
16: $C \leftarrow [(type_1, id_1, \vartheta_1, \sigma_1), \ldots, (type_n, id_n, \vartheta_n, \sigma_n)]$
17: propose $C$ for multi-valued Byzantine agreement with tag $ID|$membership$.vn$
       and predicate $Q_{ID|\texttt{membership}.vn}$
18: wait for the Byzantine agreement protocol with tag $ID|$membership$.vn$
       to decide some $\bar{C} = [(\bar{type}_1, \bar{id}_1, \bar{\vartheta}_1, \bar{\sigma}_1), \ldots, (\bar{type}_n, \bar{id}_n, \bar{\vartheta}_n, \bar{\sigma}_n)]$
19: $next\_view \leftarrow curr\_view$
20: for $id \in \bigcup_{j=1}^{n} \{\bar{id}_j \mid \bar{type}_j =$ add$\}$, in some deterministic order do
21:    $next\_view \leftarrow next\_view \bigcup \{id\}$
22: for $id \in \bigcup_{j=1}^{n} \{\bar{id}_j \mid \bar{type}_j =$ remove$\}$, in some deterministic order do
23:    $next\_view \leftarrow next\_view \setminus \{id\}$
24: $update_{reconf}($start-reconf$, [next\_view, vn + 1])$
25: wait for $p = 0$
```

Figure 5.5: Protocol GMA for Group Membership Agreement (Part I)

```
upon receiving message ($ID$, pre-proposal, $vn, type, id, \vartheta, \sigma$) from $P_j$ for the first time:

26: if ($verify(type, id, \vartheta, \sigma, j) = $ true) then
27:     $(type_j, id_j, \vartheta_j, \sigma_j) \leftarrow (type, id, \vartheta, \sigma)$
28:     $p \leftarrow p + 1$
29:     if $(type_i, id_i, \vartheta_i) = (\perp, \perp, \perp)$ then
30:         $(type_i, id_i, \vartheta_i) \leftarrow (type_j, id_j, \vartheta_j)$

function $done\_reconfigure_{gma}(new\_view, num)$:
31: $n \leftarrow |next\_view|$
32: $t \leftarrow \lfloor \frac{n-1}{3} \rfloor$
33: $vn \leftarrow num$
34: $curr\_view \leftarrow new\_view$
35: $i \leftarrow find(next\_view, P_i)$
36: for $j = 1, \ldots, n$ do
37:     $P_j \leftarrow next\_view[j]$
38: $init\_view()$
```

Let $Q_{ID|\texttt{membership}.v}$ be the following predicate:

$$Q_{ID|\texttt{membership}.v}\big([(type_1, id_1, \vartheta_1, \sigma_1), \ldots, (type_n, id_n, \vartheta_n, \sigma_n)]\big) \equiv$$

$\big($for at least $n - t$ distinct j, $type_j \neq \perp\big)$ **and**

$\big($for all $j = 1, \ldots, n$, either $(type_j = \perp)$ **or** $(verify(type_j, id_j, \vartheta_j, \sigma_j, j) = \texttt{true})$ $\big)$

Figure 5.6: Protocol GMA for Group Membership Agreement (Part II)

```
Algorithm Reconf for party $P$ and tag $ID$

initialization:
 1: $curr\_view \leftarrow \{P_1, P_2, \ldots, P_n\}$        {current membership of target group}
 2: $vn \leftarrow 0$                                          {current view number}
 3: $next\_view \leftarrow \perp$                              {next group membership}

function $update_{reconf}(type, arg)$:
 4: if $P \in curr\_view$ then
 5:   if ($type =$ start-reconf) then {Protocol GMA triggers transition to next view}
 6:     $next\_view \leftarrow arg.view$                {$arg \equiv$ [next view, next view number]}
 7:     $vn \leftarrow arg.view\_num$
 8:     $stabilize_{pabc}()$
 9:   else if ($type =$ pabc-stabilized) then
10:     $stabilize_{ape}(arg)$        {$arg \equiv$ last a-delivery sequence number in current view}
11:   else if ($type =$ ape-stabilized) then
12:     if $next\_view \setminus curr\_view \neq \emptyset$ then {new member(s) admitted}
13:       $send\_new\_view_{admission}(next\_view, vn, arg)$     {$arg \equiv$ stabilized service state
          digest}
14:     else
15:       $reconfigure_{network}(next\_view)$
16:   else if ($type =$ new_view_sent) then
17:     $reconfigure_{network}(next\_view)$
18:   else if ($type =$ network_ready) then
19:     $begin\_normal\_operation()$                {start regular operation in next view}
20: else {$P$ not member of target group yet}
21:   if ($type$ = start-reconf) then {Protocol Admission triggers transition to next
      view}
22:     $next\_view \leftarrow arg.view$ {$arg \equiv$ [next view, next view number, stable state digest]}
23:     $vn \leftarrow arg.view\_num$
24:     $state\_digest \leftarrow arg.digest$
25:     $reconfigure_{network}(next\_view)$
26:   else if ($type =$ network_ready) then
27:     $state\_update_{ape}(curr\_view, state\_digest)$
28:   else if ($type =$ ape_state_updated) then
29:     $begin\_normal\_operation()$                {start regular operation in next view}
```

Figure 5.7: Algorithm Reconf for Coordinating Group Reconfiguration (Part I)

```
function $begin\_normal\_operation()$:
    30: $done\_reconfigure_{admission}(next\_view, vn)$
    31: $done\_reconfigure_{failure\_detect}(next\_view, vn)$
    32: $done\_reconfigure_{gma}(next\_view, vn)$
    33: $done\_reconfigure_{pabc}(next\_view, vn)$
    34: $done\_reconfigure_{ape}(next\_view, vn)$
    35: $curr\_view \leftarrow next\_view$
    36: $next\_view \leftarrow \perp$

function $update\_view(view, num)$: {external directive at $P \notin curr\_view$ for
        updating $curr\_view$}
    37: $curr\_view \leftarrow view$
    38: $vn \leftarrow num$
    39: $update\_view_{admission}(view, num)$ {notify Protocol Admission of change in target group}
```

Figure 5.8: Algorithm Reconf for Coordinating Group Reconfiguration (Part II)

```
Protocol D-PABC for party $P_i$ and tag $ID$

initialization:
 1: $\mathcal{I} \leftarrow []$        {initiation queue, list of a-broadcast but not a-delivered payloads}
 2: $\mathcal{D} \leftarrow \emptyset$        {set of a-delivered payloads}
 3: $e \leftarrow 0$        {current epoch}
 4: $vn \leftarrow 0$ {♣}        {current view number}
 5: $curr\_view \leftarrow \{P_1, P_2, \ldots, P_n\}$ {♣}        {current membership of target group}
 6: $init\_view()$
 7: $init\_epoch()$

function $init\_view()$: {♣}
 8: $block \leftarrow$ false        {indicates if the procedure of switching to next view is currently
    underway}
 9: $d \leftarrow 0$        {number of a-deliveries thus far in current view}
10: $close[j] \leftarrow$ false$, 1 \le j \le n$        {indicates if a close payload initiated by $P_j$ has been
    a-delivered}
11: $closes \leftarrow 0$        {number of close payloads initiated by distinct parties a-delivered}
12: if $(ID,$ close$, vn, i, \sigma_i) \in \mathcal{I}$ then
13:    $remove((ID,$ close$, vn, i, \sigma_i), \mathcal{I})$

function $init\_epoch()$:
12: $l \leftarrow (e \bmod n) + 1$        {$P_l$ is leader of epoch $e$}
13: $log \leftarrow []$        {array of size $B$ containing payloads committed in current epoch}
14: $s \leftarrow 0$        {sequence number of next payload within epoch}
15: $complained \leftarrow$ false        {indicates if this party already complained about $P_l$}
16: $start\_recovery \leftarrow$ false        {signals the switch to the recovery mode}
17: $c \leftarrow 0$        {number of complain messages received for epoch leader}
18: $\mathcal{S} \leftarrow \mathcal{D}$        {set of a-delivered or already sc-broadcast payloads at $P_l$}

upon $(ID,$ in$,$ a-broadcast$, m)$:
19: $append(m, \mathcal{I})$
20: if $\neg block$ then {♣}
21:    send $(ID,$ initiate$, vn, e, m)$ to $P_l$
22:    $update_{\mathcal{F}_l}($initiate$, m)$
```

Figure 5.9: Protocol D-PABC for Reconfigurable Atomic Broadcast (Part I)

```
forever: {parsimonious mode}
23: if $\neg$complained then {leader $P_l$ is not suspected}
24:     initialize an instance of strong consistent broadcast with tag $ID|\texttt{bind}.vn.e.s$
25: $m \leftarrow \perp$
26: if $i = l$ then
27:     wait for $timeout(T)$ or receipt of a message $(ID, \texttt{initiate}, vn, e, m)$ such that
        $m \notin \mathcal{S}$
28:     if $timeout(T)$ then
29:         $m \leftarrow \texttt{dummy}$
30:     else
31:         $\mathcal{S} \leftarrow \mathcal{S} \cup \{m\}$
32:         $stop(T)$
33:     sc-broadcast the message $m$ with tag $ID|\texttt{bind}.vn.e.s$
34: wait for $start\_recovery$ or sc-delivery of some $m$ with tag $ID|\texttt{bind}.vn.e.s$
      such that $m \notin \mathcal{D} \cup log$
35: if $start\_recovery$ then
36:     $recovery()$
37: else
38:     $log[s] \leftarrow m$
39:     if $s \geq 2$ then
40:         $update_{\mathcal{F}_l}(\texttt{deliver}, log[s-2])$
41:         $deliver(log[s-2])$
42:     if $i = l$ and $(log[s] \neq \texttt{dummy}$ or $(s > 0$ and $log[s-1] \neq \texttt{dummy}))$ then
43:         $start(T)$
44:     $s \leftarrow s + 1$
45:     if $s \bmod B = 0$ then
46:         $recovery()$

function $deliver(m)$:
47: if $m \neq \texttt{dummy}$ then
48:     $remove(m, \mathcal{I})$
49:     $\mathcal{D} \leftarrow \mathcal{D} \cup \{m\}$
50:     if $m = (ID, \texttt{close}, vn, j, \sigma_j)$ and $\sigma_j$ is a valid signature by $P_j$ on $(ID, \texttt{close}, vn, j)$
          and $close[j] = \texttt{false}$ then {♣}
51:         $close[j] \leftarrow \texttt{true}$ {♣}
52:         $closes \leftarrow closes + 1$ {♣}
53:         if $closes = t + 1$ then {♣}
54:             $block \leftarrow \texttt{true}$ {♣}
55:             $start\_recovery \leftarrow \texttt{true}$ {♣}
56:     else
57:         output $(ID, \texttt{out}, \texttt{a-deliver}, m)$
58:         $d \leftarrow d + 1$ {♣}
```

Figure 5.10: Protocol D-PABC for Reconfigurable Atomic Broadcast (Part II)

```
**function** $complain()$:
    59: send $(ID, \texttt{complain}, vn, e)$ to all parties
    60: $complained \leftarrow \texttt{true}$

**upon** receiving message $(ID, \texttt{complain}, vn, e)$ from $P_j$ for the first time:
    61: $c \leftarrow c + 1$
    62: **if** $(c = t + 1)$ **and** $\neg complained$ **then**
    63:     $complain()$
    64: **else if** $c = 2t + 1$ **then**
    65:     $start\_recovery \leftarrow \texttt{true}$

**function** $done\_reconfigure_{pabc}(new\_view, num)$: $\{\clubsuit\}$
    66: $n \leftarrow |new\_view|$
    67: $t \leftarrow \lfloor \frac{n-1}{3} \rfloor$
    68: $vn \leftarrow num$
    69: $curr\_view \leftarrow new\_view$
    70: $i \leftarrow find(new\_view, P_i)$
    71: **for** $j = 1, \ldots, n$ **do**
    72:     $P_j \leftarrow new\_view[j]$
    73: $block \leftarrow \texttt{false}$

**function** $stabilize_{pabc}()$: $\{\clubsuit\}$
    74: compute a signature $\sigma$ on $(ID, \texttt{close}, vn, i)$
    75: $(ID, \texttt{in}, \texttt{a-broadcast}, (ID, \texttt{close}, vn, i, \sigma))$
```

Let $Q_{ID|\texttt{watermark}.vn.e}$ be the following predicate:

$$Q_{ID|\texttt{watermark}.vn.e}\Big(\big[(s_1, C_1, \sigma_1), \ldots, (s_n, C_n, \sigma_n)\big]\Big) \equiv$$
$$\big(\text{for at least } n - t \text{ distinct } j,\ s_j \neq \bot\big) \textbf{ and}$$
$$\big(\text{for all } j = 1, \ldots, n, \text{ either } s_j = \bot \textbf{ or}$$
$$(\sigma_j \text{ is a valid signature by } P_j \text{ on } (ID, \texttt{committed}, vn, e, s_j, C_j) \textbf{ and}$$
$$(s_j = -1 \text{ or } C_j \text{ completes the } \textit{sc-broadcast} \text{ with tag } ID|\texttt{bind}.vn.e.s_j))\big)$$

Let $Q_{ID|\texttt{deliver}.vn.e}$ be the following predicate:

$$Q_{ID|\texttt{deliver}.vn.e}\Big(\big[(\mathcal{I}_1, \sigma_1), \ldots, \mathcal{I}_n, \sigma_n)\big]\Big) \equiv \text{ for at least } n - t \text{ distinct } j,$$
$$\big(\mathcal{I}_j \cap \mathcal{D} = \emptyset \textbf{ and } \sigma_j \text{ is a valid signature by } P_j \text{ on } (ID, \texttt{queue}, vn, e, j, \mathcal{I}_j)\big)$$

Figure 5.11: Protocol D-PABC for Reconfigurable Atomic Broadcast (Part III)

```
function $recovery()$:
    {Part 1: agree on watermark}
    76: compute a signature $\sigma$ on $(ID, \texttt{committed}, vn, e, s-1)$
    77: send the message $(ID, \texttt{committed}, vn, e, s-1, C, \sigma)$ to all parties, where $C$ denotes
            the bit string that completes the sc-broadcast with tag $ID|\texttt{bind}.vn.e.(s-1)$
    78: $(s_j, C_j, \sigma_j) \leftarrow (\perp, \perp, \perp) \qquad (1 \le j \le n)$
    79: wait for $n-t$ messages $(ID, \texttt{committed}, vn, e, s_j, C_j, \sigma_j)$ from distinct $P_j$ such that $C_j$
        completes
           the sc-broadcast instance $ID|\texttt{bind}.vn.e.s_j$ and $\sigma_j$ is a valid signature on
        $(ID, \texttt{committed}, vn, e, s_j)$
    80: $W \leftarrow [(s_1, C_1, \sigma_1), \ldots, (s_n, C_n, \sigma_n)]$
    81: propose $W$ for MVBA with tag $ID|\texttt{watermark}.vn.e$ and predicate $Q_{ID|\texttt{watermark}.vn.e}$
    82: wait for MVBA with tag $ID|\texttt{watermark}.vn.e$ to decide some $\bar{W} =$
        $[(\bar{s}_1, \bar{C}_1, \bar{\sigma}_1), \ldots, (\bar{s}_n, \bar{C}_n, \bar{\sigma}_n)]$
    83: $w \leftarrow \max\{\bar{s}_1, \ldots, \bar{s}_n\} - 1$
    {Part 2: synchronize up to watermark}
    84: $s' \leftarrow s - 2$
    85: while $s' \le \min\{s-1, w\}$ do
    86:    if $s' \ge 0$ then
    87:       $deliver(log[s'])$
    88:    $s' \leftarrow s' + 1$
    89: if $s > w$ then
    90:    for $j = 1, \ldots, n$ do
    91:       $u \leftarrow \max\{s_j, \bar{s}_j\}$
    92:       $\mathcal{M} \leftarrow \{M_v\}$ for $v = u, \ldots, w$, where $M_v$ completes the sc-broadcast instance
              $ID|\texttt{bind}.vn.e.v$
    93:       send message $(ID, \texttt{complete}, \mathcal{M})$ to $P_j$
    94: while $s \le w$ do
    95:    wait for a message $(ID, \texttt{complete}, \bar{\mathcal{M}})$ such that $\bar{M}_s \in \bar{\mathcal{M}}$ completes sc-broadcast
              with tag $ID|\texttt{bind}.vn.e.s$
    96:    use $\bar{M}_s$ to sc-deliver some $m$ with tag $ID|\texttt{bind}.vn.e.s$
    97:    $deliver(m)$
    98:    $s \leftarrow s + 1$
```

Figure 5.12: Protocol D-PABC for Reconfigurable Atomic Broadcast (Part IV)

```
function $recovery()$: (continued...)
  {Part 3: deliver some messages}
    76: compute a digital signature $\sigma$ on $(ID, \texttt{queue}, vn, e, i, \mathcal{I})$
    77: send the message $(ID, \texttt{queue}, vn, e, i, \mathcal{I}, \sigma)$ to all parties
    78: $(\mathcal{I}_j, \sigma_j) \leftarrow (\perp, \perp) \qquad (1 \leq j \leq n)$
    79: wait for $n - t$ messages $(ID, \texttt{queue}, vn, e, j, \mathcal{I}_j, \sigma_j)$ from distinct $P_j$ such that
          $\sigma_j$ is a valid signature from $P_j$ and $\mathcal{I}_j \cap \mathcal{D} = \emptyset$
    80: $Q \leftarrow [(\mathcal{I}_1, \sigma_1), \ldots, (\mathcal{I}_n, \sigma_n)]$
    81: propose $Q$ for MVBA with tag $ID|\texttt{deliver}.vn.e$ and predicate $Q_{ID|\texttt{deliver}.vn.e}$
    82: wait for MVBA with tag $ID|\texttt{deliver}.vn.e$ to decide some $\bar{Q} = [(\bar{\mathcal{I}}_1, \bar{\sigma}_1), \ldots, (\bar{\mathcal{I}}_n, \bar{\sigma}_n)]$
    83: for $m \in \bigcup_{j=1}^{n} \bar{\mathcal{I}}_j \setminus \mathcal{D}$, in some deterministic order do
    84:     $deliver(m)$
    85: if $block$ then {♣}
    86:     $update_{reconf}(\texttt{pabc-stabilized}, d)$ {♣}
    87:     wait for $block = \texttt{false}$ {♣}
    88:     $e \leftarrow 0$ {♣}
    89:     $init\_view()$ {♣}
    90: $init\_epoch()$
    91: for $m \in \mathcal{I}$ do
    92:     send $(ID, \texttt{initiate}, vn, e, m)$ to $P_l$
```

Figure 5.13: Protocol D-PABC for Reconfigurable Atomic Broadcast (Part V)

Protocol D-APE for execution replica P_i with input parameter δ (checkpoint interval)

```
initialization:
 1: $curr\_view \leftarrow \{P_1, P_2, \ldots, P_n\}$ {♣}        {current membership of target group}
 2: $vn \leftarrow 0$ {♣}                                          {current view number}
 3: $init\_view()$

function $init\_view()$:
 4: $\mathcal{PC} \leftarrow \{P_1, P_2, \ldots, P_{t+1}\}$        {current primary committee}
 5: $corrupt \leftarrow \emptyset$        {set of replicas for which $P_i$ has sent convict messages}
 6: $slow \leftarrow \emptyset$   {set of replicas for which $P_i$ has sent indict messages since last reset}
 7: $c \leftarrow 0$                      {counter indicating number of resets of the set $slow$}
 8: $certified \leftarrow \emptyset$ {set of sequence numbers of requests for which $P_i$ has reply certificates}
 9: $\mathcal{R} \leftarrow []$           {array of size $(\delta - 1)$ containing operations requested by $\mathcal{AS}$}
10: $s \leftarrow 0$                      {sequence number in the last agree message received from $\mathcal{AS}$}
11: $u \leftarrow 0$                      {sequence number up to which $P_i$'s state is updated}
12: $replies \leftarrow \emptyset$        {set of reply messages received from replicas}
13: $suspects \leftarrow \emptyset$       {set of suspect messages for various replicas received}
14: $must\_do \leftarrow \emptyset$       {set of sequence numbers of requests that $P_i$ must execute even if
    $P_i \notin \mathcal{PC}$}
15: $last \leftarrow -1$ {♣}              {sequence number of final request to execute in current view}
```

Figure 5.14: Protocol D-APE for Reconfigurable Parsimonious Execution (Part I)

