Consistency and High Availability of Information Dissemination in Multi-Processor Networks

Dissertation submitted for the degree "Doctor of Philosophy"

Idit Keidar

Submitted to the Senate of the Hebrew University in Jerusalem (1998)

This work was carried out under the supervision of
Prof. Danny Dolev.

To Mordechai, Daphna and Merav

Acknowledgments

This thesis brings to conclusion seven wonderful years in which I have been a member of the Transis group, as an undergraduate student, a masters student, and finally a PhD student; years during which I was happy to come to the lab in the morning, and had a smile on my face most of the time (at least after my morning cup of coffee). I am thankful to those who made these years so pleasant and enlightening.

First and foremost, I am deeply grateful to my advisor and the head of the Transis group, Danny Dolev. His vision and personality made the Transis group the successful and cheerful group that it is, has been during all these years, and will surely continue to be in the future. Through these years, Danny has been to me an inspiration, as well as a shoulder to cry on. His belief in me encouraged me to proceed further with my research.

Next, I thank the "forefathers" of Transis: Dahlia Malkhi and Yair Amir, who were also my first mentors in the Transis group. I have learned much from them while they were here. Their seminal ideas inspired much of my research in the years after they left. I also had the pleasure to continue working with Dahlia later on.

This thesis reflects the second generation of Transis: it describes the concepts of the second generation systems which are being developed by a second generation of PhD students (the heirs of the Transis dominion). The system which is the focus of this thesis was envisioned and developed together with Tal Anker and Gregory (Grisha) Chockler. I also greatly enjoy fruitful collaboration on many on-going projects with both.

I have remarkably benefited from many insightful comments and suggestions from Ohad Rodeh; his original ideas were like a fresh breeze on a hot summer day. I have learned a lot from my discussions with Roman Vitenberg and Grisha Chockler on specifications of group communication systems, as reflected in the specifications in this thesis.

The membership service was developed together with Jeremy Sussman from UCSD, and the dynamic voting protocol was developed jointly with Esti Yeger Lotem. I am enormously appreciative of their contribution to this work; it was a genuine pleasure working with both.

My research has also benefited from research and lab projects that were conducted in the Transis group (some of them under my guidance) by Gabriel Benhanokh, David Breitgand, Nabil Huleihel, Zohar Levi, Ariel Nowersztern, Michael Rozman, Gadi Shamir and Jonathan Wexler.

I had the virtue to collaborate with Ken Birman and Roy Friedman and Nabil Huleihel on papers that did not make their way into this thesis. I have benefited from many fruitful discussions with them, as well as with Catriel Beeri, Mark Hayden, Nancy Lynch, Robbert van Renesse, Injong Rhee and many other people around the world, alas it is impossible to name them all.

Finally, I thank my husband Mordechai $\overset{oo}{\smile}$ Keidar for sharing with me all the experiences these years. I thank my daughters Daphna $\ddot{\smile}$ and Merav $\ddot{\smile}$ Keidar (who were both born while I was working on my PhD) for reminding me that there are other things to life besides my research. Special thanks go to my parents for their support, especially my mom, who made a huge effort to come to Jerusalem to watch the kids so I could work longer hours.

Idit Keidar,
Jerusalem, Israel, August 1998.

Contents

List of Figures

Part I

General

Chapter 1

Introduction

This thesis presents general tools for the development of highly available distributed applications such as replicated servers and *Computer Supported Cooperative Work* (CSCW) applications. A desktop and multi-media conferencing tool [Rod91] is an example of a CSCW application, incorporating various activities such as video transmission and management of replicated work space. These services are becoming popular today, with the world-wide increase of communication capacity: Replicated servers in clusters are becoming a leading solutions for scalability, fault tolerance and performance, and world-spanning conferences and interactive games over the Internet are becoming more and more popular. Unfortunately, the subtleties involved in such systems are not well understood and many industries apply ad-hoc solutions without fully understanding their limitations and guarantees.

The contribution of this thesis is in providing application builders with tools and concepts that facilitate the development of such systems while accurately understanding their limitations and guarantees. These concepts are demonstrated and were tested in prototype implementations.

This thesis suggests a comprehensive framework for the development of highly available groupware and CSCW applications, geared towards multi-process failure prone environments (e.g., the Internet). The services are fault tolerant and scalable. The suggested framework incorporates a wide variety of services ranging from efficient communication solutions to tools for maintaining consistency of distributed information in the face of faults. These services support multi-party conferencing in dynamic discussion groups, while keeping track of the dynamically changing set of participants in each group. The service architecture is presented in Chapter 4.

The services exploit the group communication paradigm: Group communication systems provide application builders with reliable multicast communication services within dynamically changing groups, as well as membership services which inform the members when other members crash or join the group. General background on group communication systems is provided in Chapter 3.

Unlike classical group communication systems, our design separates the membership services from the multicast communication substrate. The membership is implemented as a separate server (daemon) on each machine that interacts with the multicast communication substrate and with the application. This separation makes the communication services more efficient since most of the time the membership does not change. This design also facilitates using the same membership services for a variety of *quality of service (QoS)* communication options [CHKD96, BFHR98, RCHS97]. Examples of QoS options include high bandwidth, low latency, and also reliable (loss free) multicast. Decomposing the service into separate modules also makes it easier to reason about, i.e., formally specify the service guarantees and assumptions, and prove correctness. The membership service is presented in detail in Chapter 5.

In addition to the multicast communication service and the membership service, we provide application builders with session level services. These services relieve the application builder of the need to explicitly deal with the subtleties of changes in the network situation. The session services exploit the strong group communication semantics in order to efficiently maintain consistency of objects in the face of failures. This thesis focuses on important building blocks for consistent replication: Chapter 6 presents a highly available Totally Ordered Broadcast service, which may be used, e.g., for consistent replication. Chapter 7 presents a highly available service for maintaining the primary network component in the network. Other session services are described in Section 4.4.

Chapter 6 presents a Totally Ordered Broadcast protocol, which guarantees a fully serializable history of object updates. This is achieved by prohibiting arbitrary updates of the object in disjoint network components; often, only the members of a primary component may update the object. The algorithm in Chapter 6 exploits group communication as a building block. It always allows members of a primary component in the system to update the object. It may be used in conjunction with several types of primary component services that notify processes when they are members of the current primary component e.g., a service based on dynamic voting presented in Chapter 7.