```
forever:
16: wait for (receipt of first-time agree$(s+1)$ or retransmit agree$(s)$ message from $\mathcal{AS}$)
              or $(s = last)$ {♣}
17: if $s = last$ then {♣}
18:    finish_view() {♣}
   {Mode 1: Parsimonious Normal Mode}
19: else if message received was first-time agree$(s+1)$ and $(s+1)$ mod $\delta \neq 0$ then
20:    $s \leftarrow s+1$
21:    $\mathcal{R}[s \bmod \delta] \leftarrow o$, where $o$ is the service operation specified in the agree message
22:    if $P_i \in \mathcal{PC}$ then
23:       $r \leftarrow$ obtain_reply()
24:       send_reply($r$, normal)
   {Mode 2: Parsimonious Audit Mode}
25: else {retransmit agree$(s)$ or first-time agree$(s+1)$ checkpoint request}
26:    if message received was first-time agree$(s+1)$ and $(s+1)$ mod $\delta = 0$ then
       {checkpoint}
27:       $s \leftarrow s+1$
28:    repeat
29:       oldpc $\leftarrow \mathcal{PC}$
30:       if $(P_i \in \mathcal{PC})$ then
31:          $r \leftarrow$ obtain_reply()
32:          send_reply($r$, audit)
33:       update$_{\mathcal{FD}}$(start-monitor, $s$)
34:       wait for $s \in$ certified or $s \in$ must_do or oldpc $\neq \mathcal{PC}$
35:       update$_{\mathcal{FD}}$(stop-monitor, $s$)
36:    until $s \in$ certified or $s \in$ must_do
37:    if ($s \in$ certified) then
38:       send reply certificate for agree$(s)$ to $\mathcal{AS}$
   {Mode 3: Recovery Mode}
39:    else {$s \in$ must_do}
40:       $r \leftarrow$ obtain_reply()
41:       send_reply($r$, recover)
```

Figure 5.15: Protocol D-APE for Reconfigurable Parsimonious Execution (Part II)

```
function $finish\_view()$: {♣}
    Require: $s = last$                                         {precondition}
    42: $r \leftarrow obtain\_reply()$    {update state to reflect execution of all requests up to seq.
        number $last$}
    43: $send\_reply(r, \texttt{recovery})$
    44: $r \leftarrow checkpoint()$    {compute state digest after all requests in current view executed}
    45: $update_{reconf}(\texttt{ape-stabilized}, r)$    {notify Protocol Reconf that stabilization is
        complete}
    46: wait for $last = -1$

function $stabilize_{ape}(d)$: {♣}
    47: $last \leftarrow d$

function $done\_reconfigure_{ape}(new\_view, num)$: {♣}
    48: $n \leftarrow |new\_view|$
    49: $t \leftarrow \lfloor \frac{n-1}{3} \rfloor$
    50: $vn \leftarrow num$
    51: $curr\_view \leftarrow new\_view$
    52: $i \leftarrow find(new\_view, P_i)$
    53: for $j = 1, \ldots, n$ do
    54:     $P_j \leftarrow new\_view[j]$
    55: $init\_view()$

function $state\_update_{ape}(view, digest)$: {♣}   {called by Algorithm Reconf when admitted
        to the target group}
    56: send the message $(\texttt{state-request}, digest)$ to all $P_k \in view$
    57: wait for receipt of state updates such that digest of internal state after applying them
        equals $digest$
    58: $update_{reconf}(\texttt{ape-state-update})$

function $send\_reply(r, mode)$:
    59: if $(mode = \texttt{normal})$ then
    60:     send the message $(\texttt{reply}, vn, i, s, r)$ to $\mathcal{AS}$
    61: else
    62:     send the message $\langle \texttt{reply}, vn, i, s, r \rangle_{\sigma_i}$ to all replicas
```

Figure 5.16: Protocol D-APE for Reconfigurable Parsimonious Execution (Part III)

```
function $obtain\_reply()$:
63: if $\neg\big(last = s$ and $u \neq s\big)$ and
        $\exists\, r$: $\big(replies$ contains $\langle \texttt{reply}, vn, k, s, r\rangle_{\sigma_k}$ messages from $t+1$ distinct $P_k$ or
          $(\texttt{reply}, vn, i, s, r) \in replies\big)$ then {♣}
64:     return $r$
65: else {actually need to execute request to obtain result}
66:     if $u \neq (s-1)$ then
67:         $state\_update()$
68:     if $(s \bmod \delta \neq 0)$ then
69:         $o \leftarrow \mathcal{R}[s \bmod \delta]$
70:         $r \leftarrow execute(o)$
71:     else
72:         $r \leftarrow checkpoint()$                {take checkpoint, and return digest of internal state}
73:     $u \leftarrow s$
74:     $store\_reply((\texttt{reply}, vn, i, s, r), i)$
75:     return $r$

function $state\_update()$: {update state to reflect execution of requests up to seq. number $s-1$}

76: $stable \leftarrow absolute(s/\delta) * \delta$
77: if $(u < stable)$ then
78:     find digest $r$ : $(replies$ contains $\langle \texttt{reply}, vn, k, stable, r\rangle_{\sigma_k}$ messages from $t+1$ distinct
        $P_k)$
79:     $certifiers \leftarrow$ set of $t+1$ distinct $P_k$ whose reply messages form reply certificate for
        $stable$
80:     send the message $(\texttt{state-request}, stable)$ to all $P_k \in certifiers$
81:     wait for receipt of state updates such that digest of internal state after applying
        them equals $r$
82: for $u = (max(stable, u) + 1), \ldots, (s-1)$ do
83:     $o \leftarrow \mathcal{R}[u \bmod \delta]$
84:     $execute(o)$

upon receiving message $\langle \texttt{reply}, vn, j, s_j, r_j\rangle_{\sigma_j}$ from $P_j \notin corrupt$ for the first time:

85: if $i \neq j$ then
86:     $store\_reply(m, j)$
```

Figure 5.17: Protocol D-APE for Reconfigurable Parsimonious Execution (Part IV)

```
function $store\_reply(m, j)$:        where $m = \begin{cases} \langle \texttt{reply}, vn, j, s_j, r_j \rangle_{\sigma_j} & \text{if } j \neq i \\ (\texttt{reply}, vn, i, s_i, r_i) & \text{otherwise} \end{cases}$
    87: $replies \leftarrow replies \cup \{m\}$
    88: if $j \neq i$ then
    89:    for replicas $P_k$ s.t. $(\langle \texttt{suspect}, vn, k, j, s_j, c \rangle_{\sigma_k} \in suspects)$ do
    90:       forward $m$ to $P_k$
    91: if $\exists\ r$: $replies$ contains $\langle \texttt{reply}, vn, k, s_j, r \rangle_{\sigma_k}$ messages from $t+1$ distinct $P_k$ then
    92:    $certificate \leftarrow$ set of $\langle \texttt{reply}, vn, k, s_j, r \rangle_{\sigma_k}$ messages from $t+1$ distinct $P_k$
    93:    if $(s_j \notin certified)$ then
    94:       $certified \leftarrow certified \cup \{s_j\}$
    95:       if $(s_j \in must\_do)$ then
    96:          send $certificate$ to $\mathcal{AS}$
    97: $update_{\mathcal{FD}}(\texttt{got-reply}, j, s_j)$

function $pc\_refresh()$:
    98: if $|corrupt \ \cup slow| > t$ then
    99:    $c \leftarrow c + 1$
    100:    $slow \leftarrow \emptyset$
    101:    $suspects \leftarrow \emptyset$
    102: $\mathcal{PC} \leftarrow$ set of $(t+1)$-lowest-ranked replicas that are neither in $slow$ nor in $corrupt$

function $fault\_report(k, type, s_k)$:
    103: if $type = \texttt{mute-suspect}$ then
    104:    send $\langle \texttt{suspect}, vn, i, k, s_k, c \rangle_{\sigma_i}$ to all replicas
    105: else if $type = \texttt{implicate}$ then
    106:    find $P_j$ : $(P_j \notin corrupt$ and $\langle \texttt{reply}, vn, j, s_k, r_j \rangle_{\sigma_j} \in replies$ and
                 $\langle \texttt{reply}, vn, k, s_k, r_k \rangle_{\sigma_k} \in replies$ and $r_k \neq r_j)$
    107:    $proof \leftarrow \{\langle \texttt{reply}, vn, j, s_k, r_j \rangle_{\sigma_j}, \langle \texttt{reply}, vn, k, s_k, r_k \rangle_{\sigma_k}\}$
    108:    send $(\texttt{implicate}, vn, s_k, proof)$ to all replicas
    109:    $must\_do \leftarrow must\_do \cup \{s_k\}$
    110: else $\{type = \texttt{convict}\}$
    111:    $certificate \leftarrow$ set of $\langle \texttt{reply}, vn, j, s_k, r \rangle_{\sigma_j}$ messages from $t+1$ distinct $P_j$
    112:    $proof \leftarrow certificate \cup \{\langle \texttt{reply}, vn, k, s_k, r_k \rangle_{\sigma_k}\}$
    113:    send $(\texttt{convict}, vn, k, s_k, proof)$ to all replicas
    114:    $corrupt \leftarrow corrupt \cup \{P_k\}$
    115:    $must\_do \leftarrow must\_do \cup \{s_k\}$
    116:    $pc\_refresh()$
```

Figure 5.18: Protocol D-APE for Reconfigurable Parsimonious Execution (Part V)

```
upon receiving message $\langle \texttt{suspect}, vn, j, k, s_j, c\rangle_{\sigma_j}$ from $P_j \notin corrupt$ for the first time:
    117: if $P_k$'s reply message for $\texttt{agree}(s_j)$ is in $replies$ then
    118:     forward $P_k$'s reply message for $\texttt{agree}(s_j)$ to replica $P_j$
    119: $suspects \leftarrow suspects \cup \{\langle \texttt{suspect}, vn, j, k, s_j, c\rangle_{\sigma_j}\}$
    120: if $P_k \notin slow$ and $\langle \texttt{suspect}, vn, h, k, s_j, c\rangle_{\sigma_h}$ messages from $n-t$ distinct $P_h$ are in
         $suspects$ then
    121:     $not\_responsive(s_j, k)$

upon receiving message $(\texttt{indict}, vn, k, s_j, c, proof)$ from $P_j \notin corrupt$:
    122: if $(s_j \notin must\_do)$ or $(P_k \notin corrupt \cup slow)$ then
    123:     if $(proof$ contains $\langle \texttt{suspect}, vn, h, k, s_j, c\rangle_{\sigma_h}$ messages from $n-t$ distinct $P_h)$ then
    124:         $suspects \leftarrow suspects \cup proof$
    125:         $not\_responsive(s_j, k)$

function $not\_responsive(s', k)$:
    126: send $(\texttt{indict}, vn, k, s', c, proof)$ to all replicas, where $proof$ is the set of
         $\langle \texttt{suspect}, vn, j, k, s', c\rangle_{\sigma_j}$
             messages from $n-t$ distinct $P_j$
    127: $slow \leftarrow slow \cup \{P_k\}$
    128: $must\_do \leftarrow must\_do \cup \{s'\}$
    129: $pc\_refresh()$

upon receiving message $(\texttt{implicate}, vn, s_j, proof)$ from $P_j \notin corrupt$:
    130: if $\exists\ P_k, P_h : \langle \texttt{reply}, vn, k, s_j, r_k\rangle_{\sigma_k} \in proof$ and $\langle \texttt{reply}, vn, h, s_j, r_h\rangle_{\sigma_h} \in proof$ and
         $r_h \neq r_k$ then
    131:     for $m \in proof \setminus replies$ do
    132:         $store\_reply(m, \text{sender}(m))$

upon receiving message $(\texttt{convict}, vn, k, s_j, proof)$ from $P_j \notin corrupt$:
    133: if $proof$ contains $\langle \texttt{reply}, vn, h, s_j, r\rangle_{\sigma_h}$ messages from $t+1$ distinct $P_h$ and
          $\langle \texttt{reply}, vn, k, s_j, r_k\rangle_{\sigma_k} \in proof$ and $r \neq r_k$ then
    134:     for $m \in proof \setminus replies$ do
    135:         $store\_reply(m, \text{sender}(m))$
```

Figure 5.19: Protocol D-APE for Reconfigurable Parsimonious Execution (Part VI)

Chapter 6

Implementation Details

6.1 Introduction

We have developed a software toolkit that implements protocols for efficient and dynamic replication of a stateful service. We call this toolkit, the *Component-Based Framework for Intrusion Tolerance*, or *CoBFIT*. We use the term "component" here in the classic software engineering sense of the word, i.e., a component is a coherent, encapsulated part of the toolkit that provides its service operations in the form of clearly defined event/object interfaces that other components can use. The class of services that can benefit from our CoBFIT toolkit are those that can be implemented as deterministic state machines. The CoBFIT toolkit can be used to make such a service fault-tolerant using the state machine replication approach [Sch90], i.e., by replicating the service on multiple nodes of a distributed system and then coordinating client interactions with the replicas. In addition to the protocol components, the CoBFIT toolkit implements many "framework" components that provide a common foundation for implementing not only the specific protocols described in this chapter, but similar distributed fault-tolerant protocols as well.

6.1.1 Overview of Various Components

We now provide an overview of the various protocol and framework components in the CoBFIT toolkit. Figure 6.1 shows the instantiation hierarchy of components in the CoBFIT toolkit. The top-level component is the `Main` component; the term "replica start" is synonymous with the creation of the `Main` component for that replica. It is the `Main` component

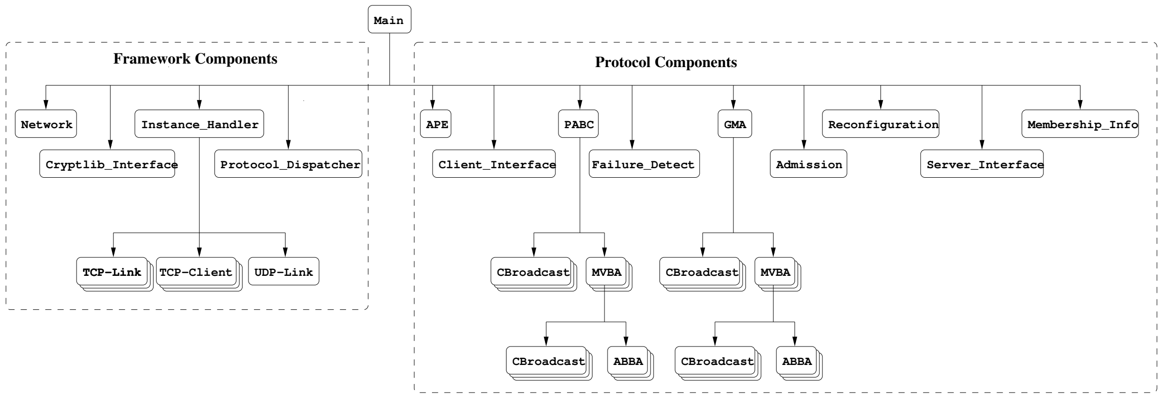


Figure 6.1: Component Instantiation Hierarchy

that creates the framework components and the highest-level protocol components.

There are four framework components:

1. The `Network` component is responsible for all of a replica's communication with the outside world (i.e., including other replicas and clients).

2. The `Cryptlib_Interface` component provides interfaces for performing cryptographic operations required by the protocol components.

3. The `Instance_Handler` is the component that instantiates all protocol components; the `Main` component asks the `Instance_Handler` to instantiate the highest-level protocol components, which may in turn ask the `Instance_Handler` to instantiate lower-level protocol components.

4. The `Protocol_Dispatcher` de-multiplexes and dispatches incoming messages received from the peer protocol components at other replicas to the appropriate protocol component.

The highest-level protocols in the CoBFIT toolkit include:

1. A message-efficient atomic broadcast protocol implemented in the `PABC` component that represents the agreement phase of state machine replication,

2. A resource-efficient execution protocol implemented in the `APE` component that represents the execution phase of state machine replication, and

3. A set of protocols implemented in the `Reconfiguration`, `Failure_Detect`, `Admission`, and `GMA` components that utilize the group membership information stored in the `Membership_Info` component, provide a group abstraction called `CoBFIT-group` for the set of replicas, and make it possible to change the group membership dynamically while maintaining replica consistency.

The atomic broadcast protocol totally orders incoming requests from clients of the replicated service, and the execution protocol invokes requests at the service in the total order. The `PABC` component does not directly interact with the clients; instead, a `Client_Interface` component communicates with the clients, filters out unauthorized requests, forwards only the authorized requests to the `PABC` component, and forwards reply certificates back to the corresponding clients. Similarly, the `APE` component does not interact with the service directly, but only through a `Server_Interface` component, which serves as an adapter and hides service-specific details from the `APE` component.

The `CBroadcast`, `ABBA`, and `MVBA` components implement protocols for consistent broadcast, binary Byzantine agreement, and multi-valued Byzantine agreement respectively. Both the atomic broadcast protocol and the group membership agreement protocol (implemented in the `GMA` component) use consistent broadcast and multi-valued Byzantine agreement as primitives; hence, both the `PABC` and `GMA` components instantiate the `CBroadcast` and `MVBA` components during their operation. The multi-valued Byzantine agreement protocol implemented in the `MVBA` component uses consistent broadcast and binary Byzantine agreement as primitives; hence, the `MVBA` component instantiates the `CBroadcast` and `ABBA` components during its operation.

6.1.2 The ACE Toolkit

The CoBFIT toolkit was developed in C++ using an object-oriented network programming toolkit called the Adaptive Communication Environment or ACE [SH02][SH03]. The ACE toolkit is open-source and widely used, and has proved to be a robust platform for building high-performance communication services. ACE provides a rich set of reusable C++ wrapper facades and framework components that perform common communication software tasks across a range of operating systems and platforms. In the following sections, classes prefixed with `ACE` (such as the `ACE_Message_Block`, `ACE_Task`, and `ACE_Input_CDR` classes) are all provided by the ACE toolkit. Where necessary, we provide a brief overview of the ACE classes used; however, for detailed documentation of the ACE classes used by the CoBFIT toolkit, we refer the reader to two books by Schmidt and Huston, *C++ Network Programming: Systematic Reuse with ACE & Frameworks* [SH02] and *C++ Network Programming: Mastering Complexity with ACE & Patterns* [SH03].

6.1.3 Organization of the Chapter

The rest of this chapter is organized as follows. Section 6.2 describes in detail the framework components that form the foundation upon which protocol components are built. Sections 6.3 and 6.4 describe the implementation of the consistent broadcast protocol and agreement protocols (respectively) that serve as primitives for implementing higher-level protocols such as the atomic broadcast protocol and the group membership agreement protocol. The `PABC` and `APE` components that implement protocols for the agreement and execution phases (respectively) of state machine replication are described in Section 6.5. Section 6.6 describes the components that implement protocols for the formation and dynamic membership management of the `CoBFIT-group`. Finally, Section 6.7 describes the `Client_Interface` component that interfaces with the clients of the replicated service and the `Server_Interface` component that interfaces with the replicated service.

6.2 Framework Components

The CoBFIT toolkit implements a number of distributed protocols that collectively accomplish efficient, Byzantine-fault-tolerant state machine replication. Although each protocol provides a distinct set of properties and serves a distinct purpose, the fact is that a number of implementation requirements are common to many of the protocols. Here, we provide a few examples of such requirements. Several protocols require communication among peer protocol instances running at the replicas. Protocols such as `ABBA`, `MVBA`, and `CBroadcast` are all instantiated by some higher-level protocol that expects them to provide an output before termination. All our protocols are designed as reactive protocols, i.e., they respond to the occurrence of an event (e.g., timeout), receipt of a message, or an input action. Since the fault model is Byzantine, no single party can be trusted; hence, all protocols make their progress from one stage to the next contingent upon the participation of enough correct parties. In many cases, the fact that enough correct parties participated in a particular stage has to be recorded in a manner that is provable to a third party (e.g., using cryptographic schemes), so that the third party can be convinced to advance to the next stage. The purpose of the CoBFIT framework components is to provide the requirements that are common to several protocols (such as the above) in a reusable manner that can result in significant savings in the implementation effort. In this section, we describe each of the four framework components in detail.

6.2.1 Network Communication

The `Network` component is responsible for all of a replica's communication with the outside world (i.e., including other replicas and clients). At a replica that is a member of `CoBFIT-group`, the component maintains multiple `TCP-Link` components. Each `TCP-Link` is used for the replica's exclusive communication with another group member and implements a secure authenticated channel abstraction using MAC-based authentication. The set of `TCP-Link` components in the `Network` is established as one of the steps during the replica's

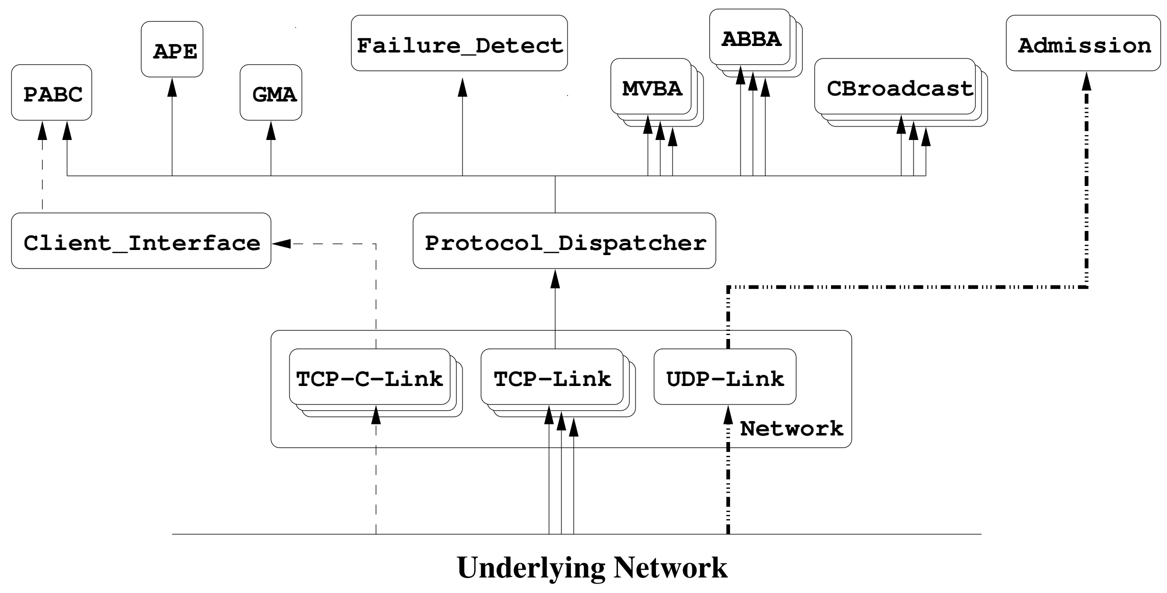


Figure 6.2: Message Inflow at a Party P_i

joining of the `CoBFIT-group`, and changes as the group membership changes. At a replica, a `TCP-C-Link` is established when an authorized client connects to a special port called the *client acceptor port*. The `Network` component spawns a separate `TCP-C-Link` component for exclusive communication with the client. A `TCP-C-Link` may span several requests and is garbage-collected when the client has closed the connection or after a period of inactivity (i.e., no valid client requests). Each `TCP-C-Link` is used for the replica's exclusive communication with a client and implements a secure authenticated channel abstraction using digital signatures. The `Network` component also instantiates a `UDP-Link` component for communication between a group member and a replica that is not yet a member but wants to be one.

Figure 6.2 depicts how incoming messages at a replica are dispatched from the `Network` component to the appropriate higher-level components. The `Protocol_Dispatcher` component *subscribes* to all messages that are received at the `Network` component via `TCP-Links`. All protocol instances at a group member that communicate with peer protocol instances at another group member register with the `Protocol_Dispatcher`. Any message received via a `TCP-Link` is examined by the `Protocol_Dispatcher` component and dispatched to the ap-

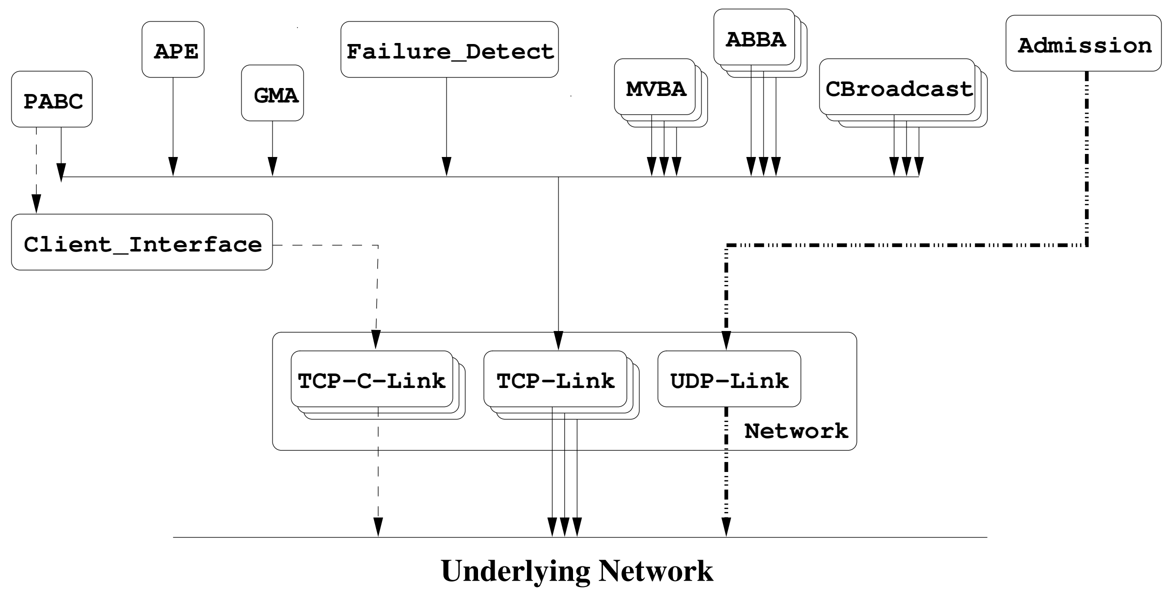


Figure 6.3: Message Outflow at a Party P_i

propriate protocol instance (we describe the dispatching in detail in Section 6.2.2). Requests received from clients via the corresponding TCP-C-Link components are directly dispatched to the Client_Interface component, which in turn forwards the requests that are authorized to the PABC component. Messages from replicas that are not yet group members are received via the UDP-Link and directly dispatched to the Admission component.

Figure 6.3 shows how messages from various components are sent over the network. All protocol components at a group member send messages to protocol components at another group member through the corresponding TCP-Link. The PABC component passes the reply certificate for a client request to the Client_Interface, which then forwards the certificate through the corresponding TCP-C-Link to the client. The Admission component at a non-member sends messages to the peer component at a group member through the UDP-Link; the component at a member also sends messages to a peer component at a non-member through the UDP-Link.

6.2.2 Management of CoBFIT Protocol Components

The `Protocol_Instance`, the `Protocol_Dispatcher`, and the `Instance_Handler` classes are central to the management of protocol instances in the CoBFIT toolkit. All CoBFIT protocol implementations derive from the `Protocol_Instance` base class. As the name indicates, the `Protocol_Dispatcher` is responsible for dispatching incoming messages from other `CoBFIT-group` members to the appropriate protocol instance. The `Instance_Handler` provides methods for instantiating and destroying protocol instances. In this section, we describe the functionalities provided by the three classes.

Brief Overview of the `ACE_Task` Framework

We defined a `Service_Component` class that inherits from the `ACE_Service_Object` class, which in turn inherits from the `ACE_Task` class [SH03]. The `Protocol_Dispatcher` and the `Protocol_Instance` classes derive from this `Service_Component` class.

The `ACE_Task` class [SH03] provides capabilities for queueing and processing messages and reactive handling of events (such as timeouts). A consequence of inheriting from the `ACE_Task` class is that each protocol instance runs in its own thread of control. `ACE_Task` has an instance of the `ACE_Message_Queue` class; the message queue helps separate data and requests from their processing. The `ACE_Task` class provides a `putq()` method that can be used to enqueue a message onto the `ACE_Message_Queue` and a `getq()` method that block waits until a message is available on the message queue and then dequeues the message.

The `ACE_Task` class provides a method `activate()` that is invoked to convert a task into an *active object*. An active object is an object that implements the Active Object pattern [SSRB01][SH03] and executes service requests in a thread separate from the caller's thread. This method is invoked only once at the task, soon after it is instantiated. The `ACE_Task` `activate()` method in turn calls the `svc()` method, which is overridden by its subclasses to implement the task's service processing.

A consequence of inheriting from the `ACE_Service_Object` class is that all components are not only `ACE_Tasks` but also services that can be dynamically linked/unlinked by the `ACE_Service_Config` framework through a configuration file if need be.

Instantiating and Destroying Protocol Instances using the `Instance_Handler`

There is a singleton `Instance_Handler` component through which all `Protocol_Instance` components are instantiated and destroyed. Each `Protocol_Instance` component is uniquely identified by a string called the *tag*. At replica creation time, the `Main` component creates the `Instance_Handler`, and it is through the latter that all the highest-level protocol components (such as the `PABC` and `APE` components) are created. All protocol instances maintain a reference to the `Instance_Handler` whose services they utilize to create lower-level protocol instances. For example, protocols such as `PABC` and `GMA` create lower-level protocols such as `MVBA` and `CBroadcast` through the `Instance_Handler` component.

The `Instance_Handler` maintains the following data structures:

- A hash table, `creations`, that maps protocol tags to the actual protocol instances. The table contains an entry for each protocol instance instantiated by the `Instance_Handler` that has not yet been garbage-collected.
- A list `destroyed` of protocol tags corresponding to protocol instances that were instantiated and subsequently garbage-collected.
- A hash table, `notify`, that maps protocol tags with sets of components. Upon the creation of a protocol instance whose identifier is one of the tag entries present in the table, the `Instance_Handler` has to notify the corresponding set of components.

The `Instance_Handler` provides three `create_protocol_instance` interfaces for instantiating protocol instances; soon after instantiating a protocol instance, the `Instance_Handler` invokes the `activate()` method at the instance and converts it into an active object. All three interfaces, given below, include a `tag` parameter, which specifies the unique protocol identifier, and a `type` parameter, which specifies the protocol type (i.e., `PABC`, `APE`, `ABBA`, etc.).

```
Protocol_Instance* create_protocol_instance (ACE_CString tag,
                    int type, ACE_Message_Block* start_value, Validation* val);
Protocol_Instance* create_protocol_instance (ACE_CString tag,
                int type, ACE_Message_Block* payload, int sender, bool strong);
Protocol_Instance* create_protocol_instance (ACE_CString tag, int type);
```

The first interface is used in the creation of validated agreement protocols such as the `ABBA` and `MVBA` protocols. The `start_value` parameter is used to provide the input value for the agreement protocol, and the `val` parameter is a `Validation` object whose `validate_input` function implements the validation predicate to determine whether a specified input value is valid from the perspective of the higher-level protocol that instantiated the agreement protocol. The second interface is used in the creation of the consistent broadcast protocol implemented by the `CBroadcast` component. The `sender` parameter is used to specify the rank of the designated sender replica. The `payload` parameter, as the name indicates, is the payload to be broadcast; the parameter is relevant only at the sender. The Boolean `strong` parameter is used to specify whether the instantiated `CBroadcast` protocol instance needs to satisfy *strong* or *ordinary* protocol semantics (we detail this later in Section 6.3). The third interface is a generic interface used to create all other protocol types (`PABC`, `APE`, etc.).

The `Instance_Handler` provides the `get_protocol_instance` interface to obtain a reference to a previously created protocol instance.

```
Protocol_Instance* get_protocol_instance (ACE_CString tag, int &errno);
```

If there is an entry for the protocol instance with identifier `tag` in the hash table `creations`, then the reference to the protocol instance will be returned; otherwise, the return value will be NULL. There are two cases in which the return value would be NULL: (1) if the protocol instance with identifier `tag` has never been created before, and (2) if the protocol instance was created and then subsequently destroyed using the `destroy_protocol_instance` interface provided by the `Instance_Handler`. In the latter case, `tag` will be part of the `destroyed` list maintained at the `Instance_Handler`. The `errno` parameter returns the error status of the execution of the `get_protocol_instance` function, and is used to distinguish between the two cases. An `errno` value of 1 indicates the former case, while a value of 2 indicates the latter case. A value of 0 is used when the `Protocol_Instance` returned is non-NULL.

Components can also register with the `Instance_Handler` to be notified later when a particular protocol instance is created. That is done using the `notify_upon_creation` interface.

```
void notify_upon_creation (ACE_CString tag, Service_Component *caller);
```

Here, `tag` is the identifier of the protocol instance, and the `caller` is the component that is interested in receiving the notification when the protocol instance with identifier `tag` is created sometime in the future. The `Instance_Handler` adds a map entry for `tag` and `caller` in its `notify` hash table. When a protocol instance with identifier `tag` is actually created through the invocation of one of the three `create_protocol_instance` methods, the `Instance_Handler` enqueues a special `instance-created` message on the message queue of the interested component, `caller`. This notification mechanism is mainly used by the `Protocol_Dispatcher` as will be explained below.

The `destroy_protocol_instance` is an interface provided by the `Instance_Handler` for shutting down a protocol instance.

```
void destroy_protocol_instance (ACE_CString tag);
```

The method is typically invoked by the component at whose behest the protocol instance with identifier `tag` was created by the `Instance_Handler`. The method looks up the `creations` table for a protocol instance with identifier `tag`; if the instance is present, the method simply enqueues a special `terminate` message in the message queue of that instance and waits until the thread corresponding to the protocol instance exits. At that point, the method adds `tag` to the `destroyed` list and deletes the reference to the protocol instance. When a protocol instance dequeues the `terminate` message, the instance exits its event loop that is executing

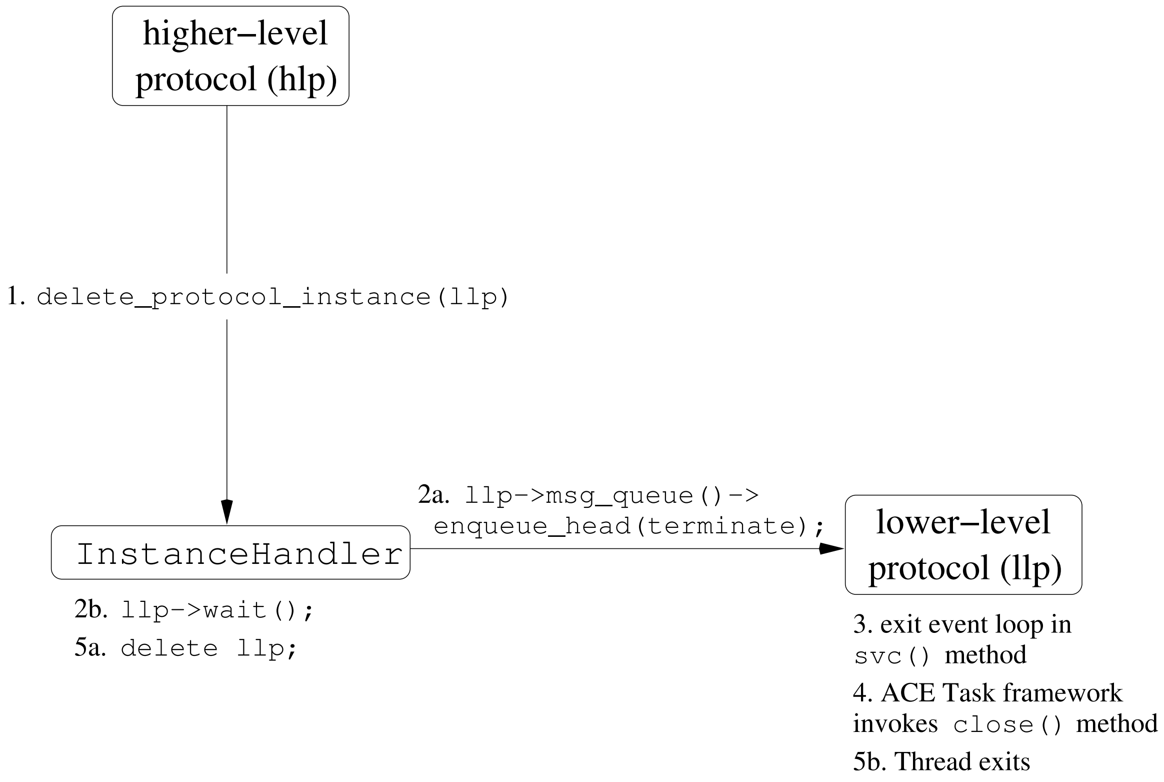

higher–level protocol (hlp)
1. delete_protocol_instance(llp)
InstanceHandler
2a. llp->msg_queue()->enqueue_head(terminate);
lower–level protocol (llp)
2b. llp->wait();
5a. delete llp;
3. exit event loop in svc() method
4. ACE Task framework invokes close() method
5b. Thread exits

Figure 6.4: Garbage Collection of Protocol Instance

in its `svc()` method (see below), which will result in the ACE task framework calling the `close()` method at the protocol instance; the `close()` method performs protocol-defined shutdown activities. When the `close()` method returns, the thread corresponding to the protocol instance exits.

The Operation of Protocol Instances

All subclasses of `Protocol_Instance` must implement three virtual methods declared by the `Protocol_Instance` class, namely:

```
virtual void start();

virtual void add_message (ACE_InputCDR &inputcdr);

virtual void close();
```

Recall that when an `ACE_Task` becomes an active object, the `svc()` method is invoked. At the `Protocol_Instance` class, the `svc()` method invokes the virtual `start()` hook method that subclasses override to perform protocol-specific initialization activities. After the `start()` method returns, the `svc()` method then starts an event loop that uses the `getq()` method of `ACE_Task` to wait for a message to arrive on the protocol instance's message queue. After the `getq()` method returns, the `svc()` method invokes the virtual `add_message()` function implemented by all protocol instances to perform protocol-defined message processing. The event loop exits, and thereby the `svc()` method returns, only when the `getq`() method returns a `shutdown` message. Once the `svc()` method completes, the `ACE_Task` framework will invoke the `close()` method at the protocol instance; the method will implement all protocol-defined shutdown activities.

In addition to the `ACE_Message_Queue` that a `Protocol_Instance` inherits from the `ACE_Task` class, the `Protocol_Instance` class itself defines another `ACE_Message_Queue` object called `result_channel`. A protocol instance uses the `result_channel` to pass results or outputs to the higher-level protocol instance that was responsible for the creation of the former protocol instance. A higher-level protocol instance maintains references to all lower-level protocol instances it creates (through the `Instance_Handler`); it then uses the reference of a lower-level protocol instance to obtain a reference to the corresponding `result_channel` and retrieve results synchronously or asynchronously. To obtain results synchronously, the higher-level protocol instance simply calls the `dequeue_head()` method at the lower-level protocol instance's message queue; the method will block wait until some result is enqueued. Results can also be obtained asynchronously as follows. The lower-level protocol sends a signal through the `ACE_Reactor` framework [SH02][SH03] when the results are enqueued on the `result_channel`. For handling the signal, the higher-level protocol must implement an event handler, which is invoked by the `ACE_Reactor` framework; the handler will dequeue the result and perform protocol-specific processing.

Message Dispatching by the `Protocol_Dispatcher`

The `Protocol_Dispatcher` subscribes to all messages received through the `TCP-Links` at the `Network` component. The `Network` component enqueues any messages received through the `TCP-Links` at the message queue of the `Protocol_Dispatcher`. All messages carry the destination protocol tags in their headers, which enables the `Protocol_Dispatcher` to determine the protocol instance for which a message is destined.

The `Protocol_Dispatcher` invokes the `get_protocol_instance()` interface at the `Instance_Handler` to obtain a reference to the destination protocol instance for a message. After determining the destination protocol instance for a message received via a `TCP-Link`, the `Protocol_Dispatcher` in turn enqueues the message onto the message queue of that protocol instance by invoking the `putq()` method. If the value returned by the `get_protocol_instance()` method indicates that the protocol instance has already been destroyed, then the `Protocol_Dispatcher` simply discards the message. If the return value indicates that the protocol instance has never been created before, the `Protocol_Dispatcher` registers with the `Instance_Handler` to be notified when the instance is created at some time in the future and, thereafter, buffers the messages destined for that protocol instance. When such a protocol instance is actually created, the `Instance_Handler` enqueues a special `instance-created` message (specifying the tag of the protocol instance) on the message queue of the `Protocol_Dispatcher`. Upon receiving the notification from the `Instance_Handler` that the protocol instance has been created, the `Protocol_Dispatcher` obtains a reference to the protocol instance (through the `get_protocol_instance` interface of the `Instance_Handler`) and then enqueues all the buffered messages onto the message queue of that protocol instance.

The implementation of the `svc()` method (inherited from the `ACE_Task` class) in the `Protocol_Dispatcher` component is basically an event loop whereby the method dequeues the message in the component's `ACE_Message_Queue`, determines the protocol instance to which the message has to be dispatched, and then enqueues/buffers the message in the

ACE_Message_Queue of the destination protocol instance.

6.2.3 Implementation of Cryptography Primitives

CoBFIT components use many cryptographic primitives, such as cryptographic hash functions, message authentication codes or MACs, digital signatures, and a pseudo-random number generator (PRNG). Cryptographic hash functions are used to compute message digests. MAC authentication is used to establish a secure authenticated point-to-point channel abstraction between any two members of the CoBFIT-group. Digital signatures are used to authenticate clients as well as for signing any message that needs to be provable to a third party (i.e., any party besides the sender and receiver of that message). A PRNG is used to generate the secret key used to obtain MACs. We used Peter Gutmann's Cryptlib [cry] as the core cryptographic library and wrote wrapper functions around it. All the functions are contained in the CoBFIT Cryptlib_Interface component, which we describe in the rest of this section.

Symmetric keys used for MAC authentication at the TCP-Link level are generated by invoking the following method:

```
int get_random_data (ACE_UINT32 buf_len, void* buf);
```

The method call fills the input data buffer buf with the required number buf_len of random bytes; the call returns 0 on success and -1 on failure. Internally, the method uses the RC4 PRNG algorithm [VvOM96] to generate the random bytes.

The cryptographic hash or the digest of a message is obtained by invoking the following method:

```
int crypt_hash_buf (void* buf, ACE_UINT32 buf_len, void* hash);
```

The method call computes the digest of the message buf of length buf_len using the SHA secure hash algorithm and fills the buffer hash with the digest. The parameter hash must be an allocated buffer of size CRYPT_MAX_HASHSIZE (= 32 bytes). The method call returns the actual length of the digest (which is expected to be 20 bytes).

The `crypt_verify_hash` method is used to check that the hash stored in a buffer corresponds to the contents of a message.