The underlying concepts demonstrated by the services constructed in this thesis are general and apply to a large family of distributed systems and applications.

Fault tolerant distributed services are now being developed by many commercial companies; highly available servers running in clusters are the leading new generation solution for scalability and performance. At the basis of many of these commercial systems lie concepts that were developed in academic projects and systems. The underlying concepts of this work will play a role in future development of such systems. In particular, the formal reasoning we apply to our systems will assists commercial system builders understand the guarantees and limitations behind the systems they construct, and also help identify the tradeoffs involved.

Chapter 8 presents a novel protocol, E3PC, for atomic commitment, that always allows a majority to make progress. The "classical" three phase commit (3PC) protocol [Ske82] sometimes allows a majority to make progress, but if failures cascade, a majority may become connected and still remain blocked. We have identified this shortcoming, and have developed a new structure that allows information to propagate through the sequence of majorities formed in the system. E3PC improves the classical 3PC without adding extra communication by following this structure.

In [FKM$^+$95] we show that this structure is common to all algorithms that always achieve agreement with a majority, and therefore is an important concept for developing fault-tolerant consistent algorithms. The protocol in Chapter 6 exploits similar principles (and bears the same structure).

Chapter 2

The Model*

This thesis is concerned with highly available groupware services in asynchronous partitionable message passing environments. All the protocols presented in this thesis are run among a set of processes connected by an underlying asynchronous communication network, and all tolerate crash and partition failures. This chapter presents the computation model, and introduces definitions which are used throughout this thesis. Section 2.1 introduces the computation model. Section 2.2 describes the asynchronous partitionable failure model, which is common to all the chapters of this thesis. Where different assumptions are made, they are explicitly stated in the relevant chapters.

It is well-known that in asynchronous failure-prone environments, agreement problems such as Consensus and non-blocking atomic commit are not solvable [FLP85, Gue95]. In order to render such problems solvable, the model is often augmented with external failure detectors. In Section 2.3, we define classes of failure detectors for the partitionable failure model. These definitions are used throughout the rest of this thesis.

2.1 The Computation Model

We assume that each process is equipped with a failure detector, which provides hints regarding which processes may be faulty at any given time. The information provided by the failure detectors need not be accurate, although some restrictions on their behavior are imposed. These restrictions are described in Section 2.3.

A process is modeled as a (possibly infinite) automaton, which takes *steps* that consist of receiving `receive` events from the network and *suspect lists* from the failure detector, doing some local computation, and then generating zero or more `send` events. A process p is said to *suspect* another process q if q is in p's suspect list. Events of type `receive` may be empty, i.e., containing null messages. (This allows processes to initiate operations spontaneously). If the message is not empty, we say that the process *receives* a message, or that the message is *delivered* to the process.

*The definitions of failure detectors in this chapter are based on work by Dolev, Friedman, Keidar and Malkhi [DFKM96, FKM$^+$95, DFKM97].

In order to distinguish between the messages sent in different send events, we assume that each message sent is tagged with a unique message identifier, which may consist, e.g., of the sender identifier and a sequence number or a timestamp. Thus, we can require that every message is sent at most once in the system.

A process can also incur `crash` and `recover` events from the environment. Every `recover` event is immediately preceded by a `crash` event, and a `crash` event may be immediately followed only by a `recover` event, or by no events at all.

A *history* of a process is a sequence of events as they occur in that process, in which a `crash` event is not followed by any other event. An *execution (run)* is a collection of histories, one for each process, in which there is a mapping from each `receive` event to a corresponding `send` event. In this paper we consider only executions in which there are no causal cycles [CL85, Lam78].

An execution σ' is a *sub-execution* of another execution σ if both include histories of the same set of processes, and the history of each process p_i in σ' is a prefix of p_i's history in σ. Given an execution σ and a sub-execution σ' of σ, the collection of history suffixes obtained by eliminating the history of each process in σ' from its corresponding history in σ is an *extension* of σ'. We denote this extension by $\sigma \setminus \sigma'$. An execution or a sub-execution is *infinite* if the history it contains for every process is either infinite or ends with a `crash` event.

2.2 The Failure Model

The underlying communication network provides datagram message delivery. There is no known bound on message transmission time, hence the system is asynchronous. Processes fail by crashing, and crashed processes may later recover. Live processes are considered *correct*, crashed processes are *faulty*. In protocols that explicitly mention the use of stable storage (e.g, the protocol in Chapter 6), recovered processes come up with their stable storage intact. Communication links may fail and recover. Malicious failures are not considered; messages are neither corrupted nor spontaneously generated by the network, as stated in the following property:

Property 2.2.1 (Message Integrity) *For any message m delivered by a process p, there is a preceding send event of m at some process q.*

Definitions

The *causal* partial order [Lam78] is defined as the transitive closure of: $m \overset{cause}{\longrightarrow} m'$ if $\text{receive}_q(m) \rightarrow \text{send}_q(m')$ or if $\text{send}_q(m) \rightarrow \text{send}_q(m')$.

Let σ be an infinite execution, σ' a sub-execution of σ, σ'' a sub-execution of σ', and let τ be the extension $\sigma' \setminus \sigma''$. Note that if σ' above is infinite, then $\sigma = \sigma'$. We use the following definitions:

alive Process p is *alive* in τ if p does not crash in τ and it incurs the same number of crashes and recoveries in σ'.

connected Processes p and q are *connected* in τ if p and q are alive in τ, p receives in σ every message that was sent from q to p in τ,[1] and vice versa.

A set of processes P is *connected* in τ, if for every two processes $p, q \in P$, p and q are connected in τ.

If σ' is infinite, p and q are (P is) *permanently connected* in σ.

detached Processes p and q are *detached* in τ if p does not receive (in σ) any message that q sends in τ and vice versa.[2]

A set of processes P is *detached* from a set of processes Q in τ if for every process $p \in P$ and every process $q \in Q$, p and q are detached in τ. Note that the definitions of detached and connected do not complement each other.

If σ' is infinite, p and q (P and Q) are *permanently detached* in σ.

connected component [3] A set of processes P is a *connected component* in τ, if P is *connected* in τ, and P is detached from $N \setminus P$ in τ.

If P has k members, it is also called a *k-connected component* in τ.

If σ' is infinite, P is a *permanently connected component* in σ.

A permanently connected component defines a *stable* situation in which members of the component are alive and can exchange messages among themselves, but cannot receive any message from processes outside the component. Note that, although messages are guaranteed to be delivered within a permanently connected component, there is no bound on the latency of these messages.

2.3 Failure Detectors

Failure detectors are useful abstractions for specifying services and protocols in a distributed environment prone to failures. Failure detectors provide a clear analysis of the effects of failures on the solvability of certain problems in distributed environments.

Chandra and Toueg [CT96] defined classes of *distributed* failure detectors: Each process has access to a local *failure detector module* which maintains a list of the processes that it currently

[1]Since network latency is not zero, and processes continuously send messages, requiring that the message will arrive in τ would be too strong.

[2]Note, in particular, that a crashed process is detached from any other process.

[3]A **component** is sometimes called a **partition**. In our terminology, a partition splits the network into several components.

suspects to have crashed. Failure detectors are categorized according to their degree of *completeness*, i.e., their success in detecting failures, and their *accuracy*, i.e., their ability to avoid false suspicions. The failure detectors definitions presented in [CT96] are described in the fail-stop model, which is limited to crash failures only. Below, we extend these definitions to the partitionable failure model.

The most intuitive way to extend failure detector definitions to a partitionable environment is to detect *detached* processes as well as faulty ones. We adapt Chandra and Toueg's definitions of strong completeness and accuracy to detect detached processes as follows:

Strong completeness If p and q are permanently detached from each other and p is alive then p eventually permanently suspects q (i.e., there is a time after which p suspects q forever).

Strong accuracy If p and q are permanently connected then p does not suspect q.

Eventual strong accuracy If p and q are permanently connected then there is a time after which p does not suspect q.

Using these properties, we now define perfect and eventually perfect failure detectors:

Perfect failure detector A perfect failure detector is a failure detector which fulfills the strong completeness and strong accuracy properties.

Eventual perfect failure detector An eventually perfect failure detector is a failure detector which fulfills the strong completeness and eventual strong accuracy properties.

Is it important to note that in environments in which messages are never lost, non-crashed processes are always connected. Thus, in such environments, our definitions are compatible with those proposed by Chandra and Toueg in [CT96].

Definitions of failure detectors that detect detached processes in partitionable environments were also suggested by Babaoğlu *et al.* in [BDM95]. In [BDM95], eventual symmetry of connected processes (i.e., if p is not connected to q then q is eventually not connected to p) is a requirement of the model, while in our approach, if communication does not preserve eventual symmetry then there are no restrictions on the failure detector's behavior. Like [BDM95], we focus on eventual perfect failure detectors and use these reachability detectors to analyze the solvability of a partitionable membership service that eventually stabilizes in the partitionable failure model.

Other possible detection properties, such as weak completeness, are meaningless w.r.t. detecting detached processes; Babaoğlu *et al.* [BDM95] observe that the resulting eventual weak and eventual strong reachability detector classes are not equivalent. Furthermore, neither of them is strong enough for solving the partitionable membership problem defined in [BDM95].

Chapter 3

Group Communication Systems and Specifications*

The service presented in this thesis follows the group communication paradigm. This chapter provides general background about this paradigm: Section 3.1 describes typical services of group communication systems (GCSs) and Section 3.2 discusses contemporary research in this area. Section 3.3 addresses the challenge of formally specifying the guarantees of group communication systems.

3.1 Background: Typical Group Communication Services

Group communication is a powerful paradigm for the development of fault-tolerant distributed applications and for CSCW groupware and multi-media applications. GCSs introduce the notion of group abstraction that allows processes to be easily arranged into multicast groups. A multicast group is identified by the logical name assigned to it when the group is created. Each message targeted to the group's logical name is guaranteed to be delivered to all the currently connected and operational group's members. This allows to handle a set of processes as a single logical connection. Furthermore, processes may dynamically join or leave these groups.

Some of the leading GCSs today are: Consul [MPS91b], Ensemble [HvR96], Horus [vRHB94], ISIS [BvR94, BSS91], Newtop [EMS95], Phoenix [MFSW95], Relacs [BDGB94], RMP [WMK95], Totem [AMMS$^+$95, MMSA$^+$96] and Transis [ADKM92b].

GCSs typically support reliable multicast and membership services. The task of the *membership service* is to maintain a listing of the currently active and connected processes in each group and to deliver this information to the application whenever it changes. The output of the membership service is called a *view*. The reliable multicast services deliver messages to the current view members.

*This chapter is based on the introduction to a survey by Vitenberg, Keidar, Chockler and Dolev [VKCD98].

A membership service may either be *primary component*[1] or *partitionable*; in a partitionable membership several disjoint network components may co-exist while in a primary component membership, only members of one connected component are considered alive. The first and most well known group membership service was the primary component membership of ISIS [BvR94]. It was followed by many more primary component membership services, e.g., [MS94, MPS91b, RV92]. The first partitionable membership service was introduced as part of the Transis [ADKM92b, DM96, ADKM92a] group communication system. Later, numerous new GCSs featuring a partitionable membership have emerged, e.g., Totem [AMMS$^+$95, MMSA$^+$96], Horus [vRBM96], RMP [WMK95], Newtop [EMS95] and RELACS [BDGB94].

Typically, GCSs provide the application builder with various types of efficient reliable multicast services. For example, the *causal* multicast service guarantees that the reply to a message is never delivered before the message. The *totally ordered* multicast service extends the *causal* service in such a way that all messages are delivered in the same order at all targets.

A GCS usually runs in an environment in which processes and communication links can fail, and in which messages may be lost or arbitrarily delayed. In such environments, the GCS simulates a "benign" world in which message delivery is reliable within the set of reachable (live and connected) processes. Furthermore, several GCSs provide semantic models such as *Virtual Synchrony* [BJ87], *Strong Virtual Synchrony* [FvR95] and *Extended Virtual Synchrony* [MAMSA94]. Such models define relationships between view changes and message delivery which enable the application to derive some useful information regarding which processes delivered the message (as explained in [VKCD98, ACDV97]).

3.2 Modularity: The New Trend in GCS

Experience with group communication systems and reliable distributed applications has shown that there are no "right" system semantics for all applications [Bir96]: different GCSs are tailored to different applications, which require different semantics and *qualities of service (QoS)*.

The Horus [vRBM96] system tackled this problem with a new paradigm: modularity. Horus and its successor Ensemble [HvR96] are flexible GCSs comprised of independent protocol *layers* that implement different service levels and semantics. This approach allows the application builder to tailor a GCS to his needs, treating protocol layers as building blocks.