```
int crypt_verify_hash (void* buf, ACE_UINT32 buf_len, void* hash);
```

The method call checks whether `hash` is the correct digest for the message `buf` of length `buf_len`. The method call returns 0 if the digest is correct; otherwise, it returns -1.

For a MAC-authenticated message m to be sent from group member P_i to another member P_j, the sender P_i computes the hash of the string obtained by concatenating m with the symmetric key that P_i and P_j share; P_i then sends both m and the hash. The receiver P_j computes the hash in a similar way and checks whether the computed hash is the same as the hash received from P_i. MAC-authentication thwarts the threat posed by a malicious interceptor who could modify the message and replace the digest with the digest of the modified message, for the interceptor won't have access to the secret key.

The `Cryptlib_Interface` component implements wrapper functions around Cryptlib for key generation, signing, and verification using the RSA algorithm. The public key/private key pairs for all parties in the universal-list and client-list are generated a priori. When a replica is created, as part of the initialization procedure, it is handed over its private key and the public keys of all parties. This information is used to populate the `universal_table` and the `private_key` data structures in the `Membership_Info` object, and the `client_table` data structure in the `All_Clients_Info` object.

The `crypt_sign_buf` method is used to generate a digital signature on the contents of a buffer `buf` of length `buf_len`. The computed signature is stored in the buffer `sign` provided as input. The buffer `sign` must have been allocated before the method call and must have a size `CRYPT_MAX_SIGNSIZE` (= 4096 bytes). For efficiency, the signature is generated around a hash of `buf` rather than `buf` itself. When an optional input parameter, `hash`, of non-zero value is provided, the hash of the buffer is stored in `hash`. The method call returns the actual length of the signature.

```
int crypt_sign_buf (void* buf, ACE_UINT32 buf_len, void* sign, void* hash=0);
```

The `crypt_verify_sign` method is used to verify whether a digital signature is correct.

```
int crypt_verify_sign (void* buf, ACE_UINT32 buf_len, void* sign,
                                           ACE_TCHAR* key_id, bool hash=false);
```

The method obtains the public key of the party identified by `key_id` and checks whether the buffer `sign` contains the correct signature of the party. The method call returns 0 if the signature is correct; otherwise, it returns -1. If the optional parameter `hash` is `false`, then the method checks whether the buffer `sign` contains the correct signature of the party on *the hash* of the buffer `buf` of length `buf_len`. Otherwise, if `hash` is set to `true`, then the method checks whether the buffer `sign` contains the correct signature of the party on simply the buffer `buf` of length `buf_len`.

The `crypt_encrypt_buf` method is used to encrypt a buffer `buf` of length `buf_len` using the public key of the buffer's intended receiver which is identified using `key_id`. The encryption is done in place, so the encrypted data is also stored in `buf`. The method call returns a positive integer indicating the length of the encrypted buffer on success and -1 otherwise.

```
int crypt_encrypt_buf (void* buf, ACE_UINT32 buf_len, ACE_TCHAR* key_id);
```

The `crypt_decrypt_buf` method is used to decrypt an encrypted buffer using the private key.

```
int crypt_decrypt_buf (void* buf, ACE_UINT32 buf_len);
```

The method decrypts the buffer `buf` of length `buf_len` and stores the decrypted information in place. The method call returns a positive integer indicating the length of the decrypted buffer on success and -1 otherwise.

6.3 Broadcast Protocol

The `CBroadcast` component implements *transferable consistent broadcast* [CKPS01]. In consistent broadcast, there is a designated sender party that is supposed to send the payload to be delivered to all parties. The protocol ensures *consistent delivery*, i.e., the payload delivered at any two correct parties is the same. If the sender is faulty, it is possible that one or more correct parties do not deliver anything. In such cases, *transferability* is a useful mechanism that allows a correct party that did deliver the payload to send information called the *completing string* to convince other parties to also deliver the payload.

Only after the designated sender has gathered signed approvals or *echoes* for its payload from a large enough quorum can the payload be consistently delivered at a correct party. The completing string essentially contains the quorum of signatures. Although a quorum size of $\lceil \frac{n+t+1}{2} \rceil$ is sufficient to ensure the standard properties of consistent broadcast, a version of the consistent broadcast protocol called *strong consistent broadcast* that has a quorum size of $n - t$ is useful in some situations, for example, in Protocol PABC. Consequently, we developed the `CBroadcast` component in a manner that makes it possible to specify in the constructor (given below) whether an instantiated consistent broadcast protocol instance needs to satisfy strong or standard quorum size.

```
CBroadcast (ACE_CString tag, int type, ACE_Message_Block* payload,
                                              int sender, bool strong);
```

Just as at any subclass of `Protocol_Instance`, the `start()` method implements protocol-specific initialization tasks. At a party that is not the designated sender, the method does nothing; only at the designated sender, the method marshals and sends a `c-send` message containing the payload. After the `start()` method returns, the `svc()` method enters the event loop that invokes the `add_message()` method for each message dequeued from the protocol instance's `ACE_Message_Queue`. The `add_message()` method demarshals and processes messages as specified in the protocol [CKPS01].

The `CBroadcast` protocol instance at a party is ready to output or consistently deliver a payload when the designated sender's `c-final` message that includes a quorum of signed echoes for that payload has been received. At that point, the `CBroadcast` protocol instance outputs an `ACE_Message_Block` containing the payload to be delivered and the `c-final` message that serves as the completing string needed to ensure transferability. After enqueueing that `ACE_Message_Block` on the `result_channel`, the `CBroadcast` protocol instance is ready to be garbage-collected. Garbage collection is triggered when the higher-level protocol instance (`PABC`, `GMA`, or `MVBA`) that created the `CBroadcast` protocol instance invokes the `destroy_protocol_instance()` method at the `Instance_Handler` passing as an argument to the identifier of the `CBroadcast` protocol instance.

6.4 Agreement Protocols

There are two validated agreement protocols currently implemented in the CoBFIT toolkit. They are the binary agreement protocol implemented in the `ABBA` component, and the multi-valued agreement protocol implemented in the `MVBA` component. The former protocol allows a group of parties to decide on a binary decision value despite the malicious corruption of up to one-third of the members of the group. The latter protocol allows a group of parties to decide on an arbitrary bit string that is acceptable to the higher-level protocol or application that created the `MVBA` component despite the malicious corruption of up to one-third of the members of the group.

There is an abstract `Agreement` class from which both the `MVBA` and `ABBA` classes derive. The class provides the following constructor:

```
Agreement (ACE_CString tag, ACE_Message_Block* start_value, Validation* val);
```

The `start_value` denotes the input value that the agreement protocol instance at this replica proposes to be considered as a candidate for the decision value. The `val` parameter, an instance of the `Validation` class, provides a way to verify whether the proposals put forth by other parties are valid from the point of view of the higher-level protocol or application that instantiated this agreement protocol instance. The decision value agreed upon by the parties may be the proposal of any party (even a corrupted one) as long as the proposed value is valid.

The `Validation` class is an abstract class that declares the following pure virtual method:

```
virtual bool validate (ACE_Message_Block* input)=0;
```

Corresponding to every higher-level protocol at whose behest an agreement protocol instance is created, a subclass of the `Validation` class that implements the `validate()` method has to be defined. The method should take an `ACE_Message_Block*` object that represents some party's input to the agreement protocol instance as a parameter, and implement the logic for checking whether the input is valid from the point of view of the higher-level protocol.

6.4.1 The `ABBA` Component

Before terminating, the `ABBA` protocol instance outputs either a 0 or 1 on its `result_channel`. The protocol ensures that the output binary value is the same at all correct parties. The `ABBA` class implements Cachin et al.'s asynchronous binary agreement protocol described in [CKS05], with two major differences (1) The implementation replaces threshold signatures with an equivalent array of digital signatures, called *multi-signatures*; thus, where threshold signatures shares from k distinct parties are needed, the implementation instead uses digital signatures from k distinct parties. (2) The threshold coin-tossing primitive has been replaced with a pseudo-random function implemented using the AES block cipher; implicit in this substitution is the assumption that the adversary that corrupts and gains control of parties is distinct from and does not collude with the adversary that controls message scheduling on the network.

Just as at any subclass of `Protocol_Instance`, the `start()` method implements protocol-specific initialization tasks. In the `ABBA` class, the method marshals and sends a `pre-process` message with the input value (0 or 1). After the `start()` method returns, the `svc()` method enters the event loop that invokes the `add_message()` method for each message dequeued from the protocol instance's `ACE_Message_Queue`. The `add_message()` method demarshals messages as specified in the protocol [CKS05]. Just before enqueuing the binary decision value b on the `result_channel`, the `ABBA` protocol instance sends a special `decided` message to the peer protocol instances at other parties that will convince the recipients also to decide on value b. At that point, the `ABBA` protocol instance is ready to be garbage collected; garbage collection is triggered when the higher-level protocol instance that created the `ABBA` protocol instance invokes the `destroy_protocol_instance()` and passes the identifier of the `ABBA` protocol instance as an argument.

At present, the only protocol among the CoBFIT protocols that uses the binary agreement as a primitive is the `MVBA` protocol, which requires that the `ABBA` protocol's decision value be biased towards 1. To "bias the `ABBA` decision value towards 1" is to guarantee that if $t + 1$ correct parties propose 1 along with appropriate validation information, then the `ABBA` protocol must decide 1. The `MVBA` protocol involves a round in which all parties convey their proposed input values to other parties through consistent broadcast. Then, for each "candidate" party in succession, an `ABBA` protocol instance is created to decide whether the candidate party's proposal was indeed consistently delivered by enough correct parties. When some `ABBA` protocol instance decides 1, the `MVBA` protocol decides on the value proposed by the corresponding candidate party and then terminates. For a given candidate party P_i, the `MVBA` protocol at a correct party P_j will instantiate the `ABBA` protocol with a start value of 1 only if the P_j had consistently delivered P_i's proposed input value. To prove that P_j did indeed consistently deliver P_i's proposal and hence that its `ABBA` input value of 1 for candidate P_i is indeed valid, the `MVBA` protocol at P_j must provide validation information through the `start_value` argument of the following `ABBA` constructor.

```
ABBA (ACE_CString tag, ACE_Message_Block* start_value, Validation* val);
```

There is an `ABBA_MVBA_Validation` class derived from the abstract `Validation` class; the `validate()` method in the `ABBA_MVBA_Validation` class checks whether a given input `ACE_Message_Block*` object contains proof of consistent delivery of the candidate's proposal. A pointer to an object of the `ABBA_MVBA_Validation` class is passed as the argument `val` to the `ABBA` constructor.

6.4.2 The `MVBA` Component

Before terminating, the `MVBA` protocol instance outputs a bit string representing the decision on its `result_channel`. The protocol ensures that the output bit string is the same at all correct parties. The MVBA class implements Cachin et al.'s multi-valued validated Byzantine agreement protocol described in [CKPS01]. The MVBA class has the following constructor:

```
MVBA (ACE_CString tag, ACE_Message_Block* start_value, Validation* val);
```

The value to be proposed by a party is passed to the MVBA protocol instance through the start_value parameter; that parameter also includes the proof that the proposed value would be acceptable as output to the higher-level protocol that instantiated the MVBA protocol instance. The val parameter provides a way to validate the proposals of various parties. Currently, the MVBA protocol may be instantiated either by the PABC protocol or the GMA protocol. As described in Chapter 3, Protocol PABC proceeds in *epochs*, and in each epoch, the protocol creates two MVBA protocol instances, one to agree on the watermark sequence number and the other to agree on the set of payloads to deliver before switching to the next epoch. For each of the above three uses of the MVBA protocol, a subclass of the Validation class is defined to help validate the values proposed by various parties.

After enqueuing the decision value on the result_channel, an MVBA protocol instance is ready to be garbage-collected. Just as in the case of the ABBA protocol instances, garbage collection is triggered when the higher-level protocol instance (PABC or GMA) that created the MVBA protocol instance invokes the destroy_protocol_instance() and passes the identifier of the MVBA protocol instance as an argument.

The start() method implementation at the MVBA class creates n CBroadcast instances, where n is the size of the CoBFIT-group. If the rank of a party is k, then the party plays the role of the designated sender for the k^{th} CBroadcast instance at which the party consistently broadcasts its proposed input value (for MVBA) along with validation information; the input value and validation information are obtained from the start_value parameter passed as an argument to the MVBA constructor. After the consistent delivery of payloads (i.e., MVBA proposals) that satisfy the specified validity condition at $n - t$ distinct CBroadcast protocol instances, the start() method marshals and sends a v-vote message [CKPS01] for the first candidate; the vote is either 0 or 1 depending on whether or not the party received the candidate's consistently broadcast proposal satisfying the validity condition. A 1 vote is accompanied by proof of consistent delivery of the candidate's proposal that will convince any third-party to also consistently deliver the candidate's proposal. After sending the `v-vote` message, the `start()` method returns and the `svc()` method that invoked the `start()` method now enters that event loop waiting for messages to arrive on the `MVBA` protocol instance's message queue.

In the event loop of the `svc()` method, the `MVBA` protocol demarshals and processes `v-vote` messages. For a given candidate party, once `v-vote` messages with proper justifications have been obtained from $n - t$ distinct parties, an `ABBA` protocol instance biased towards 1 is started to decide whether the candidate's proposal was consistently delivered at enough correct parties. The input value for the `ABBA` protocol instance is 1, if a single valid `v-vote` for the candidate was received with value 1. As mentioned before in Section 6.4.1, the validity of ABBA input values is checked using the `ABBA_MVBA_Validation` object; the `validate` method of the object checks for proof that the candidate's proposal was consistently delivered if the input value is 1, while no justification is needed for an input value of 0. If the `ABBA` protocol instance corresponding to a given candidate party decides 1, the `MVBA` protocol decides on the value proposed by the candidate party and then terminates; otherwise, the `v-vote` messages for the next candidate party are exchanged, followed by a binary agreement to decide whether the new candidate's MVBA proposal was consistently delivered at enough correct parties, and so on.

6.5 State Machine Replication Protocols

The `PABC` and `APE` components respectively implement the agreement and execution phases of state machine replication. The message-efficient atomic broadcast protocol described in Chapter 3 is implemented in the `PABC` component. The resource-efficient execution protocol described in Chapter 4 is implemented in the `APE` component.

Figure 6.5 shows the components and steps involved in client request processing. At a

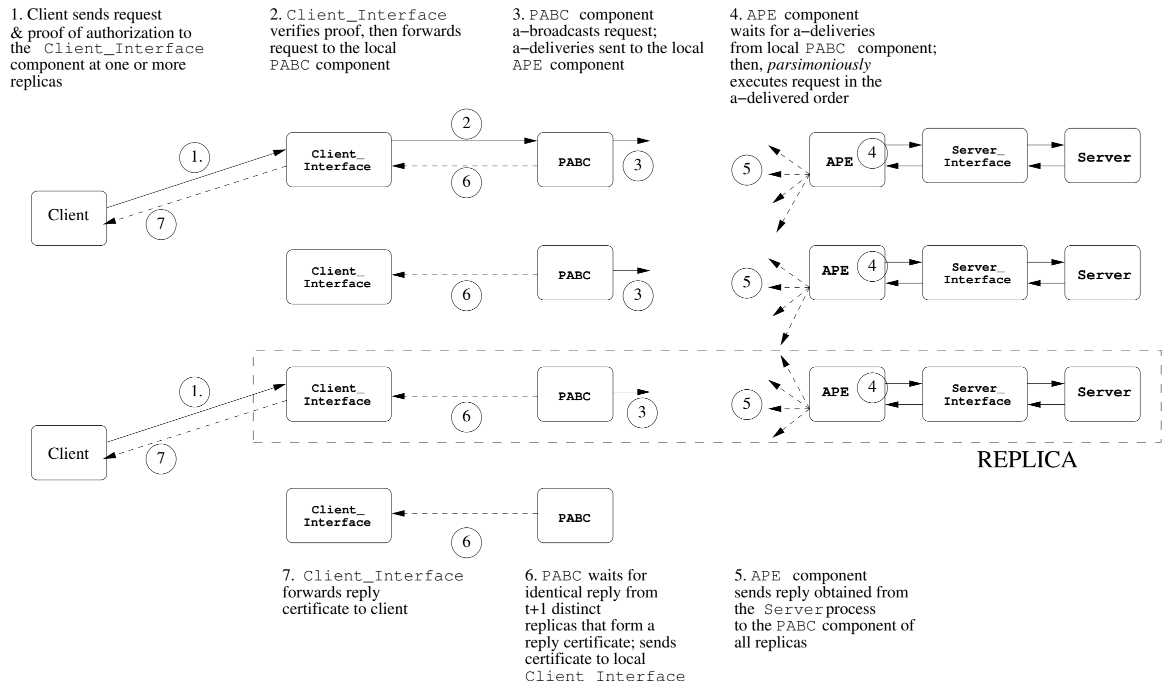

1. Client sends request & proof of authorization to the Client_Interface component at one or more replicas
2. Client_Interface verifies proof, then forwards request to the local PABC component
3. PABC component a–broadcasts request; a–deliveries sent to the local APE component
4. APE component waits for a–deliveries from local PABC component; then, parsimoniously executes request in the a–delivered order
Client
Client_ Interface
PABC
APE
Server_ Interface
Server
REPLICA
7. Client_Interface forwards reply certificate to client
6. PABC waits for identical reply from t+1 distinct replicas that form a reply certificate; sends certificate to local Client Interface
5. APE component sends reply obtained from the Server process to the PABC component of all replicas

Figure 6.5: Steps Involved in Client Request Processing

replica, authorized client requests from the local `Client_Interface` component serve as the input to the `PABC` component. The `a-delivery` messages that bind a sequence number to a client request form the output of the `PABC` component; this output is sent to the local `APE` component. At the `APE` component, an `a-delivery` message received from the local `PABC` component binding a unique sequence number to a particular client request forms an agreement certificate. Agreement certificates define a total order on the sequence of client requests, and the `APE` component schedules the execution of requests (via the `Server_Interface`) at the `Server` process in that order. The `Server` process represents the service that is state-machine-replicated for fault-tolerance whose state has to be kept consistent at all the execution replicas. After obtaining the result of the executed request from the `Server` process (via the `Server_Interface`), the `APE` component forwards the result in the form of a `reply` message to the `PABC` components at all the replicas. For a particular `a-delivered` client request, `reply` messages with identical result values from $t + 1$ distinct execution replicas constitute a reply certificate. After obtaining a reply certificate, the `PABC` component passes the certificate to the local `Client_Interface`, which can then forward it to the appropriate client.

Unlike the `MVBA` and `ABBA`, which output a single decision value and then terminate, the `PABC` and `APE` components, once instantiated, can handle an unlimited number of client requests. We now describe the implementation of the `PABC` and `APE` protocol components; we also detail their interaction with each other and other components in the CoBFIT toolkit.

6.5.1 The `PABC` Component

The `PABC` class derives from the `Protocol_Instance` class and has the following constructor:

```
PABC (ACE_CString tag);
```

It is the `Instance_Handler` component that creates the `PABC` component at the behest of the top-level `Main` component.

Just as at any subclass of `Protocol_Instance`, the `start()` method implements protocol-specific initialization tasks. The method invokes a private `init_epoch()` method that initializes the protocol variables; the `init_epoch()` method in pseudocode form is given in Figure 5.9. After the `start()` method returns, the `svc()` method enters the event loop that invokes the `add_message()` method for each message dequeued from the protocol instance's `ACE_Message_Queue`. The `add_message()` method demarshals and processes messages as specified in Protocol D-PABC. The `Client_Interface` passes authorized client requests to the `PABC` component through the latter's `a_broadcast` interface. These requests form the payloads that need to be atomically delivered to the `APE` component at the execution replicas.

So as to not overwhelm the `APE` component with `a-delivery` messages, the `PABC` component implements the following "acknowledgment" mechanism. After sending out each `a-delivery` message to the `APE` components at the execution replicas, the `PABC` component waits until it obtains a reply certificate correponding to that `a-delivery` before sending out the next `a-delivery` message to the `APE` components. Such an acknowledgment mechanism ensures that, at a `PABC` component, the number of `a-delivery` messages sent is equal to either the number of received reply certificates or the number of received reply certificates plus one. After obtaining a reply certificate, the `PABC` component invokes the `from_CoBFIT()` method (described in Section 6.7) to pass the certificate to the local `Client_Interface` component.

We have also implemented mechanisms to ensure that the set of payloads `a-delivered` by the `PABC` component at all correct replicas in any particular membership of the `CoBFIT-group` is the same. We describe these mechanisms later in Section 6.6 when discussing group reconfiguration.

6.5.2 The `APE` Component

The `APE` class derives from the `Protocol_Instance` class and has the following constructor:

```
APE (ACE_CString tag);
```

It is the `Instance_Handler` component that creates the `APE` component at the behest of the top-level `Main` component.

The `start()` method initializes the protocol variables according to the initialization pseudocode given in Figure 5.14. After the `start()` method returns, the `svc()` method enters the event loop that invokes the `add_message()` method for each message dequeued from the protocol instance's `ACE_Message_Queue`. The `add_message()` method demarshals and processes messages as specified in Protocol D-APE. After obtaining an agreement certificate that binds the next expected sequence number to a client request, the `APE` component passes the client request to the `Server_Interface` component by making a blocking call to the `from_CoBFIT()` method (described in Section 6.7) at the latter component. The method returns the result of executing the client request; the `APE` method encapsulates the result in a `reply` message that is sent to the `PABC` component at all replicas.

We have also implemented mechanisms to ensure that the state of the `Server` process at all correct execution replicas at the end of any particular membership of the `CoBFIT-group` is the same. We describe these mechanisms later in Section 6.6 when discussing group reconfiguration.

6.6 Group Membership and Reconfiguration

The protocols in the CoBFIT toolkit are concerned with a set of replicas that constitute a group called the `CoBFIT-group`. In this section, we describe the formation and dynamic membership management of the `CoBFIT-group`.

6.6.1 The `Membership_Info` object

As the name indicates, the `Membership_Info` object stores all information regarding the membership (*view*) of `CoBFIT-group`. The object maintains the view number x, which is initialized to 0. At a `CoBFIT-group` member, x indicates the number of times the membership of `CoBFIT-group` has changed since the formation of the initial group.

There is a list of identifiers of all replicas that are ever allowed to join `CoBFIT-group`, called the *universal-list*. A replica's identifier consists of its IP address and its acceptor port. We call the specified port the *acceptor port* since it is through this port that replicas accept connections from other intended members.

The `Membership_Info` object maintains a hash-table `universal_table` that maps the identifier of each replica P_i in the universal-list to a `Replica_Info` object that contains the following information:

- P_i's public key,
- a `convict` flag (initialized to `false`) indicating whether P_i was previously removed from `CoBFIT-group` for exhibiting a *provable Byzantine fault* (explained later in Section 6.6.3).
- `removal_time` (initialized to 0) indicating the local clock time when P_i was last removed from the group.

- `join_time` (initialized to 0) indicating the local clock time when P_i most recently became a member of the group.

- `rejuv_time` (initialized to a fixed value R_t); the rejuvenation time for P_i indicates the minimum time that P_i has to spend as a non-member before being considered for readmission into `CoBFIT-group` (explained later in Section 6.6.3).

- `max_rtt`, which indicates the expected maximum round-trip time between P_i and this replica. In a real-world setting, this information would be based on round-trip message delays measured under stable network conditions.

- the symmetric key, `sym_key`, used for MAC-authenticated point-to-point communication with P_i through the `TCP-Link` component.

- an integer, `role`, that indicates whether P_i participates only in the agreement phase or in both phases.

The `Membership_Info` object also contains an array `view` of identifiers of members of `CoBFIT-group`. The array `view` is always a subset of the universal-list. The position at which a replica's identifier appears in the array indicates that replica's *rank* in the group. For notational simplicity, let us use V_x to denote the contents of the `view` array after the x^{th} update. V_x is a totally ordered list, where the $\leq$ relation defines a total order on the elements of V_x based on their rank. Each element of V_x has a distinct rank that ranges from 1 to V_x.

The `Membership_Info` object also stores the private key of the replica and the fault resilience t of the `CoBFIT-group`.

6.6.2 Formation of the Initial Group

We use the term *initial group* to denote the first membership of `CoBFIT-group`. We assume that the replicas constituting the initial group do not get corrupted until after the formation of the initial group. The initial group must have a fault resilience of at least 1, i.e., there must be at least four initial members.

At a replica P_i that is intended to be part of the initial group, the `view` array of the `Membership_Info` object is initialized with the identifiers of the initial group members in the order of their intended ranks. Information about the universal-list of replicas and their public keys is provided to all initial group members; the information is used to initialize the `universal_table` of the `Membership_Info` object. P_i is also initialized with its private key, which is stored in the `Membership_Info` object. The fault resilience variable t in the `Membership_Info` object is also appropriately initialized.

Formation of the initial group involves establishment of a `TCP-Link` between each pair of the initial group members that use the *acceptor-connector* paradigm. The rank determines which replica plays the role of acceptor and which replica plays the role of connector based on the following rule: *If P_i and P_j are two intended members of the initial group, where P_i has a lower rank than P_j, then P_i is the acceptor and P_j is the connector.* For example, if the initial group consists of four members P_1, P_2, P_3, and P_4 listed in order of increasing rank, then replica P_3 will play the role of connector for establishing the `TCP-Link` components with P_1 and P_2, and the role of acceptor for the `TCP-Link` with P_4. There is a `TCP-Link` that each replica P_i establishes for sending messages to itself; in establishing this particular `TCP-Link`, P_i simultaneously plays the role of both acceptor and connector.

Establishment of a two-way `TCP-Link` between an acceptor P_i and connector P_j proceeds as follows. The `Network` component at connector P_j instantiates a `TCP-Link` component for exclusive communication with P_i. The `TCP-Link` at P_j then tries to establish a socket connection with the acceptor port of P_i. P_j keeps trying until it finally succeeds. Acceptor P_i just waits for a socket connection at its acceptor port. When a connection is made, P_i accepts the connection, spawns a new socket, and instantiates its own `TCP-Link` with the new socket information.

Messages exchanged between P_i and P_j through the `TCP-Link` are MAC-authenticated.

For this purpose, each pair of replicas shares a symmetric key. The symmetric keys are generated using the RC4 pseudo-random number generator (PRNG). The MAC of a message m is obtained by applying a collision-free hash function (SHA-1) on the string obtained by concatenating m and the symmetric key.

After the acceptor and connector have established a `TCP-Link` for exclusive communication with the other replica, they perform the following handshake procedure to verify the authenticity of the MAC keys.

1. The `TCP-Link` at the connector generates a key using the RC4 pseudo-random number generator, encrypts the key with the public key of the acceptor, and sends the encrypted key to the acceptor.

2. The `TCP-Link` at the acceptor decrypts the received key using its private key, and sends a random challenge.

3. The `TCP-Link` at the connector waits to receive the challenge and in turn sends its own challenge.

4. The connector P_j then uses the symmetric key shared between P_i and P_j to compute the MAC of the string obtained by the concatenation of P_j's rank, P_j's identifier, P_i's identifier, P_j's challenge, and P_i's challenge. The connector then sends the MAC to the acceptor P_i.

5. Acceptor P_i then verifies the received MAC, and then computes a MAC of a similar string obtained by the concatenation of P_i's rank, P_i's identifier, P_j's identifier, P_i's challenge, and P_j's challenge. P_i then sends the MAC to the connector P_j.

6. The connector verifies the received MAC. At this time, the authenticity of the channel is established.

Once a replica has established one `TCP-Link` with every member of the initial group (including itself), the `Network` component of that replica is ready to provide communication service for a higher-level protocol (such as atomic broadcast) with the peer protocols at other group members. At that time, the `Network` component invokes the `network_ready()` method at the `Reconfiguration` component The `Reconfiguration` component in turn invokes the `done_reconfigure()` method at the `PABC` component. After that point, the `PABC` component can begin its regular operation.

6.6.3 Dynamic Changes to the Group Membership

After the initial `CoBFIT-group` has formed, non-members that are specified in the universal-list can be dynamically added to the group and members can be dynamically removed from the group. Figure 5.1 shows at a high level the triggers and steps for dynamically changing the membership of the `CoBFIT-group` while maintaining consistency of service state and membership information at all correct group members. The trigger for group reconfiguration can come either from the `Admission` component when it recommends that a non-member be added to the group or from the `Failure_Detect` component when it determines that a member should be removed from the group. This section describes the implementation of each of those triggers and steps in detail.

Adding a Non-Member to the `Admission` Component's `recommend` list

Consider a replica P_k that wants to become a member of `CoBFIT-group`. When P_k is started, it is in its own singleton group. We now describe the steps by which P_k becomes a member of the `CoBFIT-group`.

At P_k, the `view` array of the `Membership_Info` object is initialized with the identifiers of the replicas constituting the current membership of the `CoBFIT-group` in the order of their ranks. The fault resilience variable t in the `Membership_Info` object is also appropriately initialized. Information about the universal-list of replicas and their public keys is also provided to P_k; the information is used to initialize the `universal_table` of the `Membership_Info` object. P_k is also initialized with its private key, which is stored in the

Membership_Info object. All the above initialization information is provided to P_k at the time of its instantiation by the entity that instantiated P_k.

Each replica has an Admission component. At a non-member replica P_k, the component helps P_k to join CoBFIT-group. At a replica P_i that is already a member of CoBFIT-group, the component controls which non-members are allowed to join the group dynamically.

Since the TCP-Link components are established only among CoBFIT-group members, communication between a non-member and current members is done using UDP datagrams. It is for this purpose that the Network component at each replica has a UDP-Link. The UDP-Link components at all replicas are initialized with the same fixed UDP port.

The Admission component at a member P_i maintains a recommend list of non-members that P_i recommends to be added to CoBFIT-group. If non-member P_k is added to the recommend lists of $t + 1$ correct replicas, then we guarantee that P_k will eventually become a member of CoBFIT-group.

We now describe the implementation of two example admission control policies, one for first-time admission into the group and another for parties seeking readmission into the group.

Becoming a First-time Member. Consider a replica P_k in the universal-list that has never been a member of CoBFIT-group. We assume that P_k is not corrupted at the time when it joins CoBFIT-group for the first time. The adversary, however, is free to corrupt P_k anytime after P_k has become a first-time member of CoBFIT-group.

The Admission component at P_k generates a request-join message that contains its identifier (as it appears in the universal-list) and signs the digest of the message using its private key. The signed message is then sent to the UDP-Links of all members of the CoBFIT-group. The Admission component at a recipient group member P_i first checks whether P_k's identifier is in the universal_table of the Membership_Info object. If it is, the component checks whether the removal_time for P_k in the universal_table is 0. If it is, then that would confirm that P_k has never been part of the group. Then, the component verifies the validity of P_k's signature on the `request-join` message by using the corresponding public-key information stored in the `universal_table`. If the signature is valid, then P_k is added to the `recommend` list.

Readmission to the Group. It is possible that a correct replica is removed from the group for being slow due to imperfections in failure detection or transient load/network conditions. To account for that factor, we allow the readmission of replicas that were removed from the group for the sole reason of being slow. Note that if a replica had been removed from the group for exhibiting a provable Byzantine fault, then the `convict` flag for the replica in the `universal_table` would be set to `true` at the time of membership change. Such a replica will never be readmitted into the group.

Readmission has to be done carefully; otherwise, it may be exploited by a corrupt process to create a cycle of removals and readmissions. Since group reconfigurations are expensive and involve blocking the operation of the agreement and execution phases, such a cycle would essentially be a denial-of-service (DoS) in which the group members spend all their time in group reconfiguration without getting any useful work done.

Our readmission policy is based on the heuristic that a replica P_k that has been removed from the group for being slow is likely to be slow if readmitted to the group soon after its removal. Hence, group members consider P_k for readmission only after a sufficient *rejuvenation time* has elapsed since P_k's last removal from the group. The rejuvenation time for each replica in the universal-list is stored in the `rejuv_time` field of the `universal_table`. When P_k is removed from the group, the `removal_time` field for P_k's entry in the `universal_table` is updated, and the rejuvenation time for P_k is recalculated as:

$\texttt{rejuv\_time} \leftarrow \max(T_R, 2 \cdot \texttt{rejuv\_time} - log_2(1 + \texttt{removal\_time} - \texttt{join\_time}))$.

The $2 \cdot \texttt{rejuv\_time}$ factor increases the P_k's rejuvenation time exponentially every time P_k is removed from the group for being slow; the exponential growth ensures that, for sufficiently large T_R, the above DoS is avoided. Since the average group reconfiguration time is expected to be relatively large, T_R may be set to that time value. The $log_2(1 + \texttt{removal\_time} -$

join_time) factor allows P_k to earn a form of "good behavior credit" based on how long it stayed in the group since its last admission. The credit allows a correct replica that was removed from the group (perhaps multiple times) for being slow because of temporary load/network conditions to negate the high rejuvenation time once stable conditions return. However, this credit grows very slowly compared to the exponential growth factor.

For a replica P_k seeking readmission, the addition of P_k to the recommend list at a group member P_i involves the following steps:

1. P_k first sends a signed pre-join message (containing P_k's identifier as it appears in the universal-list) to all the group members.

2. A recipient CoBFIT-group member P_i will respond to P_k's pre-join message only after P_k has fulfilled its rejuvenation time outside the group; P_i checks whether P_k has done so from the rejuv_time and removal_time fields corresponding to P_k's entry in universal_table and from P_i's current local clock time. If it did, then P_i sends P_k a random challenge encrypted with P_k's public key.

3. P_k echoes back the challenge to P_i.

4. If P_i has received the echo from P_k in a timely fashion, then P_i sends a signed approval message containing the current view number (obtained from the variable x of the Membership_Info object) to P_k. Timeliness of the echo is determined from P_i's local clock and the max_rtt field of the Replica_Info object corresponding to P_k maintained at P_i's Membership_Info object.

5. P_k waits until it has collected $t+1$ approval messages with the same view number from distinct group members and then sends a signed request-join message containing those approval messages.

6. After verifying that the request-join message has a set of $t + 1$ valid approval messages from distinct group members with the same view number as itself, a group

member P_i will add P_k to its `recommend` list.

Note that all the above communication between P_k and P_i takes place through `UDP-Links`. Steps 3 and 4 are included to decrease the likelihood that a replica P_k previously removed for being slow because of temporary load/network conditions gets readmitted into the group when those conditions still prevail. Although the steps cannot completely eliminate that likelihood, they are still included, because the cost of executing them is considerably less than the cost of a potential group reconfiguration if the replica were to be removed again after readmission.

Adding a Member to the `Failure_Detect` Component's `remove` list

We provide the capability for removing members dynamically from the group. The fact that removal of corrupt members is not required allows us to be very conservative in using that capability. An example situation in which the capability may be useful from a performance point of view is when a member exhibits a fault that shows without any doubt that the member is corrupted.

Since a group member corrupted by the adversary can behave perfectly normally, a member can be labeled faulty only if it exhibits a *detectable Byzantine fault* [KMMS03][DGGS99], i.e., a fault that can be observed by other members based solely on the messages they receive from that member, or a fault that can be attributed to that member. Detectable Byzantine faults can be classified into muteness and malicious faults. A *muteness* fault occurs when a replica does not send a required message to one or more correct replicas. A *malicious* fault occurs when a replica sends a message that it should not have sent according to the system specifications.

As previously described in Section 5.5, when removing members from the group, it is useful to ensure the following two properties. First, if enough correct members want a member to be removed from the group, then that member must be eventually removed. Second, it must not be possible for corrupted members alone to effect the removal of a member from the group; if it were, then the corrupted parties may act in collusion to cause the disintegration of the group.

The `Failure_Detect` component helps provide the above two guarantees by exercising control over which members will be removed from the group. The component at each `CoBFIT-group` member P_i maintains a `remove` list of group members that it recommends be removed from the group. We ensure the first property by guaranteeing that if a member P_k is added to the `remove` lists of $t + 1$ correct replicas in a given view V_x, then P_k will be removed from `CoBFIT-group` in the next view. We ensure the second property by imposing the restriction that entries in the `remove` list must be *transferable*. The `remove` list of a member P_i is transferable if and only if for each member P_k in its `remove` list, P_i can present information called *validation data* that will cause any correct member P_j to add P_k to its `remove` list too. The validation data for member P_k in P_i's `remove` list will differ based on the type of fault that P_k exhibited and also on the particular component (`PABC`, `APE`, `MVBA`, `ABBA`, `CBroadcast`, etc.) that helped identify the fault.

Validation data for ensuring the transferability of malicious faults take the form of message(s) signed by the faulty member proving deviation from component specification, e.g., two signed *c-broadcast* messages with different payloads for the same *c-broadcast* protocol instance. Since muteness faults cannot be perfectly detected in an asynchronous system, one can place the restriction that validation data for labeling a member as mute include signatures from at least $2t + 1$ members.

For a given CoBFIT component, there may potentially be many ways to deviate from the component specification. It is up to the component to specify which of those deviations warrant removal from the group. For each deviation that warrants removal, a subclass of the abstract `Validation` class is defined; the subclass implements the `validate` interface declared in the abstract class.

```
bool validate (ACE_Message_Block* validation_data);
```

The interface implementation helps make the deviation exhibited by replica P_k and observed by replica P_i transferable to a third replica P_j in the following manner. If the Failure_Detect component at P_j invokes the validate interface implementation in the subclass with validation data supplied by P_i and the invocation returns true, then P_j will be convinced that P_k indeed exhibited that deviation.

Since there may be multiple CoBFIT protocols specifying deviations that warrant removal from the group, there must be a way for the Failure_Detect component to select the appropriate Validation subclass. To provide that way, the validation data itself indicates the deviation type and protocol type, and the Failure_Detect component maintains a hash table fault_validation_map that maps the protocol type and the deviation type to an instantiation (i.e., object) of the appropriate Validation subclass.

Each CoBFIT component implements hooks for identifying detectable and transferable Byzantine faults exhibited by the peer component at another replica. When a component has validation data that warrants removal of a member P_k from the group, then the component enqueues a fault-detected message containing the P_k's identifier and validation data at the Failure_Detect component's message queue. After dequeuing that message, the Failure_Detect component adds P_k to the remove list.

Group Membership Agreement - GMA component

The GMA component is responsible for maintaining consistent group membership information at all correct members of CoBFIT-group. The component ensures that V_x at any correct member of the CoBFIT-group consists of the exact same set of replicas.

The Admission component enqueues an admit message in the GMA component's message queue when a replica P_k is added to the recommend list. The message contains the identifier of replica P_k as it appears in the universal-list and the justification for adding P_k to the recommend list, namely a valid request-join message signed by P_k. Similarly, the Failure_Detect component enqueues a remove message in the GMA component's message queue when a replica P_k is added to the remove list. The message contains the identifier of replica P_k as it appears in the universal-list and the validation data that justifies P_k's addition to the `remove` list, and ensures transferability. Note that in a given view, the `Admission` component enqueues at most one `admit` message, and the `Failure_Detect` component enqueues at most one `remove` message in the `GMA` component's message queue.

The `start()` method of the `GMA` component initializes the protocol variables according to the initialization pseudocode given in Figure 5.5. After the `start()` method returns, the `svc()` method enters the main event loop that invokes the `add_message()` method for each message dequeued from the protocol instance's `ACE_Message_Queue`. The `add_message()` method demarshals and processes messages as specified by Protocol GMA, described in Chapter 5.

When the `GMA` component dequeues an `admit` or `remove` message from its message queue, it initiates the protocol for agreement on the next group membership, unless that protocol has already been initiated in the current view.

Upon receiving a `pre-proposal` message from another replica P_i, a replica P_j's `GMA` component checks the validity of the message by invoking the following static `validate` interface at the `Admission` or `Failure_Detect` component depending on whether the flag field in the message specifies an addition or removal respectively.