Modular design has another important advantage: It is possible to separately reason about the

[1]A *primary component* was originally called a *primary partition*.

guarantees of each layer and the correctness of its implementation. Recently, the I/O automata formalism was exploited for specification and reasoning about GCSs [FLS97, Cho97, DPFLS98, CHD98]. The modular "layered" design nicely maps into compositions of I/O automata. This approach uncovers the subtleties of the interaction between the GCS and its applications, as well as among the layers of the GCS.

Another benefit of modularity is its flexibility to incorporate a variety of QoS options. Recently, several emerging projects addressed the challenge of incorporating QoS communication into the framework of group communication. For example, the MMTS [CHKD96] extends the Transis [ADKM92b, DM96] GCS by providing a framework for *synchronization* of messages with different QoS requirements; Maestro [BFHR98] extends the Ensemble [HvR96] group communication system by coordinating several protocol stacks with different QoS guarantees and the Collaborative Computing Transport Layer (CCTL) [RCHS97] implements similar concepts, geared towards distributed collaborative multimedia applications.

3.3 On the Formal Specifications of Group Communication

In this section we discuss the difficulties one encounters when trying to formally specify meaningful group communication systems.

3.3.1 The Impossible

Group communication systems typically run in asynchronous failure prone environments. Unfortunately, in such environments, agreement problems that resemble the services provided by GCSs are not solvable. An example of such a problem is Terminating Reliable Broadcast (TRB) [HT93], which requires non-faulty processes to either deliver every message sent or to declare the sender as faulty. Ideally, group multicast should resemble TRB in that it should require non-faulty processes to either deliver every message, or to deliver a view change that excludes the sender. Hence, an ideal group multicast service is also impossible to implement.

The issue of solvability of the group membership problem was recently the subject of extensive research. Ideally, we would like a membership service to eventually reflect the actual network connectivity. For example, consider the case that the network stabilizes so that a group G of processes remains permanently alive and connected, and all other processes have crashed. If such stabilization occurs, we would like the membership service to eventually report G as the current view and then report of no further view changes.

Unfortunately, this desired membership service is impossible to achieve. To see this, note that in the asynchronous fail-stop model, where communication links are reliable and crashed processes never recover, such network stabilization always eventually occurs. Therefore, when restricted to the fail-stop model, a desired membership service is as strong as an *eventual perfect failure detector* [CT96], which is not implementable in asynchronous fail-stop environments. In fact, it has been proven by [CHTCB96] that even a very weak version of the group membership problem is not solvable in asynchronous environments.

3.3.2 The "Best Effort" Principle

Practical systems cannot do the impossible, they can only make their "best-effort". This concept is illustrated by the following example: No system builder can guarantee that his group membership service will always correctly reflect the network situation. A powerful adversary that fully controls the communication can force every deterministic membership algorithm to be incorrect or to constantly change its mind. However, existing group communication systems make a "best-effort" attempt to reflect the network situation as much as possible, and indeed succeed most of the time. Note that the group communication systems we are concerned with are not intended for critical (real-time) applications; they run in environments in which such applications can not be realized. The usefulness of these systems stems from the fact that real networks rarely behave like vicious adversaries.

Many formal specifications of group communication systems do not capture this notion of "best effort". This results in specifications that can in fact be implemented by algorithms weaker than the actual implementations (or even by trivial algorithms) [ACBMT95]. However, since the "best effort" principle is an important consideration of system builders, actual systems provide more than their specifications require. For example, the Internet Protocol (IP) [Pos81] is an unreliable datagram protocol that does not guarantee to deliver any message. Yet few would argue that IP is useless.

3.3.3 Circumventing the Impossibility Result

Specifications of group communication were made non-trivial using a variety of techniques. The first attempts at non-trivial membership specifications [DMS94] ruled out only those classes of trivial algorithms which, despite changes in the actual network situation, might at some point cease reporting view changes. These attempts were criticized as too trivial in [ACBMT95].

Later specifications explicitly linked the behavior of the GCS to the output of an external failure detector module. The output of a failure detector is a list of suspects, i.e., processes which are suspected to be faulty. For example, [FvR95, DMS95] require that a process not be removed from the view unless it is a suspect.

Other specifications [MS94, BDM95, VKCD98] take this approach one step further, and exploit the notion of eventual perfect failure detectors (defined in Section 2.3). The specifications in [MS94, BDM95] guarantee that if the external failure detector is an eventual perfect one then the membership service will at some point begin to correctly reflect the network situation. We follow this approach in this thesis. Note that the eventual perfect failure detector is used here as an analysis tool to identify external conditions under which the membership service behaves correctly.

Another approach [FLS97] is to guarantee correct behavior of the GCS at periods during which the underlying network is stable and timely. The specifications of [FLS97] are stronger than the failure detector based ones in that they guarantee the timeliness of the service and not just eventual termination. Of course, such guarantees can only be made when network message delivery and process scheduling are timely. The specifications are parameterized by timeouts suited for the underlying networks.

Part II

Highly Available Groupware Services

Chapter 4

The Service Design*

This chapter presents a general framework for the construction of groupware and *computer supported cooperative work (CSCW)* applications. Examples of such applications include: multi-media and desktop conferencing, distance learning, interactive games and simulations, and collaborative computing.

The suggested framework integrates a comprehensive set of services which supports sharing of a variety of applications among dynamically changing groups of users. The services are fault tolerant and scalable, and are therefore appropriate for multi-processor failure prone networks such as the Internet. The services exploit the group communication paradigm for dynamic discussion groups, and for keeping track of the dynamically changing set of participants.

The main services are over-viewed in this chapter, and are presented in more detail in the next chapters.

4.1 The Service Architecture

This thesis presents a general framework for the construction of highly available distributed applications such as replicated servers, groupware and *computer supported cooperative work (CSCW)* applications. Examples of CSCW applications include: multi-media and desktop conferencing, distance learning, interactive games, and simulations. The service design follows the group communication paradigm.

Unlike classical group communication systems, our design separates the membership services from the multicast communication substrate. The membership is implemented as a separate server (daemon) on each machine that interacts with the multicast communication substrate and with the application. This separation makes the communication services more efficient since most of the time the membership does not change. This design also facilitates using the same membership services for a variety of quality of service (QoS) communication options [BFHR98, RCHS97, CHKD96]. Decomposing the service into separate modules also makes it easier to reason about, i.e., formally

*This chapter is based on a paper by Anker, Chockler, Dolev and Keidar [ACDK97].

specify the service guarantees and assumptions, and prove correctness. An overview of the membership services is described in Section 4.3. Chapter 5 describes the membership services in detail.

The service exploits a novel concept: a *multi-media multicast transport service (MMTS)* [CHKD96], that supports multiple QoS group communication options[1]. This makes the services inherently fault tolerant, and allows the application builder to define the tradeoffs between the level of synchronization/reliability and the timeliness of message delivery. The MMTS is described in Section 4.2.

In addition to the multicast communication service and the membership service, we provide application builders with session level services. These services relieve the application builder of the need to explicitly deal with the subtleties of changes in the network situation. The session services exploit the strong group communication semantics in order to efficiently maintain consistency of objects in the face of failures. An overview of the session services is presented in Section 4.4.

The service architecture is depicted in Figure 4.1.

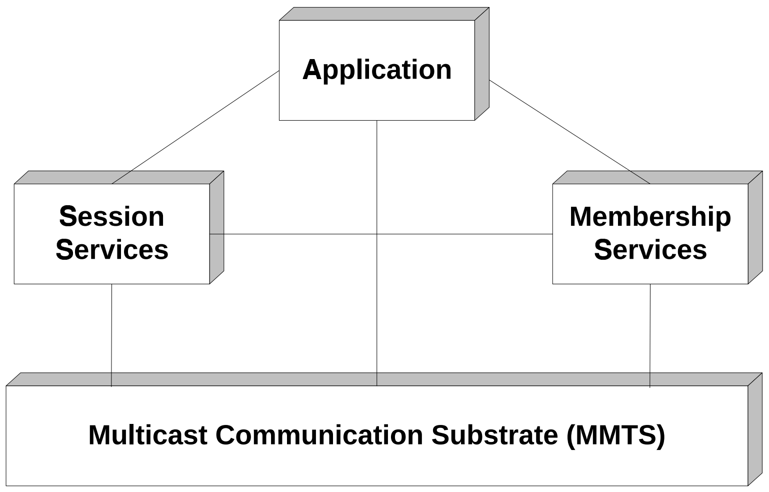


Figure 4.1: The system architecture.

4.2 Multimedia Multicast Transport Services (MMTS)

In [CHKD96] a novel concept is introduced: a *multi-media multicast transport service (MMTS)* that supports QoS group communication. The MMTS provides a framework for *synchronization* of messages with different QoS requirements.

The MMTS concept is particularly beneficial for applications that integrate services with a

[1]In [CHKD96] the membership services are regarded as part of the MMTS. Here, we follow the approach taken by Maestro [BFH97], which separates the group multicast services from the membership services.

variety of QoS needs. For example, multi-media and desktop conferencing systems require *Quality of Service (QoS)* communication for video transmission. Nonetheless, such applications are concerned with more than just transmitting a stream of video: they need to exchange messages for connection establishment, dynamic group management, and *negotiation and re-negotiation* of *Quality of Service (QoS)* [RR96]. Furthermore, it is desirable to make such systems fault tolerant.

Recently, several emerging projects addressed the challenge of incorporating QoS communication into the framework of group communication. Maestro [BFHR98] extends the Ensemble [HvR96] group communication system by coordinating several protocol stacks with different QoS guarantees. The Collaborative Computing Transport Layer (CCTL) [RCHS97] implements similar concepts, geared towards distributed collaborative multimedia applications. Both systems implement and elaborate the concept of MMTS.

The VIC [MJ95] video conferencing tool over the MBone[2] is a flexible framework for packet video. This approach uses a *conference bus* for broadcasting the various media in a conference session (e.g., whiteboard media, audio, and video). In the VIC architecture, the conference bus may be used along with a *coordination tool.* The MMTS can be viewed as integrating both the conference bus and the coordination tool.

The MMTS concept is flexible, it can exploit various underlying communication protocols and technologies, e.g., RSVP [ZDE+93], ST-II [Top90] and ATM QoS. Furthermore, it modularly supports integration of new QoS options, e.g., the cyclic UDP QoS [Smi94] that was implemented as a protocol layer in the Horus system [VvR94].

One of the important challenges that need to be addressed when using groupware toolkits for a multimedia application is how to combine services with strong semantics with the QoS required by the multimedia application.

The MMTS allows the user to provide optional *synchronization barriers* among streams of messages of different QoS types. Synchronization barriers are implemented using reliable messages. Using synchronization barriers, the user may enforce order semantics w.r.t. messages of different streams. These barriers may delay "faster" messages until the arrival of "slower" messages that they depend on. For soft real-time applications, that can tolerate some bounded delay, a best-effort synchronization mechanism is provided. The best-effort service delays the message delivery for some pre-defined timeout in order to try to synchronize the different channels used by the application. After this timeout, the message may be either discarded or delivered in spite of the

[2]Information about the MBone can be found in http://www.best.com/ prince/techinfo/mbone.html.

lack of synchronization, according to the application's specification.

This concept generalizes the Δ-*Causal* communication mode [BMR96, Yav92]. In this communication mode, messages may be lost. Each message has a *lifetime*, Δ, after which its data is no longer meaningful, hence the message may be discarded.

Another example of best-effort semantics is the cyclic UDP [Smi94] *prioritized* best-effort message recovery mechanism. Cyclic UDP allows the user to specify priorities for messages. Messages with a higher priority are recovered before messages with a lower priority. Message recovery attempts are stopped after a certain timeout period. Cyclic UDP may be incorporated in the MMTS, (as described in [CHKD96]), by recovering lost messages only until the synchronization barrier message is delivered.

4.3 Advanced Group Membership Services and Policies

We provide a hierarchical group membership service with support for group policies. These services are valuable building blocks for conferencing applications and interactive games. The membership service is implemented as a separate process (daemon).

4.3.1 Group Membership Services

The basic membership service is based on the CONGRESS CONnnection-oriented Group-address RESolution Service [ABDL97, ABDL96], which is designed for ATM networks, but may be exploited in other networks as well. CONGRESS supports two types of services: *address resolution*, which is a single query about the group membership, and *incremental updates*, which provides the user with updates every time the group membership changes.

CONGRESS provides basic efficient group resolution services for performance driven applications. It does not impose agreement on the order of membership changes, thus different members may incur the same membership changes in different orders. Furthermore, CONGRESS does not deal with message transmission, and in particular, does not impose any semantics on message ordering w.r.t. to membership changes.