```
    bool validate (ACE_INET_Address addr, ACE_Message_Block* proof);
```

The parameters passed are the fields of the received `pre-proposal` message. For addition, the `validate` function at the `Admission` component checks whether the `proof` parameter justifies a non-member replica P_k's admission to the group. The parameter `addr` specifies P_k's identifier as it appears in the universal-list. For removal, the `validate` function at the `Failure_Detect` component in turn invokes the `validate` function at the appropriate `Validation` subclass, which is chosen based on the protocol type and fault type. The `validate` function at the subclass instance checks whether the `proof` parameter justifies the member replica P_k's removal from the group. If the justification is valid, then the `validate` call to the `Admission` or the `Failure_Detect` component (as the case may be) returns `true`.

P_i collects $|V_x| - t$ `pre-proposal` messages from distinct replicas in a proposal vector $\mathcal{C}$ and proposes $\mathcal{C}$ for MVBA. The `GMA` component creates a MVBA instance with protocol identifier `memagree|x+1` passing a `Mem_Validation` object to the MVBA constructor. The `Mem_Validation` class derives from the abstract `Validation` class and implements the `validate` interface declared in the abstract class. The object is used in verifying the validity of any replica's proposal for the MVBA. The interface has the form

```
    bool validate (ACE_Message_Block* proposal);
```

The implementation of the interface in the `Mem_Validation` class checks the validity of each of the $|V_x| - t$ `pre-proposal` messages contained within the `proposal` by invoking the static `validate` interface of the `Admission` or `Failure_Detect` component as described above.

The next group membership V_{x+1} is computed deterministically from the decision vector of the MVBA as $V_{x+1} = V_x \setminus \mathcal{D} \cup \mathcal{A}$. Here, $\mathcal{D}$ such that $\mathcal{D} \cap V_x = \mathcal{D}$ is the deterministically ordered list of all replicas for which the decision vector has a `pre-proposal` message specifying removal from the group. $\mathcal{A}$ such that $\mathcal{A} \cap V_x = \emptyset$ is the deterministically ordered list of all replicas for which the decision vector has a `pre-proposal` message specifying addition to the group. By the agreement property of MVBA, all correct members of V_x will decide on the same value for V_{x+1}.

Stabilization of CoBFIT Protocol Instances before Membership Change

After V_{x+1} is determined, the `GMA` component notifies the `Reconfiguration` component by invoking the following interface at the latter:

```
    void start_reconf (ACE_Array<ACE_CString> next_view);
```

The invocation results in a series of steps that transition the CoBFIT protocols from the current view to the next view. The first among these steps is to ensure that the agreement and execution phase protocols at all correct replicas stabilize, i.e., suspend their operation in the current view at the same protocol state. That means that, at all correct replicas, the atomic broadcast protocol implemented by the `PABC` component atomically delivers the same set of messages in the same order in a given view. Similarly, at all correct replicas, the execution protocol (implemented by the `APE` component) executes all and only those requests specified in the messages atomically delivered in the current view, where the execution order is specified by the order of atomic delivery.

Stabilization of Atomic Broadcast: The `Reconfiguration` component invokes the `stabilize()` method at the `PABC` component. The `Reconfiguration` component then blocks waiting until the `PABC-stabilized()` method at the `Reconfiguration` component is invoked by the `PABC` component.

Inside the `stabilize()` method of the `PABC` component at replica P_i, a special `close` payload of the form $(\texttt{close}, i, \sigma)$ is *a-broadcast*, where σ is P_i's signature on $(\texttt{close}, i)$. The `PABC` component implements the following private function:

```
    void deliver (ACE_Message_Block* payload);
```

The component keeps track of the number of times the `deliver()` function has been passed a `close` payload. At the end of the epoch in which `close` payloads *a-broadcast* by $t + 1$ distinct replicas are *a-delivered*, the `PABC` component invokes the following method at the `Reconfiguration` component:

```
    void PABC-stabilized (ACE_Message_Block* payld);
```

After that invocation, the `PABC` component will not *a-deliver* any further payloads in the view V_x. The `payld` parameter indicates the last atomically delivered non-`close` payload; `payld` also indicates the last client request that has to be executed by the `APE` component in the view V_x.

Stabilization of Service State: After dequeuing the `PABC-stabilized` message, the `Reconfiguration` component invokes the `stabilize (ACE_Message_Block* payld)` method at the `APE` component; the `payld` parameter indicates the last atomically delivered non-`close` payload in this view. The `Reconfiguration` component then blocks waiting for the

APE-stabilized method call from the APE component.

After the invocation of the stabilize method, every time the APE component receives an agreement certificate, it checks to see whether the request certificate contained in the agreement certificate is the same as payld. If so, then the APE component does not have to execute any further client requests in the view V_x. However, the component has to perform two additional tasks: (1) execution of a checkpoint request and (2) bringing of the service state up to date to reflect the execution of all requests up to the checkpoint request. The first task is accomplished just like any other checkpoint request handled by Protocol APE. After a backup has obtained a reply certificate containing the digest of the stable checkpoint, the backup can complete the second task according to the pseudocode of the *state_update*() function given in Figure 5.17.

After completing those two tasks, the APE component invokes the following method at the Reconfiguration component:

```
    void APE-stabilized (ACE_Message_Block digest);
```

The parameter digest contains the digest of the last stable checkpoint.

Conveying the Next Membership to the Joining Replica

If V_{x+1} includes new members, then after the APE-stabilized method invocation, the Reconfiguration component invokes the following method at the Admission component:

```
    void send_new_view (ACE_Array<ACE_CString> next_view,
                                        ACE_Message_Block* stable_state_digest);
```

The parameter next_view is an array of identifiers indicating the next CoBFIT-group membership, and the parameter stable_state_digest indicates the digest of the last stable checkpoint in the current view V_x.

Inside the send_new_view method, the Admission component determines the list of new members by comparing the argument next_view (also denoted by V_{x+1}) with the current view V_x, and to each new member P_k, the component sends a new-view message through the `UDP-Link`. The message contains the next group membership `next_view` and the digest `stable_state_digest` of the last stable checkpoint in view V_x. After sending the message, the `send_new_view` method call returns.

At the new member P_k, the `Admission` component demarshals the received `new-view` message. The component maintains a hash-table mapping the string representation of `next_view` and `stable_state_digest` to a counter that indicates the number of `new-view` messages containing that information received from distinct members of view V_x. The hash-table provides a way to check whether `new-view` messages with the same `next_view` (that includes P_k) and `stable_state_digest` information have been received from $t + 1$ distinct members of view V_x. If they have been, the new member prepares to become a member of the `CoBFIT-group` from view V_{x+1}. The preparation involves two steps, namely the establishment of `TCP-Link`s with the other members of view V_{x+1} and the updating of the state; we describe these steps below.

Reconfiguration of the `Network` Component

The `Network` component provides the following interface for receiving reconfiguration notifications from the `Reconfiguration` component:

```
    void reconfigure (ACE_Array<ACE_CString> next_view);
```

If the `next_view` array includes new members, the `Reconfiguration` component invokes the above interface after dequeuing the `new_view_sent` message from the `Admission` component; otherwise, the `Reconfiguration` component invokes the interface after dequeuing the `APE-stabilized` message from the `APE` component.

The `Network` component then reconfigures the `TCP-Link`s appropriately. At a replica $P_i \in V_x \cap V_{x+1}$, any `TCP-Link` that connects with a replica $P_j \in V_x \setminus V_{x+1}$ is garbage-collected.

Once a replica P_k that wants to join the group has received `new-view` messages with the same `next_view` (that includes P_k) and `stable_state_digest` information from $t+1$ distinct members of view V_x, replica P_k starts the procedure of establishing `TCP-Link` components with other members of view V_{x+1}. In view V_{x+1}, P_k will have a higher rank than any $P_i \in V_x \cap V_{x+1}$. So, according to the acceptor-connector rule mentioned in Section 6.6.2, P_k plays the role of a connector and P_i the role of the acceptor for establishing point-to-point communication between them. The `Network` component at connector P_k instantiates a `TCP-Link` component for exclusive communication with P_i. The `TCP-Link` at P_k then tries to establish a socket connection with the specified acceptor port of P_i. P_k keeps trying until it finally succeeds. Acceptor P_i just waits for socket connections from new members at its acceptor port. When a connection is made, P_i accepts the connection, spawns a new socket, and instantiates its own `TCP-Link` with the new socket information.

After P_i and P_k have established a `TCP-Link` for exclusive communication with each other, they obtain a secure authenticated channel from the `TCP-Link` through the same handshake procedure described in Section 6.6.2.

After the authenticity of $|V_{x+1}| - t$ `TCP-Link`s has been established, the `Network` component at a member of V_{x+1} invokes the `network_ready()` method at the `Reconfiguration` component.

Transferring Service State to the Joining Replica

Inside the `network_ready()` function at the `Reconfiguration` component of a new group member P_k, the `start_state_update()` function at the `APE` component is invoked. The `Reconfiguration` component then block waits for the `APE_state_updated()` method call-back from the `APE` component.

The `APE` component at P_k starts the procedure to bring its (service) state up-to-date, so that the state digest is the same as the `stable_state_digest` information in the `new-view` messages from $t+1$ distinct members of V_x that confirmed P_k's inclusion in the view V_{x+1}. P_k can complete the state update according to the pseudocode of the $state\_update_{ape}$ function (with `new-view` and `stable_state_digest` as arguments) given in Figure 5.16. After the state update is complete, the `APE` component invokes the `APE_state_updated` method at the

`Reconfiguration` component.

Starting Regular Operation in the New View

After dequeuing the `Network-ready` message, the `Reconfiguration` component at a replica $P_i \in V_x \cap V_{x+1}$ invokes the `done_reconfigure()` methods successively at the `Admission` component and the `Failure_Detect` component. The method is passed two arguments: `next_view`, which is an array of identifiers of parties constituting the next membership, and `x+1`, which is the new view identifier.

Inside the `done_reconfigure` method call, the `Admission` component resets the `recommend` list to $\emptyset$. The component updates the `removal_time` and the `rejuv_time` fields in the `universal_table` for each removed replica $P_k \in V_x \setminus V_{x+1}$. For each new replica $P_k \in V_{x+1} \setminus V_x$, the component updates the `join_time` field in the `universal_table`.

Replicas that requested admission to the `CoBFIT-group` in view V_x but were unsuccessful have to retry the admission protocol in view V_{x+1}. The trusted dealer (that creates any replica) notifies such replicas of the new view V_{x+1}, so that they may retry.

When the `done_reconfigure` method call at the `Admission` component returns, the `Reconfiguration` component invokes the `done_reconfigure` method at the `Failure_Detect` component. Inside the `reconfigure` method call, the `Failure_Detect` component resets the `remove` list.

When the `done_reconfigure` method call at the `Failure_Detect` component returns, the `Reconfiguration` component at a replica $P_i \in V_x \cap V_{x+1}$ invokes the `done_reconfigure()` method successively at the `GMA`, `PABC`, and `APE` components. At that point, those components can begin their regular operation in the view V_{x+1}.

At a new group member, $P_k \in V_{x+1} \setminus V_x$, the `Reconfiguration` component invokes the `done_reconfigure()` method successively at the `Admission`, `Failure_Detect`, `PABC`, `GMA`, `PABC`, and `APE` components after the `APE_state_updated()` method callback from the `APE` component.

6.7 Interaction between Applications and CoBFIT Protocols

The protocols in the CoBFIT toolkit were designed with the client-server paradigm in mind and can be used to make the server fault-tolerant using the state machine replication approach. Both the client and server can be viewed as applications, the former using the services provided by the latter. This section describes how the CoBFIT protocols interface and interact with the *client-type application* and *server-type application*.

There is only one CoBFIT protocol component that needs to communicate with the client-type application for the purpose of receiving request certificates and sending back reply certificates, and that is the atomic broadcast protocol implemented in the `PABC` component. All interaction of the `PABC` component with the client-type application is through the `Client_Interface` component.

There is also only one CoBFIT protocol that needs to communicate with the server-type application, and that is the execution protocol implemented in the `APE` component. The `APE` component interacts with the server-type application only through the `Server_Interface` component.

Both the `Client_Interface` and the `Server_Interface` derive from a common abstract class, `Application`, which is in turn derived from the `Protocol_Instance` class. The `Application` class declares the following interface:

```
virtual int from_CoBFIT (ACE_Message_Block* in_mblk,
                                          ACE_Message_Block* out_mblk = 0) = 0;
```

A CoBFIT protocol component (such as `PABC` or `APE`) invokes the `from_CoBFIT()` method to pass data onto a client-type or server-type application. An implementation of the method by `Client_Interface` or `Server_Interface` will demarshal the data represented by the `in_mblk` parameter and pass it on to the respective application. The `out_mblk` parameter is used to pass the result of the method invocation back to the calling protocol component.

The parameter is not used in the method implementation at the `Client_Interface` component. However, the method implementation at the `Server_Interface` component may use the `out_mblk` parameter to pass back some response to the `APE` component; if, for certain operations, there is no response to be sent back, then `out_mblk` will be set to NULL.

6.7.1 Interfacing with the Client-Type Application

The `All_Clients_Info` Object

There is a *client-list* consisting of the identifiers (IP address and port) of all clients from which requests are accepted by the server-type application. The `All_Clients_Info` object, as the name indicates, stores all information regarding the clients in the client-list. The object is directly created by the `Main` component at all replicas. The object maintains a hash-table `client_table` that maps the identifier of each client C_i in the client-list to a `Client_Info` object that contains the following information:

- C_i's public key,
- an integer `privilege` indicating the privilege level for C_i,
- a `convict` flag (initialized to `false`) indicating whether C_i has previously requested any operation that does not conform to its assigned privilege level,
- `request_timestamp` (initialized to 0) indicating the highest timestamp among the requests received so far from C_i,
- `reply_timestamp` (initialized to 0) indicating the timestamp on the request from C_i for which a reply certificate has been obtained, and
- `reply_cert` (initialized to NULL), which stores the reply certificate corresponding to C_i's request timestamped with `reply_timestamp`.

The Client_Interface Component

It is the requests from the client-type application that Protocol PABC atomically delivers to Protocol APE. The client-type application is not replicated and is run on hosts distinct from the hosts at which replicas are run. Each instance of the client-type application represents a distinct client.

The Client_Interface component is loaded along with the PABC component at all replicas; the component is instantiatedby the Main component. The component listens for client requests and forwards the requests to the PABC component through direct method invocations. The client sends its requests to the Client_Interface component at one or more replicas. We guarantee that if a client request is sent to the Client_Interface component at $t + 1$ distinct replicas, then the client will eventually obtain a reply. However, before sending a request to areplica, the client has to establish a *session* with that replica.

The establishment of a session between the client-type application and areplica involves the following steps. At any replica, there is a fixed *client acceptor port* to which the client-type application connects. The Network component at areplica listens for client connections to the port. When a signed connection request is received, the Network component uses the All_Clients_Info object to verify that the connection is from one of the clients in the client-list; if the connection is valid, the component spawns a separate socket for the client and instantiates a new TCP-C-Link component with that socket information. Although not implemented currently, one can change the client authorization to be based on login-password or SSL.

After establishing a session with areplica, the client directly communicates with the TCP-C-Link, which forwards all information received from the client to the Client_Interface component. The Client_Interface in turn sends replies back to the Network component, which dispatches them to the intended client through the appropriate TCP-C-Link component. The session may last several client requests and is terminated when the client closes the connection or after a prolonged period during which no valid client requests were received.

The `Client_Interface` encapsulates the client request as an `ACE_Message_Block` which is passed to the local `PABC` component through the `a_broadcast` interface provided by the `PABC` component, resulting in the atomic broadcast of the client request. Upon obtaining a reply certificate from the execution replicas, the `PABC` component passes the certificate as the argument to the `from_CoBFIT()` method invocation. If a `TCP-C-Link` with the client that sent the request exists, the `from_CoBFIT()` method implementation at the `Client_Interface` component forwards the reply certificate to the client.

By hiding from the `PABC` component the details (such as the request format) that are specific to the particular client-type application being used, and by sending the reply values in a format expected by the client, the `Client_Interface` acts as a two-way adapter. Thereby, it enables the `PABC` component to be implemented in a generic way that can be used for various types of client-type applications.

6.7.2 Interfacing with the Server-Type Application

Server-type applications are the ones being replicated at the execution replicas. It is the state of a server-type application that needs to be kept consistent among execution replicas. They receive requests from client-type applications and send replies to client-type applications via the CoBFIT toolkit. The server-type application, denoted by `Server`, is a separate process that is started along with the CoBFIT `Main` process at the execution replicas.

It is through the `Server_Interface` component that any CoBFIT component interacts with the `Server` process. As mentioned before, the `Server_Interface` component derives from the `Application` abstract class. The component is loaded along with the `APE` component at all execution replicas and is instantiatedby the `Main` component through the `Instance_Handler` component. The component implements the following three methods, all of which are directly invoked by the `APE` component:

```
int from_CoBFIT (ACE_Message_Block* in_mblk, ACE_Message_Block* out_mblk);
ACE_Message_Block* get_state ();
```