Our membership service allows applications that require consistency to agree upon the order of membership changes, and thus incur membership changes in the same order. This is done by using CONGRESS incremental updates in conjunction with a one round agreement protocol, as described in Chapter 5. The agreement protocol is run only for groups that explicitly request this service.

Group communication systems usually provide strong semantics of message ordering w.r.t. membership changes, e.g., virtual synchrony [BJ87, FvR95, MAMSA94]. Virtual synchrony requires synchronization among the applications and the membership service. This service is costly: it incurs a delay period in which messages may not be transmitted [FvR95]. This synchronization greatly facilitates the design of applications that require consistency (e.g., applications with shared data [BJ87, ABCD96, KD96, ADMSM94, SM98]), but is too costly for applications that require real-time message delivery (e.g., video transmission).

We provide virtually synchronous communication for groups and message types that explicitly request this service. The implementation of virtual synchrony is described in more detail in Chapter 5.

4.3.2 Hierarchical directory services

An important innovation of our membership service is the support for hierarchical directory services. We maintain a hierarchy of groups: A group may be a *sub-group* of a parent group. A parent group may contain a number of sub-groups. The listing of sub-groups and their memberships are available only to the members of the parent group. This concept is useful for applications containing a number of logically related groups, e.g., a conferencing applications with several discussion groups.

In order to supports the notion of a *secure multicast group* [RKBvR94, RHDB98, RD97], the access to the group hierarchy is regulated by an authentication server. Only authorized members are allowed to join secure groups. The hierarchical directory services allows secure groups to be hidden from unauthorized parties, by coupling them as sub-groups of the same secure parent group.

The membership service may maintain two membership sets for each process group: *active members* who may provide input in the group, and *passive members* who receive messages sent to the group but cannot send messages to the group.

4.3.3 Group Policies

The membership service may also allow users to determine *policies* regarding the membership and nature of communication in a group. The policies are declared when the group is created. If no policy is declared, then the policies are inherited from the parent group. There are two basic types of policies: *membership policies* and *run-time policies.*

Membership policies restrict the ability of processes to become members of the group. Restrictions may be imposed on the number of members in a group, and also on the properties of the members. For example, a conference over the Internet may allow only two members from each country to participate in the discussion. If due to a membership policy, a user's *join* request may currently not be fulfilled, we allow the user to *block* until the *join* will become possible. Membership policies are enforced by the membership service.

Run-time policies are specified at group creation time, and are enforced at run-time. Run time policies may define, for example, the number of users that may concurrently provide input in a group, and who is responsible for dispensing the right of speech. The floor control mechanism (described in Section 4.4.2) enforces such policies.

Run-time policies are used in conjunction with membership policies. An example application that exploits both types of policies is an interactive chess server. The chess server allows two players to play in each game (actively join the "players" group), and allows other users to watch the game and exchange comments (passively join the "players" group, and actively join the "voyeur" group). Permission to play a game is granted according to the player's rank. Each of the two players may make a move only when it is his turn.

4.4 The Session Services

The service architecture allows supporting a wide range of session level services geared towards the needs of typical classes of distributed applications. Among them are tools for coordination and floor control in conferencing systems, consistent object replication, security, *etc*. In this thesis we elaborate on consistent replication services, which are over-viewed in Section 4.4.1. In Section 4.4.2 we discuss coordination services and floor control.

4.4.1 Support for replication

Numerous distributed applications use replication in order to increase their availability and reliability. This raises the need for a service that would preserve replicas in a consistent state despite network and machine failures: When the network partitions into several disjoint components, the states of disconnected replicas may diverge. When processes reconnect, all the processes should be brought to a common state.

We distinguish between two levels of consistency services: *short-term* and *long-term*. The *short-term consistency* service guarantees to preserve consistency within a group of connected processes. When a partition is mended, the states of previously disconnected replicas are unified using a state transfer protocol. The protocol in [ACDV97] exploits group communication for efficient implementation of state transfer.

The *long-term consistency* service guarantees a 1-copy serializable history of object updates.

That is, for every execution σ of the replicated data service there is a sequence τ of (non-distributed) object updates such that σ produces the same output and has the same effect on the data as executing τ on a single copy of the object (cf. [BHG87]). This is achieved by prohibiting arbitrary updates of the object in disjoint network components, often, only the members of a *primary component* may update the object. The long-term consistency service is described in Chapter 6. We also provide a dynamic voting-based *primary component service* that notify processes when they are members of the current primary component which is described in Chapter 7.

4.4.2 Coordination and Floor Control

Different groups may impose different *run-time policies* on the eligibility of members to provide input of various types (e.g., video, audio, text) in a group. The policy is defined when the group is created. The floor control mechanism enforces this policy. An example policy may allow all the participants to type text concurrently in a text chat, and yet allow only one member to update a shared file at a given time. Another possible policy may designate a group of parties as the conference managers which are responsible for dispensing the right of speech. Ordinary members are allowed to speak only when they obtain permission to speak.

The floor control mechanism manages the dynamic switching of the right to produce input among multiple conference parties. This service is particularly useful in distance learning applications, in which students are typically not allowed to intervene when the teaching is in progress. Nevertheless, the teacher may grant students permission to ask questions at the end of a topic presentation.

The floor control supports the *token* abstraction to designate a party (or group of parties) that are currently allowed to produce the input. The interface also allows parties to indicate their wish to obtain the token. If some participants were defined as conference managers they can pass the token among the ordinary parties at any given time.

If all the group members are equal in rights, they can freely compete for the token. The reliable totally ordered multicast service helps guarantee the uniqueness of the token holder. When the current token holder finishes its "monologue", he can explicitly pass the token to another party or return it to the system so that other parties can compete for it.

Chapter 5

Scalable Group Membership Services*

This chapter presents a new architecture for a scalable group membership service for wide area environments. This architecture provides two different service levels and their semantics, each geared to different applications with different needs. This chapter focuses on a novel scalable group membership algorithm, which provides virtually synchronous communication semantics.

The novelty of our design is in its client-server approach: In our design, membership is not maintained by every process, but only by a few dedicated servers. Thus, membership maintenance induces very low overhead when membership changes are infrequent. Our design is inherently scalable and suitable for wide area networks. It allows lightweight clients to benefit from advanced membership services. Furthermore, our design supports the coexistence of full-fledged clients along with thin clients.

5.1 Introduction

Group communication [ACM96] is an important abstraction, widely used for distributed and communication-oriented applications. Such applications typically require the coordination of large and dynamic sets of processes at different sites. The group communication abstraction is essential for the modular design of groupware and other multi-user applications in such networks. The most important aspects of this abstraction are the maintenance of group membership and the semantics of interleaving membership change notifications within the flow of regular messages.

Different applications utilize group communication for different purposes, and hence require different semantics from the group membership service they utilize (as explained in [BFHR98, CHKD96, Bir96]). For example, video conferencing applications need a general knowledge of which peers are interested in joining the conference, in order to know where to multicast the video stream, and where to receive it from. Such applications do not require the synchronization of membership

*This chapter is based on a paper by Anker, Chockler Dolev and Keidar [ACDK98] and on work by Keidar, Sussman, Dolev and Marzullo [KSDM].

change notifications with regular messages.

On the other end of the spectrum, consistent data replication may greatly benefit from strong semantics [BJ87, ABCD96, KD96, FLS97, ADMSM94, SM98]. For example, some group communication systems provide virtual synchrony semantics, which synchronize membership notifications with regular messages and thus simulate a "benign" world in which message delivery is reliable within the set of live processes. This enables synchronization among applications, but is costly: it incurs a delay period in which messages may not be transmitted [FvR95]. Therefore, it is not appropriate for applications that require real-time message delivery (e.g., video transmission).

Computer Supported Cooperative Work (CSCW) [Rod91] groupware and multimedia applications involve different services that require different Qualities of Service (QoS) and different semantics from the group membership which they use, for example, an on-line conferencing application may incorporate multimedia multicast as well as coordination and sharing of consistent information (e.g., a shared white board).

Extensive research is currently being carried out to optimize scalable reliable multicast protocols in order to meet the demands of such applications [Car94, FJM$^+$95, PSK94, PSLB97]. Many of these applications make use of highly dynamic *multicast groups.* Such protocols often need to be complemented by a membership mechanism that maintains the dynamically changing set of members in each multicast group.

However, the design of a scalable membership service for WANs is a challenging task. Issues that need to be addressed include:

- Message latency tends to be large and highly unpredictable in a WAN, as compared to the relative consistency of message latency in a local-area network (LAN). This high latency works against algorithms in which processes repeatedly exchange messages in order to reach a decision.

- Failure detection in a WAN is usually less accurate than failure detection in a LAN. Inaccurate failure detection may cause a membership algorithm to change views frequently. This is costly as it can cause applications to engage in additional communication for re-synchronizing their shared state.

- There is no efficient support for the flooding of messages in a WAN, as opposed to a LAN. A group membership service supporting multiple groups in a WAN must take care not to flood the network.

This chapter describes a new architecture for construction of a scalable group membership service for wide area environments. The membership service provides two different service levels and semantics, each geared to different applications with different needs. In addition, our membership server provides advanced services such as a hierarchical directory of groups and secure groups.

The two different service semantics are geared towards different kinds of applications: the *CONnection-oriented Group-address RESolution Service (*CONGRESS*)* [ABDL97, Ank97] membership service provides simple semantics of membership approximation, and the *Membership Object-oriented Service for Heterogeneous Environments (*MOSHE*)* service, which extends CONGRESS, provides full virtual synchrony semantics. In this chapter we focus on implementing MOSHE atop of CONGRESS.

There are many different formulations of group membership services (some examples may be found in [VKCD98, DMS94, DMS95, BDM97]), and various definitions of virtual synchrony semantics (and variants such as strong virtual synchrony, extended virtual synchrony), e.g., [VKCD98, BJ87, FvR95, MAMSA94, FLS97]. MOSHE provides semantics which have been proven useful for several distributed applications [ABCD96, KD96, FLS97, ADMSM94, SM98]. In particular, the total ordering protocol presented in Chapter 6 exploits these semantics.

Numerous group membership protocols providing similar semantics were constructed (e.g., [CS95, AMMS$^+$95, MMSA$^+$96, EMS95, ADKM92a, MPS91a, MSMA91, DMS94, MS94, BDM97]). The novelty of MOSHE is in its client-server approach: The servers maintain the membership of clients in groups. The client-server design is a major challenge, since the protocol has to synchronize different entities. Our implementation focuses on minimizing the number of messages sent in order to achieve preciseness, without sacrificing efficiency.

In our design, membership is not maintained by every process, but only by a few dedicated servers. Thus, membership maintenance induces very low overhead when membership changes are infrequent, and the strong semantics required by some parts of the application induce no overhead for those parts which require weaker semantics.

The rest of this chapter is organized as follows: Section 5.2 describes the environment model and Section 5.3, the system architecture. The features of the basic membership service, CONGRESS, are described in Section 5.4. The guarantees of the virtually synchronous membership service, MOSHE, are specified in Section 5.5. Section 5.6 overviews the implementation of MOSHE. Section 5.7 compares MOSHE with other membership algorithms. Section 5.8 describes the advanced membership services.

5.2 The Environment Model

The membership service exploits an external *failure detector (FD)* module (as explained in Chapter 2). It is assumed that the failure detector fulfills *strong completeness*, i.e., it eventually suspects every process that has permanently crashed or disconnected. We do not discuss here how the failure detector is implemented. A framework for implementing a failure detector in a WAN is provided in [Vog96].

We further assume that the failure detector module operates in conjunction with the underlying communication, so that no messages are ever received from a suspected process, i.e., a `receive` event cannot occur for a message whose sender is in the receiver's suspect list.

We assume that the communication between pairs of processes preserves the FIFO order, as specified below:

5.2.1 Reliable FIFO Multicast Communication Channels

Our membership service is constructed atop a reliable FIFO multicast service, which fulfills the following properties:

Property 5.2.1 (Reliable FIFO Order) *If process p sends two messages to process q: m_1 and later m_2, and if q receives both messages, then these messages are received in the order in which they were sent.*

Furthermore, while process q does not suspect process p, q receives p's messages without gaps, i.e., if process p sends a message m' between m_1 and m_2, and between the receipt of m_1 and m_2, q does not suspect p, then q also receives m' between m_1 and m_2.

Property 5.2.2 (No Duplication) *Every message received by a process p is received only once by p.*

The reliable FIFO service preserves the Message Integrity property (Property 2.2.1) of the underlying communication links.