```
int set_state (ACE_Message_Block* s);
```

The `APE` component invokes the `from_CoBFIT()` method at the `Server_Interface` encapsulating the client request in the `in_mblk` parameter. The method implementation feeds the request to the `Server` process and block waits for the request processing to complete. If the `Server` process returns a response then the `Server_Interface` encapsulates the response as a message block, which is passed on back to the `APE` component as the `out_mblk` parameter.

Before a server-type application can be made fault-tolerant using the PABC and APE protocols, it must meet two requirements: (1) the application must be deterministic, and (2) the application state must not change in the duration between return of the `from_CoBFIT()` method call at the `Server_Interface` and when the next method invocation.

The `get_state()` and `set_state()` interfaces, as the names indicate, are used to retrieve and update the server-type application state. The interfaces are invoked by the `APE` component as part of the state transfer mechanism it implements for bringing a backup replica up-to-date and for initializing the state of the `Server` process at an execution replica that newly joins the `CoBFIT-group`. The `get_state()` method call obtains the server-type application state in a format specific to the particular application, encapsulates the state in a chain of `ACE_Message_Block`s, and returns the chain. For the `set_state()` method call, the state information encapsulated in the form of a chain of `ACE_Message_Block`s is given as an input parameter; the method converts the encapsulated state into a format that can be processed by the `Server` process.

The `Server_Interface` component encapsulates the details specific to the particular server-type application being used and thereby allows the `APE` component to be implemented in a generic way that can be reused without much change for various types of server-type applications.

6.7.3 Fractal Generator: A Sample Client/Server Application

To experimentally evaluate the performance of the CoBFIT protocol components, we chose a representative application and developed a replicated service by combining the application with the CoBFIT toolkit. The chosen application, Fractal Generator, is run as the `Server` process at the execution replicas, whereas the client is a user that sends fractal parameters via http/php. Client request processing is cpu-intensive and involves the generation of an image file based on the fractal parameters; the reply certificate sent to the client contains the generated image file as the result value. Upon receiving the reply certificate, the client verifies whether the certificate contains signatures from $t+1$ distinct execution replicas on the same result value; if it does, the client accepts the result value to be valid and recreates (through a php script) a `.png` file to display.

The code for the `Server` process was developed using the 3D (quaternionic) fractal generator Quat-1.20 distributed by the University of Stuttgart under open source license and available from `http://www.physcip.uni-stuttgart.de/phy11733/index_e.html`.

An application like Fractal Generator would be useful in an environment where images or maps are to be generated based on various parameters entered by a remote client or gathered via other means, such as sensors. The image generated (in this case, a .png file) may be important and sensitive enough to require the use of replicated servers and the provision of high availability and intrusion tolerance.

Chapter 7

Experimental Evaluation

In this chapter, we describe the experimental evaluation of the protocols implemented in the CoBFIT toolkit. We describe a stand-alone evaluation of Protocol APE and Protocol PABC. Then, we evaluate the two protocols working in concert to implement a replicated version of the Fractal Generator application. Finally, we describe the evaluation of the reconfigurable RMan suite of protocols.

In all our experiments, digital signatures, when used, were generated using 1024-bit RSA cryptography. MACs for realizing secure authenticated channels were computed using the SHA-1 algorithm. Confidence intervals are for 95% confidence and were computed under the assumption that the samples were from a student's t-distribution.

7.1 Experimental Evaluation of Protocol APE

In this section, we describe the experimental evaluation of 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.

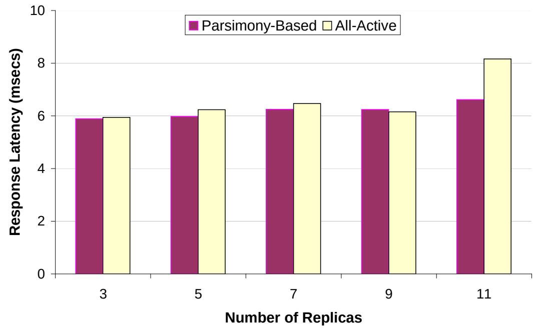
The fact that the execution phase of a BFT-replicated service will be service-specific poses a challenge to obtaining useful results. The resources involved during request processing will be service-specific, and even request-specific. In our experiments, we have tried to account for that fact by varying 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.

7.1.1 Experimental Setup

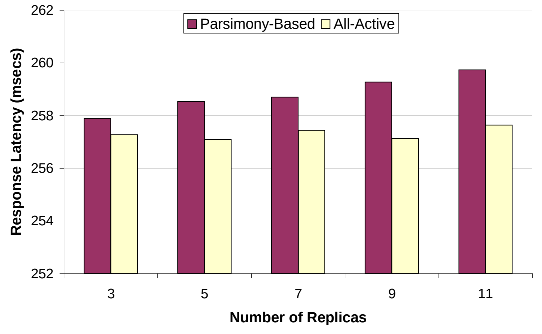
We conducted our experiments for 3, 5, 7, 9, and 11 execution replicas that can tolerate $t = 1, 2, 3, 4,$ and 5 simultaneous replica faults, respectively. In a real-world setting, $\mathcal{AS}$ would consist of a set of $3t + 1$ agreement replicas; however, to keep the focus on the execution phase of BFT replication, the clients and the agreement replicas were represented by a single $\mathcal{AS}$ process that generated requests and provided the properties given in Section 4.2.

The setup consisted of a LAN 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 $\mathcal{AS}$ process. At most one execution replica ran on the other machines. Each replica maintained about 1 MB of service-specific state, organized into 1 KB blocks and loaded into its main memory at initialization time. The machines were connected by a lightly loaded full-duplex 100 Mbps switched Ethernet network. Communication among the replicas was through IP multicast involving UDP datagrams.

$\mathcal{AS}$ sends two kinds of requests: *retrieve-compute* requests and *update-compute* requests. Additionally, for Protocol APE, every 200th $\mathcal{AS}$ request is a checkpoint request. A *retrieve-compute* request specifies a block to be retrieved. A replica performs some computation on the contents of the block, and returns the result 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



(a) CL-2 Requests



(b) CL-100 Requests

Figure 7.1: Request Latency

of the `reply` message for either type of request contains the result of the computation and has the size of a block (1 KB). $\mathcal{AS}$ sends a new request after obtaining a reply certificate for its last request.

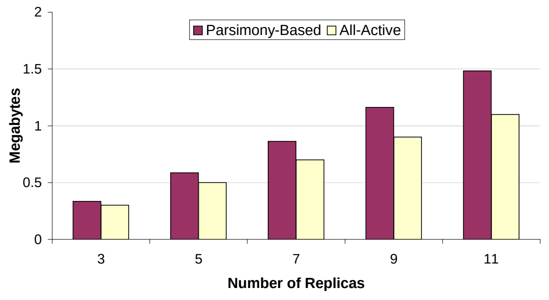
7.1.2 Behavior in Fault-Free Runs

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

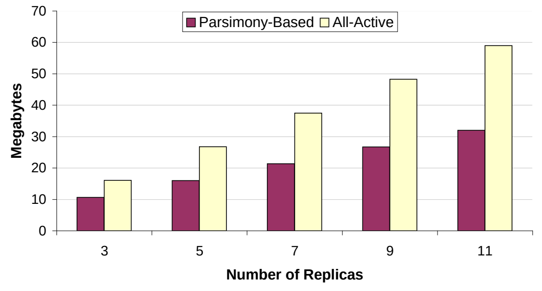
We measured request latency, which is the time elapsed from when $\mathcal{AS}$ sends a request until it obtains a reply certificate for the request. Figure 7.1(a) compares the request latencies of the parsimonious and the all-active execution approaches for CL-2 requests. Figure 7.1(b) does the same 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 $\mathcal{AS}$ process sending about 10,000 requests. $\mathcal{AS}$ 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.

Figures 7.1(a) and (b) show only a small difference in the request latencies between all-active and parsimonious execution. For CL-2 requests (Figure 7.1(a)), the request latencies for all-active execution are slightly higher than those for parsimonious execution. The reason is that in all-active execution, even though $\mathcal{AS}$ needs only $t+1$ `reply` messages with identical result values to accept the result, it will receive `reply` messages from all replicas (i.e., $2t+1$ messages), since the runs were fault-free. Though $\mathcal{AS}$ fully processes only $t+1$ of those messages and discards the other t, there is overhead involved in receiving the additional unnecessary messages and examining their headers. Thus, one can expect higher latencies for all-active execution if the `reply` message sizes are increased. For compute-intensive CL-100 requests (Figure 7.1(b)), the latencies for all-active execution are slightly lower than those for parsimonious execution. The reason is that all-active execution allows $\mathcal{AS}$ to choose the fastest $t+1$ replies among the $2t+1$ replies that will eventually be received at $\mathcal{AS}$.

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). Figures 7.2(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,



(a) Overall Ingress Traffic



(b) Overall Egress Traffic

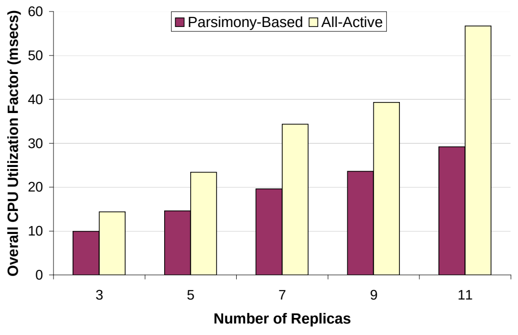
Figure 7.2: Overall Traffic Across All Replicas for 5000 CL-2 Requests

as the only difference would be the size of request messages from $\mathcal{AS}$.

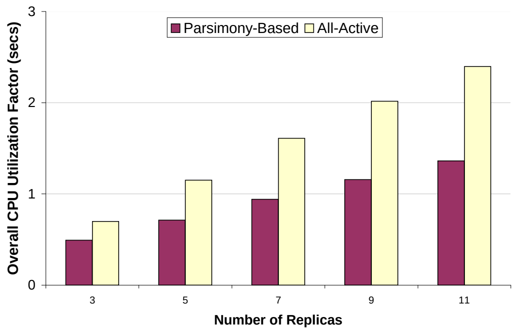
Figure 7.2(a) shows that the overall ingress traffic for all-active execution is smaller than that of parsimonious execution, and that the difference grows modestly as the number of replicas increases. The only incoming messages at a replica for all-active execution are the request messages from $\mathcal{AS}$. For parsimonious execution, in addition to the request messages, `reply` messages are received for infrequent checkpoint requests from $\mathcal{PC}$ members. 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).

Figure 7.2(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 $\mathcal{AS}$. The difference (and hence the advantage of parsimonious execution) is more pronounced as t increases. If overall (ingress + egress) network traffic were to be considered, this benefit of parsimonious execution would far outweigh its drawback with respect to overall ingress traffic.

Since in our experiments the CPU is the dominant resource used at a replica in processing $\mathcal{AS}$ 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



(a) CL-2 Requests

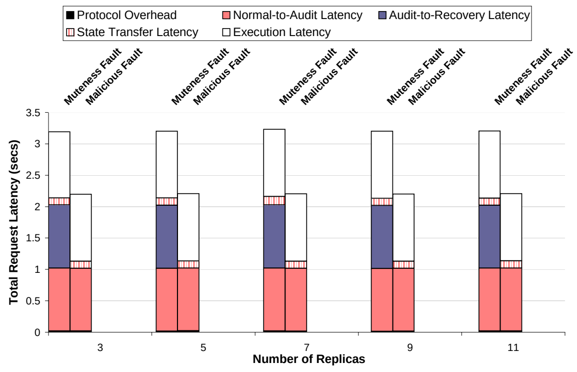


(b) CL-100 Requests

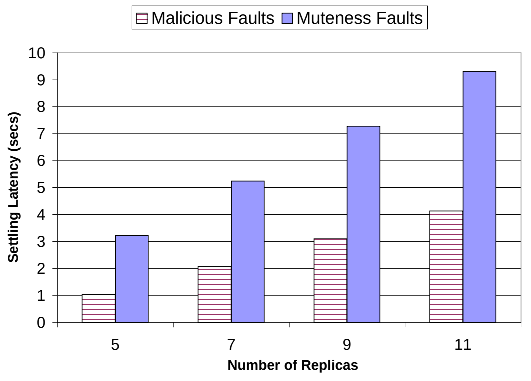
Figure 7.3: Overall CPU Utilization Factor Per Request

run (a run spanned the time it took to process 10,000 $\mathcal{AS}$ 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. Figures 7.3(a) and (b) show the overall CPU utilization factor for CL-2 and CL-100 requests. The utilization factor for parsimonious 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 4.5.2), 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.



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



(b) Settling Latency for Multiple Correlated Faults

Figure 7.4: Behavior Under Fault Injections

7.1.3 Behavior in the Presence of Fault Injections

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

Figure 7.4(a) shows the different factors that contributed to the $\mathcal{AS}$ request latency when a $\mathcal{PC}$ member was fault-injected after servicing a sufficiently large number of CL-2 requests (about 5,000). We injected both muteness faults and malicious faults in our protocol. A muteness fault injection was done by crashing a $\mathcal{PC}$ member upon receipt of a specified $\mathcal{AS}$ request. A malicious fault injection was done by making a $\mathcal{PC}$ member send wrong values in its `reply` messages.

The highest latency is obtained when a muteness fault injection is done. The latency comprises two timeout values, state update latency, and the protocol-specific overhead. The first timeout value of 1 second (represented by "normal-to-audit latency" in the graph) is used at $\mathcal{AS}$ before $\mathcal{AS}$ 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 $n - t$ `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 $\mathcal{AS}$ 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 $\mathcal{AS}$ 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 $\mathcal{AS}$ to the replica (for the request message from $\mathcal{AS}$ 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 $\mathcal{PC}$). We see that an overwhelmingly large portion of the request latency when a muteness fault is injected depends on tunable system parameters (like timeout) and service-specific values, such as the size of the application state, the checkpointing technique used, the number of requests beyond the last stable checkpoint that have to be executed to bring the state up to date, and the normal request processing latency. The actual overhead due to Protocol APE is less than 20 milliseconds.

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

Figure 7.4(b) quantifies the effect that multiple correlated fault injections have at the replicas. After servicing a sufficiently large number of requests (about 5,000), we injected multiple faults at the replicas so that a new $\mathcal{PC}$ member fault was activated every time an $\mathcal{AS}$ request arrived until the fault resiliency t of the replication group was exhausted. As before, the $\mathcal{AS}$ 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 $\mathcal{AS}$ 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 $\mathcal{AS}$ timeout plus the timeout at replicas), as opposed to only $(t-1)$ $\mathcal{AS}$ timeouts for t malicious fault injections.

7.2 Experimental Evaluation of Protocol PABC

In this section, we describe the experimental evaluation of Protocol PABC under both fault-free conditions and controlled fault injections in both LAN and WAN environments.

7.2.1 Experimental Setup

We conducted our LAN experiments for $n = 4$, 7, and 10 agreement replicas that can tolerate $t = 1$, 2, and 3 simultaneous replica faults, respectively. The setup consisted of a testbed

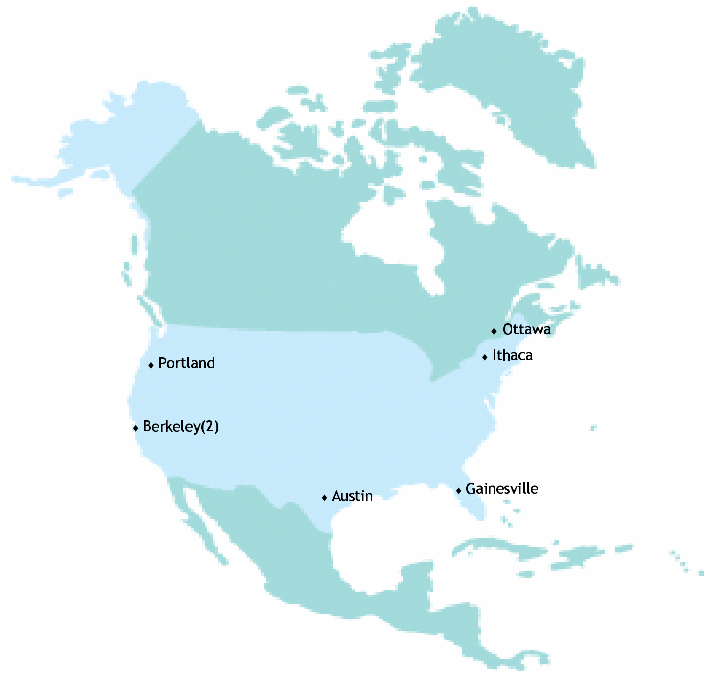
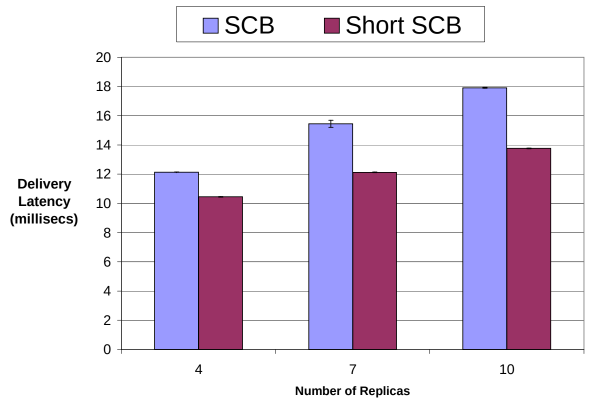


Figure 7.5: Locations of Planetlab Nodes Used

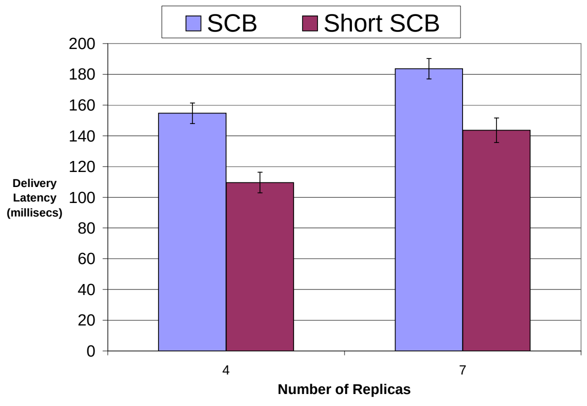
of 10 otherwise unloaded machines. Each machine had a single Athlon XP 2400 processor and 512 MB DDR2700 SDRAM running Fedora Core 3. At most one replica ran on each machine.

To study the behavior of Protocol PABC in WAN settings, we conducted experiments on Planetlab, which is a planetary-scale meta-testbed that provides a large set of geographically distributed machines. The machines are connected by an overlay network that has the characteristics of a realistic network substrate: congestion, failures, and diverse link behaviors. On Planetlab, we conducted experiments for $n = 4$ and 7 agreement replicas that can tolerate $t = 1$ and 2 simultaneous replica faults, respectively. Replicas were placed in distinct machines. The locations of the machines used for our experiments are shown in Figure 7.5. The machines had different hardware configurations and varying load conditions. However, all the machines ran Fedora Core 2. The average round-trip message delays between the machines was on the order of tens of milliseconds.

In both LAN and Planetlab experiments, communication among the replicas was through



(a) LAN



(b) Planetlab

Figure 7.6: Delivery Latency: Strong Consistent Broadcast versus Short Strong Consistent Broadcast

secure authenticated channels implemented using `TCP-Link`s as described in Section 6.2.1.

7.2.2 Behavior in Fault-Free Runs

Strong Consistent Broadcast Latency

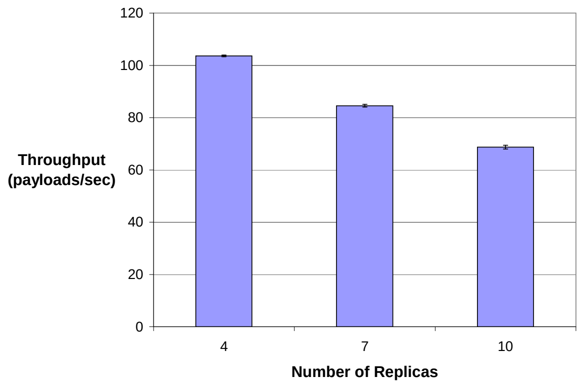
In the parsimonious mode, the performance of Protocol PABC depends on that of strong consistent broadcast. Hence, we conducted experiments for evaluating the latency of a payload that is delivered among the parties using strong consistent broadcast.

For that purpose, we devised an experiment in which the parties are started and wait for the group size to reach a specified target group size (4, 7, or 10) before beginning to transmit messages. Each party then instantiates a strong consistent broadcast instance in which the party with rank 0, namely P_0, is the designated sender. Party P_0 *sc-broadcasts* a payload of size 512 bytes. On *sc-delivering* the payload, that *sc-broadcast* instance is garbage-collected, and the parties instantiate the next broadcast instance in which P_1 is the designated sender. On *sc-delivering* the payload from P_1, the parties then garbage-collect the corresponding *sc-broadcast* instance, instantiate the next broadcast instance in which P_2 is the designated sender, and so on. Party P_0 *sc-broadcasts* its next payload when it *sc-delivers* the payload for the broadcast instance in which P_{n-1} is the designated sender. The results reported here were obtained from 50 independent runs for the LAN testbed and 25 independent runs for Planetlab; each run collected data for varying group sizes, and involved, for each group size, the *sc-delivery* of 225 payloads. We discarded the measures from the first 25 broadcast instances, thereby allowing the group to reach a more stable condition after its formation.

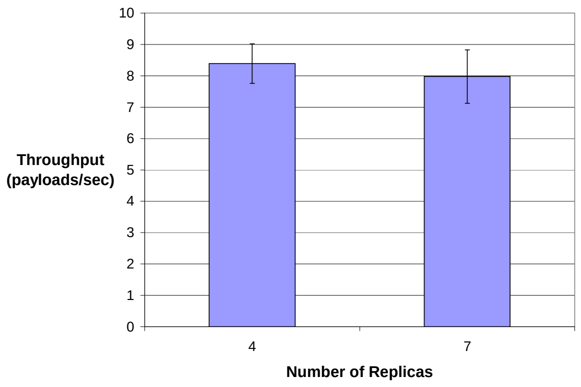
Figures 7.6(a) and (b) compare the average *sc-delivery* latency for a payload for strong consistent broadcast and short strong consistent broadcast. The figures show that the *sc-delivery* latency in Planetlab is an order of magnitude higher than that in the LAN testbed. That is expected, since the message transmission delays on Planetlab are significantly higher than those in the LAN testbed. The figures also show the average *sc-delivery* latency for a payload as the group size increases. We can see from the figure that the latency increases with increase in the group size. There are two reasons. First, the message traffic increases with an increase in the group size. Second, the increase in group size (n) in our experiments is accompanied by a corresponding increase in fault resilience (t); hence, the number of signed echoes that need to be verified ($n - t$) for each *sc-delivery* also increases. We can also note from the figure that short strong consistent broadcast has lower *sc-delivery* latency than strong consistent broadcast. That is because short strong consistent broadcast involves only 2 communication steps, whereas strong consistent broadcast involves 3.

Throughput of Protocol PABC

The throughput of Protocol PABC in the parsimonious mode depends on the availability of non-`dummy` payloads to be *sc-broadcast* by the leader. We devised an experiment in which the leader's initiation queue was filled with a large number of payloads, where each payload was 400 bytes. The leader *sc-broadcasts* a payload and waits for its *sc-delivery* before *sc-broadcasting* the next payload (i.e., only one active *sc-broadcast* instance at a time). Figures 7.7(a) and (b) show the throughput of Protocol PABC (after it reached stable levels) measured as the number of *a-deliveries* per second. The decrease in throughput with increase



(a) LAN



(b) Planetlab

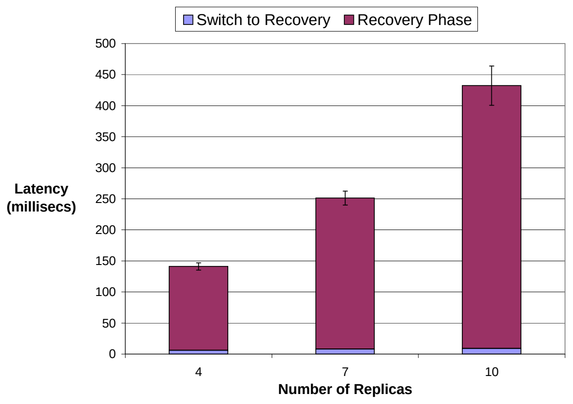
Figure 7.7: Fault-free Throughput of Protocol PABC

in group size is due to the increased *sc-delivery* latency (shown above in Figures 7.6(a) and (b)).

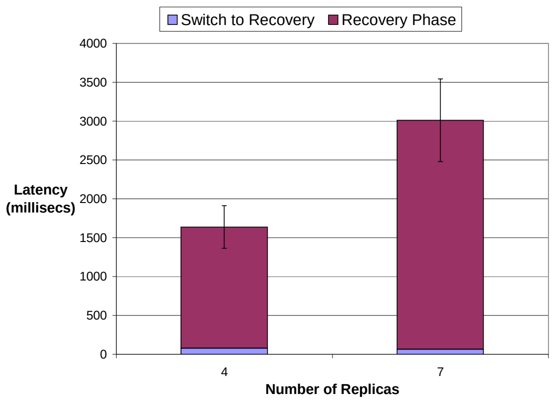
Note that two optimizations that would have enhanced the throughput were not applied here. One is *request batching*, in which each *a-broadcast* carries a batch of (say 100) payloads instead of a single payload. The other is a sliding window mechanism for processing multiple sequence numbers in parallel by processing a window of *sc-broadcast* instances concurrently, instead of processing just one *sc-broadcast* instance at a time.

7.2.3 Behavior Under Fault Injections

Figures 7.8(a) and (b) show the worst-case *a-delivery* latency for an initiated payload minus the timeout value used for leader complaints, when there are no other initiated payloads. To obtain the measurements, we devised an experiment in which all parties initiated the same payload, i.e., sent the payload to the leader so that it could be *sc-broadcast*. The parties then started a timeout, expecting the *a-delivery* of the payload before timeout expiry. The leader was fault-injected to ignore `initiate` messages from other parties. As a result, when the timeouts eventually expired at other parties, they exchanged `complaint` messages about the leader, causing the protocol to switch to the recovery mode. Since more than t parties



(a) LAN



(b) Planetlab

Figure 7.8: *A-delivery* Latency Under Faulty Leader

initiated the payload, the payload was *a-delivered* in part 3 of the recovery mode. The results show the difference in time between when the payload was initiated and when it was *a-delivered*.

For each group size, the figure shows two parts to the *a-delivery* latency. One is the time it takes for the exchange and processing of a sufficient number $(2t + 1)$ of signed `complaint` messages to cause a transition from the parsimonious to the recovery mode. The other is the time taken for the recovery mode to complete. As expected, the recovery mode is expensive, since it involves two MVBA instances. The figure also shows a marked increase in the time taken by the recovery mode as the group size (and thereby the fault resilience) increases.

7.3 Experimental Evaluation of a Replicated Service

In this section, we evaluate a service that is replicated for fault tolerance using the RMan suite of protocols. First, we present results for the agreement phase and the execution phase working in concert to process client requests. Second, we present results for group reconfiguration when the number of replicas is dynamically changed.

We deployed the replicated service in both the LAN testbed and Planetlab. The experimental setup is similar to that described in Section 7.2.1. For the experiments involving the static version of the replicated service, an additional node was used to host the client. For the group reconfiguration experiments, an additional node was used to host a non-member replica that was seeking admission into the replication group. In the LAN testbed, the additional node had the same hardware configuration as the replica nodes. In the Planetlab experiments, the additional node was located at the University of Kentucky, USA.

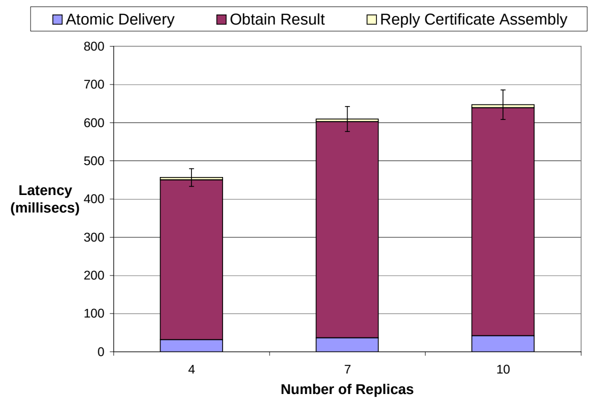
All replicas participated in the agreement phase, but among those only $n-t$ (the ones with ranks from 0 to $n-t-1$) participated in the execution phase. At each replica participating in the execution phase, the Fractal Generator application was loaded as a separate process using the ACE Service Configurator framework.

The results reported in this section are for fault-free runs in which both Protocol APE and Protocol PABC functioned in their respective parsimonious modes. The $t+1$ lowest-ranked replicas constituted the primary committee for Protocol APE. The replica with the rank 0 served as the leader for Protocol PABC.

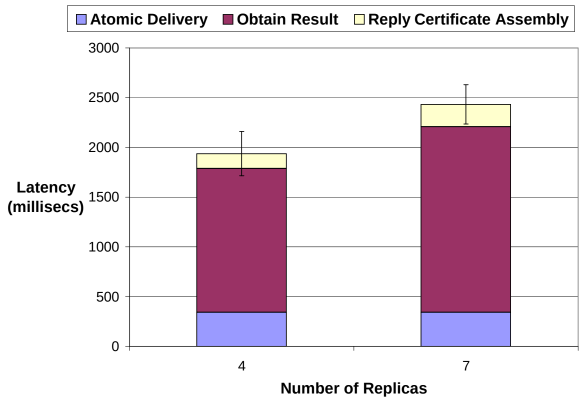
7.3.1 Client Request Processing

Figures 7.9(a) and (b) show the latency for a "lone" client request that brings out the behavior of Protocol PABC when no further payloads are available after the *sc-delivery* of an application payload. Under such circumstances, in order to cause the *a-delivery* of the *sc-delivered* payload, the leader must *sc-broadcast* two `dummy` payloads, each after a timeout. The total request latency is the time between when the client sends its request to the replicas and when it extracts the result from a valid reply certificate. The figures show the total request latency minus twice the timeout value (one for each `dummy` payload) for varying group sizes.

There are three parts to the latency shown in the figures. The first part (indicated by "Atomic Delivery") is the atomic delivery latency, which is essentially the time for three *sc-deliveries*, the first of which is the payload containing the client request, and the other two



(a) LAN



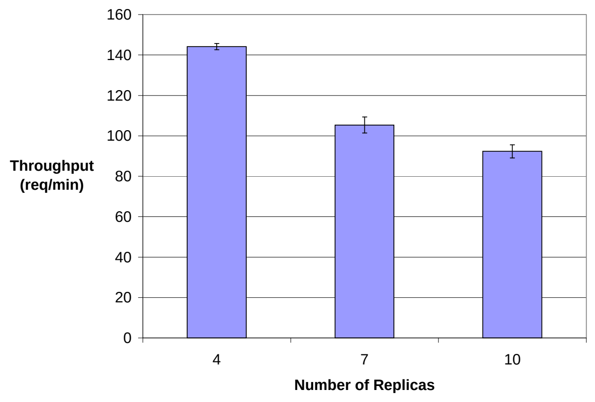
(b) Planetlab

Figure 7.9: Latency for a Lone Client Request

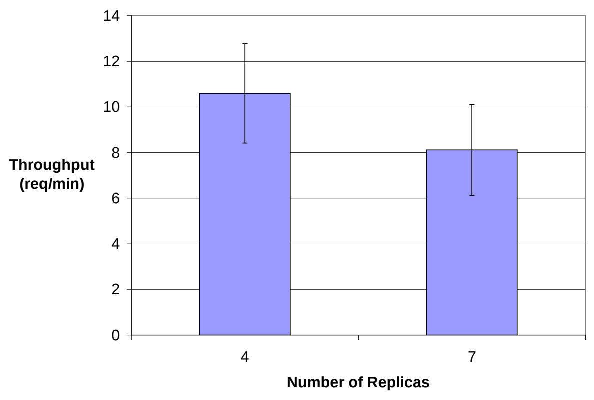
of which are for the `dummy` payloads. The second part of the latency (indicated by "Obtain Result") shown in the figures is the actual request execution latency, i.e., the time between when Protocol APE at a committee member issues a request to the Fractal Generator service to execute the client request and when a result value is returned. The third part of the latency (indicated by "Reply Certificate Assembly") is the time to gather a reply certificate from the signed `reply` messages sent by the committee members and forward the certificate to the client.

Request processing by the Fractal Generator is quite resource-intensive and takes on the order of a few hundred milliseconds. Hence, request processing dominates the latency shown in the figure.

Figures 7.10(a) and (b) show the throughput for the replicated service (after it reached stable levels) measured in number of requests per minute. We devised an experiment in which there were always enough client requests to be initiated that it obviated the need for *sc-broadcasting* of `dummy` payloads. The decrease in throughput with increase in group size is expected given that (1) the throughput of Protocol PABC decreases with increasing group size (shown above in Figures 7.7(a) and (b)) and (2) the time to assemble a reply certificate also increases with group size (shown above in Figures 7.9(a) and (b)).



(a) LAN



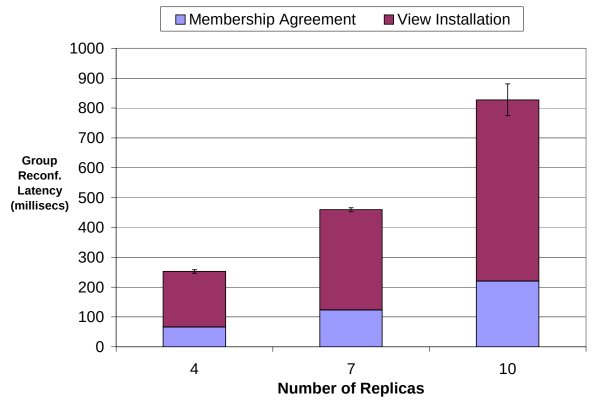
(b) Planetlab

Figure 7.10: Throughput of the Replicated Service

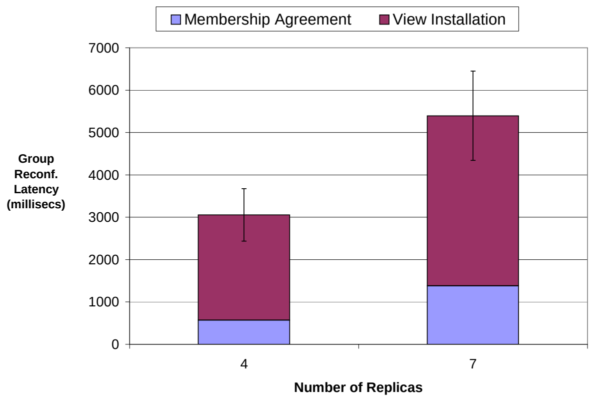
7.3.2 Group Reconfiguration

We devised an experiment in which after the formation of the initial group of specified size (4, 7, or 10), a non-member party sends a request for admission into the group. At each group member, after verification that the request is from one of the non-member parties in the universal set, the group reconfiguration procedure is initiated.

Figures 7.11(a) and (b) show the time it takes to effect a dynamic change to the group membership. The figures show a marked increase in the group reconfiguration latency with increase in group size. Group reconfiguration consists of two parts: group membership agreement (in which parties agree on the next group membership) and view installation (in which the protocols at a party make a systematic transition to the next view). Group membership agreement involves one MVBA, whereas view installation includes the stabilization of Protocol PABC (which itself may involve up to two MVBA instances in the recovery mode), stabilization of Protocol APE, conveyance of the new membership to the newly admitted replica, establishment of secure authenticated channels with the newly admitted replica, and transfer of service state to the new replica. The normal operation of the replicated service is blocked for the duration of the view installation. As one would expect, because of the greater number of steps involved in view installation compared to group membership



(a) LAN



(b) Planetlab

Figure 7.11: Group Reconfiguration Latency

agreement, the former dominates the group reconfiguration latency.

In the above experiment, there was no state transfer involved, as our application implementation was stateless. However, in a real-world setting, there will be additional latency due to state transfer; the exact value of that latency will depend on the size of the application state and the state transfer technique used.

Chapter 8

Conclusion

The design and implementation of trustworthy distributed systems is a research area that deserves increasing attention because of society's increased dependence on such systems. We considered the subject of enhancement of a system's trustworthiness through Byzantine-fault-tolerant state machine replication. Fault tolerance is an important means of providing trustworthiness, which is judged by a system's ability to meet stated goals despite accidents, design and implementation errors, operator errors, and malicious attacks. Because of the Byzantine fault model's generality and the fact that it makes no assumptions about the behavior of a faulty node, we chose to model intrusions and accidents with Byzantine faults.

In this thesis, we presented parsimonious protocols for both the agreement and execution phases of BFT replication. The atomic broadcast protocol for the agreement phase can be made to be parsimonious either in the number of protocol messages that need to be communicated or in the number of communication steps involved per atomically delivered payload. The execution protocol is parsimonious in the overall amount of resources used for request execution.

To provide parsimony, we have taken an optimistic approach based on the practical observation that conditions are normal during most of a system's operation. Under conditions that are perceived to be normal, the protocols operate in a parsimonious mode that maximizes efficiency (measured in terms of overall resource use for the execution protocol and either communication overhead or latency for the atomic broadcast protocol). Under conditions that are indicative of failures or instability, they temporarily switch to a more expensive recovery mode that ensures that some progress is eventually made, after which the protocols switch back to the parsimonious mode. Indications of normality are obtained based on timing assumptions. While inaccuracy of these timing assumptions may affect the protocols' ability to provide parsimony, the protocols are designed such that those assumptions never affect the protocols' ability to provide safety and liveness.

We presented a suite of group management protocols that allow the fault resilience of the replication group to be dynamically changed. The suite includes reconfigurable versions of the atomic broadcast and execution protocols. While group reconfiguration capability has been previously provided in the context of group communication systems, the replication and even group management protocols in those systems required the removal of nodes perceived to be faulty to make progress. Such a requirement is a serious denial-of-service vulnerability that, if exploited, could result in the disintegration of the group. To avoid this pitfall, we have designed our atomic broadcast and execution protocols so that they never require the removal of faulty replicas to make progress. Such a design affords the luxury of using group reconfigurability very selectively and conservatively.

We have developed a software toolkit called CoBFIT that combines the implementation of the parsimonious execution, parsimonious atomic broadcast, and group management protocols within a reusable software framework and that can be used to build trustworthy Internet-scale services. We described the experimental evaluation of the protocols implemented in the CoBFIT toolkit. The results showed the trade-off between significantly superior efficiency characteristics obtained during perceived normal conditions and higher latencies during perceived failure or instability conditions. It is reasonable to expect that a system's operation will alternate between long periods of normality and short periods of instability. That motivated our decision to optimize the system for the common case, even if it means paying a slightly higher cost during periods of instability.

Although the work in this thesis takes an important step forward in enhancing the applicability of BFT replication for building trustworthy distributed systems, there are specific and challenging ways to extend that work.

An extension that could be made to our parsimonious protocols would be to substitute places where a quorum of digital signatures is required with threshold signature shares. Such an extension would be useful from the client's point of view, since instead of having to remember the public key of each replica, it would have to know only one public key, that of the replicated service. Though this extension is fairly easy for the static versions of our protocols, it requires more work for the dynamic versions. That is because the private key of the replicated service needs to be *reshared* among the new members of the replication group whenever there is a group membership change; for this purpose, an asynchronous threshold key resharing protocol is required. With such a resharing protocol, one could easily generalize our group reconfiguration suite of protocols to include both group membership changes and proactive recovery.

The RMan suite of protocols provide the capability for dynamically changing the replication group membership. Policies that guide the decisions regarding when to use that capability are determined by a trusted entity (such as the system administrator) at initialization and do not change thereafter. However, in an security environment that is dynamically changing due to new kinds of attacks, it is desirable to have policies that can be dynamically changed with little manual intervention.

The efficacy of our parsimonious protocols depends on the accuracy of what is perceived to be the system's stable behavior. A challenging course for future work is to adapt this perception based on changes to the system, such as upgrade of a node's hardware capabilities, discovery of faster routes between nodes, and so forth.

References

[BB93] P. Berman and A. A. Bharali. Quick Atomic Broadcast. In *Proc. 7th International Workshop on Distributed Algorithms (WDAG)*, volume 725 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 1993.

[BG93] P. Berman and J. A. Garay. Randomized Distributed Agreement Revisited. In *Proc. 23th International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 412–419, 1993.

[BJ87] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. Computer Systems*, 5(1):47–76, February 1987.

[Bra84] G. Bracha. An Asynchronous $[(n-1)/3]$-Resilient Consensus Protocol. In *Proc. 3rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 154–162, 1984.

[BSTM93] N. Budhiraja, F. Schneider, S. Toueg, and K. Marzullo. The Primary-Backup Approach. In S. Mullender, editor, *Distributed Systems*, pages 199–216. ACM Press - Addison Wesley, 1993.

[Cac01] C. Cachin. Distributing Trust on the Internet. In *Proc. International Conference on Dependable Systems and Networks (DSN-2001)*, pages 183–192, 2001.

[CKLS02] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous Verifiable Secret Sharing and Proactive Cryptosystems. In *Proc. 9th ACM Conference on Computer and Communications Security (CCS)*, pages 88–97, November 2002.

[CKPS01] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and Efficient Asynchronous Broadcast Protocols. In J. Kilian, editor, *Advances in Cryptology: CRYPTO 2001*, Lecture Notes in Computer Science, pages 524–541. Springer-Verlag, 2001.

[CKS05] C. Cachin, K. Kursawe, and V. Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement using Cryptography. *Journal of Cryptology*, 18(3), 2005.

[CL02] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, November 2002.

[CP02] C. Cachin and J. A. Poritz. Secure Intrusion-Tolerant Replication on the Internet. In *Proc. International Conference on Dependable Systems and Networks (DSN-2002)*, pages 167–176, June 2002.

[CR93] R. Canetti and T. Rabin. Fast Asynchronous Byzantine Agreement with Optimal Resilience. In *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 42–51, 1993.

[cry] Cryptlib Toolkit. http://www.cs.auckland.nz/~pgut001/cryptlib/.

[CT96] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.

[Des88] Y. Desmedt. Society and Group Oriented Cryptography: A New Concept. In Carl Pomerance, editor, *Advances in Cryptology: CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 120–127. Springer, 1988.

[DGGS99] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness Failure Detectors: Specification and Implementation. In *Proc. of the European Dependable Computing Conference (EDCC-1999)*, pages 71–87, 1999.

[DGM02] A. Das, I. Gupta, and A. Motivala. SWIM: Scalable Weakly-Consistent Infection-style Process Group Membership Protocol. In *Proc. International Conference on Dependable Systems and Networks (DSN-2002)*, pages 303–312, Washington DC, USA, June 2002.

[DRS90] D. Dolev, R. Reischuk, and H. R. Strong. Early Stopping in Byzantine Agreement. *Journal of the ACM*, 37(4):720–741, October 1990.

[DS02] X. Défago and A. Schiper. Specification of Replication Techniques, Semi-Passive Replication, and Lazy Consensus. Technical Report IC-2002-07, EPFL, Switzerland, February 2002.

[Eln93] E. N. Elnozahy. *Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication.* PhD thesis, Department of Computer Science, Rice University, 1993.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):372–382, April 1985.

[GMR88] S. Goldwasser, S. Micali, and R. L. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[Gol04] O. Goldreich. *Foundations of Cryptography*, volume I & II. Cambridge University Press, 2001–2004.

[Hay98] M. Hayden. *The Ensemble System.* PhD thesis, Department of Computer Science, Cornell University, January 1998. Also as Technical Report no. TR98-1662.

[HT93] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. *Distributed Systems*, pages 97–145, 1993.

[KMMS01] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing Group Communication System. *ACM Transactions on Information and System Security*, 4(4), 2001.

[KMMS03] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Byzantine Fault Detectors for Solving Consensus. *The Computer Journal*, 46(1):16–35, 2003.

[KMT03] Y. Kim, D. Mazzocchi, and G. Tsudik. Admission Control in Peer Groups. In *Proc. IEEE Intl. Symp. on Network Computing and Applications (NCA)*, pages 131–139, 2003.

[KS05] K. Kursawe and V. Shoup. Optimistic Asynchronous Atomic Broadcast. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3580 of *Lecture Notes in Computer Science*, pages 204–215. Springer, 2005.

[Kur02] K. Kursawe. Optimistic Byzantine Agreement. In *Proc. Symposium on Reliable Distributed Systems (SRDS-2002)*, pages 262–267, October 2002.

[Lam78] L. Lamport. Time, Clocks and Ordering of Events in Distributed Systems. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam98] L. Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM TOPLAS*, 4(3):382–401, July 1982.

[MAD02a] J-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine Storage. In *Proc. 16th International Symposium on Distributed Computing (DISC 2002)*, volume 2508 of *Lecture Notes in Computer Science*, pages 311–325. Springer-Verlag, 2002.

[MAD02b] J-P. Martin, L. Alvisi, and M. Dahlin. Small Byzantine Quorum Systems. In *Proc. International Conference on Dependable Systems and Networks (DSN-2002)*, pages 374–388, June 2002.

[MMR00] D. Malkhi, M. Merritt, and O. Rodeh. Secure Reliable Multicast Protocols in a WAN. *Distributed Computing*, 13(1):19–28, January 2000.

[MR97] D. Malkhi and M. Reiter. Unreliable Intrusion Detection in Distributed Computations. In *Proc. 10th Computer Security Foundations Workshop (CSFW97)*, pages 116–124, 1997.

[MR98] D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Journal of Distributed Computing*, 11(4):203–213, 1998.

[MR00] D. Malkhi and M. Reiter. An Architecture for Survivable Coordination in Large Distributed Systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):187–202, March 2000.

[Rab83] M. O. Rabin. Randomized Byzantine Generals. In *Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 403–409, 1983.

[RCL01] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using Abstraction to Improve Fault Tolerance. In *Proc. 18th Symposium on Operating Systems Principles*, pages 15–28, October 2001.

[Rei94] M. K. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proc. 2nd ACM Conference on Computer and Communication Security*, pages 68–80, November 1994.

[Rei95] 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*, volume 938 of *Lecture Notes in Computer Science*, pages 99–110. Springer-Verlag, 1995.

[Rei96] M. K. Reiter. A Secure Group Membership Protocol. *IEEE Transactions on Software Engineering*, 22(1), 1996.

[RPL$^+$02] H. V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders. Quantifying the Cost of Providing Intrusion Tolerance in Group Communication Systems. In *Proc. International Conference on Dependable Systems and Networks (DSN-2002)*, pages 229–238, 2002.

[Sar02] L. F. G. Sarmenta. Sabotage-Tolerance Mechanisms for Volunteer Computing Systems. *Future Generation Computer Systems*, 18(4):561–572, 2002.

[Sch90] F. B. Schneider. Implementing Fault-Tolerant Services using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[Sch97] A. Schiper. Early Consensus in an Asynchronous System with a Weak Failure Detector. *Distributed Computing*, 10(3):149–157, 1997.

[Sch99] F. B. Schneider, editor. *Trust in Cyberspace.* National Academy Press, 1999.

[SCS03] S. Singh, M. Cukier, and W. H. Sanders. Probabilistic Validation of an Intrusion-Tolerant Replication System. In *Proc. 2003 Intl. Conf. on Dependable Systems and Networks*, pages 615–624, June 2003.

[SH02] D. Schmidt and S. Huston. *C++ Network Programming: Mastering Complexity Using ACE and Patterns.* Addison-Wesley Longman, 2002.

[SH03] D. Schmidt and S. Huston. *C++ Network Programming: Systematic Reuse with ACE and Frameworks*. Addison-Wesley Longman, 2003.

[Sho00] V. Shoup. Practical Threshold Signatures. In Bart Preneel, editor, *Advances in Cryptology: EUROCRYPT 2000*, volume 1087 of *Lecture Notes in Computer Science*, pages 207–220. Springer, 2000.

[SSRB01] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley, 2001.

[SZ04] F. B. Schneider and L. Zhou. Distributed Trust: Supporting Fault-Tolerance and Attack-Tolerance. Technical Report TR 2004-1924, Cornell Computer Science Department, January 2004.

[VNC03] P. E. Veríssimo, N. F. Neves, and M. P. Correia. Intrusion-Tolerant Architectures: Concepts and Design. In R. Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*. Springer, 2003.

[VvOM96] S. A. Vanstone, P. C. van Oorschot, and A. Menezes. *Handbook of Applied Cryptography*. CRC Press, 1996.

[YMV$^+$03] 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*, pages 253–267, October 2003.

[ZSvR02] L. Zhou, F. B. Schneider, and R. van Renesse. COCA: A Secure Distributed Online Certification Authority. *ACM Transactions on Computer Systems*, 20(4):329–368, 2002.

Vita

HariGovind Venkatraj Ramasamy was born in Chennai, India on June 18, 1978. He received the Bachelors degree in computer science and engineering from the College of Engineering, Guindy, Anna University in 1999. Hari then moved to Champaign, Illinois, to pursue graduate study in distributed systems. He completed a Master of Science degree in Computer Science from the University of Illinois in 2002. Following the completion of his Ph.D., Hari will begin work for the Computer Science department of IBM Zurich Research Labs, as a postdoctoral fellow doing research on security and privacy.