5.3 The System Architecture

Our membership service differs from those of other group communication systems in that it complies with the client-server paradigm (please see Figure 5.1). Processes that communicate with each other are *clients* of the membership service. The clients communicate with each other using reliable FIFO multicast channels[1] which allow them to multicast messages to all the members of a group.

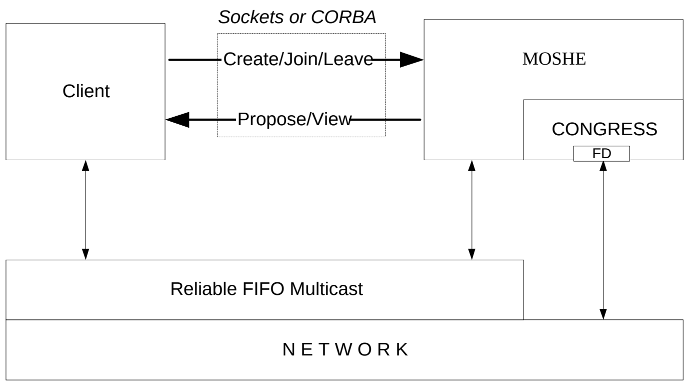


Figure 5.1: The membership service architecture.

The task of the membership service is to maintain a listing of the currently active and connected group members, and to deliver this information to clients in a consistent manner, when the membership changes. The changes in the group membership are reported in *views*. A client becomes a member of a group by *joining* the group, and stops being a member by *leaving* the group, or by crashing.

We currently support two options for client-server interaction: The first option is based on a reliable point-to-point FIFO service built directly atop the low-level socket interface. The second option utilizes the *Common Object Request Broker Architecture (CORBA)* [OMG98] which is the industrial standard for building client-server applications. Some of the advantages of using CORBA include: simplified object-oriented design, network transparency, client-server failure detection, and the possibility of using standard CORBA services (e.g., security, naming and event services).

Within the CORBA framework, objects (i.e., entities consisting of an interface and an implementation) are *registered* over a virtual *software bus*, called the *Object Request Broker (ORB)*. Whenever a CORBA application issues a request to a previously registered (possibly remote) object, the ORB locates the object and forwards the request to it. For more details, please see [ACDK98].

The membership server is designed according to the object-oriented paradigm and written in the Java programming language. The membership server consists of two objects: MOSHE and CONGRESS. The CONGRESS substratum accumulates the group membership and failure detection information and disseminates it among the membership servers. CONGRESS resides directly on top

[1]The clients may use a multimedia multicast transport service (MMTS) as described in Section 4.2. The MMTS provides a variety of QoS options, among them reliable FIFO multicast.

of a network layer (such as ATM or IP).

MOSHE extends CONGRESS to provide membership services with strong membership and message delivery semantics. Examples of such semantics include virtual synchrony and the ordered delivery of views. In addition, MOSHE provides some advanced services such as hierarchical and secure group services. In order to synchronize multicast message delivery with membership change events, MOSHE clients multicast synchronization messages via the MMTS.

Client requests are first processed by the MOSHE object. For each client request, this processing includes updating the group hierarchy (if necessary) as well as authorization and/or authentication for secure groups. Then, the request is delegated to the CONGRESS object for further dissemination among the other membership servers.

When a change in the membership of some group is reported by CONGRESS, MOSHE checks if the group requires strong semantics. If it doesn't, MOSHE immediately informs the group members of the new membership. Otherwise, MOSHE initiates an additional synchronization round at the end of which the view is reported.

5.4 The CONGRESS Basic Service

CONGRESS [ABDL97] is a protocol for *resolving* (i.e., mapping a multicast group name into a set of members identifiers) and maintaining the membership of multicast groups. CONGRESS operates over point-to-point connections, and is scalable to a WAN.

In order to be scalable and efficient, CONGRESS minimizes the network traffic required to maintain the dynamic group membership. The saving in network traffic is achieved by using a hierarchy of dedicated servers, which propagate necessary information about multicast groups to clients in the server's area. Furthermore, CONGRESS does not flood the WAN on every group membership change. This is achieved through careful maintenance of a distributed spanning tree for each of the multicast groups. A single membership change in a multicast group G incurs $O(|G|)$ protocol messages.

5.4.1 CONGRESS Services

The CONGRESS services are provided by an interface that consists of the following basic functions:

- **join**(G): Make the invoking client a member of group G.
- **leave**(G): Remove the invoking member from the membership in G.

- **resolve**(G): Request to resolve a multicast group name G into an approximated set of members identifiers.

A client may learn of the membership of a group from the following *membership notification events*:

- **resolve-reply** is a response to a **resolve** request. It consists of an approximated list of members.

- **Incremental membership notification (MN)** is a notification of a change in the group's membership, due to **join**, or **leave** events, or due to a change in the suspect list reported by the failure detector. A change in the suspect list can occur due to a suspected process crash or communication link failure, or due to recovery of either a process or a communication link.

  Incremental membership notifications report only the difference between the new membership and the one previously reported. For example, MN("join", a, G) denotes a membership notification that reflects the fact that process a has joined group G, either due to a **join** request or due to recovery of the communication link to a.

Definitions

We now introduce the following definitions:

- An incremental membership notification *reflects* one of the following events w.r.t. a process: **join**, **leave**, process crash, failure of the communication link to a process, false suspicion of a process, or a refutation of a false suspicion. In some cases, an incremental membership notification may reflect the outcome of a series of such events, e.g., if a process has disconnected, and later crashed and recovered and then reconnected, then the outcome of latter three events would be reflected in a single "join" notification. A **resolve-reply** *reflects* the final outcome of a series of events.

- The *membership of group G calculated by a client* is constructed by resolving a group name once, and subsequently applying the incremental membership notifications as they arrive. For example, if the **resolve-reply** received in response to the initial **resolve** request for group G was $\{b, c\}$, then applying the membership notification MN(a, G) yields the new membership: $\{a, b, c\}$.

- We say that process a is a *potential member* of group G if a has joined G (i.e., issued a join), and afterwards a has neither crashed nor issued a leave.

5.4.2 CONGRESS Guarantees

In this section, we describe the properties that CONGRESS guarantees w.r.t. the membership information it provides.

The first property guarantees that two membership notifications w.r.t. the process are received in the order of the events which they reflect. This guarantee is called *per-source chronological ordering of membership events.*

Property 5.4.1 (Per-Source Chronological Ordering of Membership Events) *Membership notification events reflecting events w.r.t. the same process occur in the same order as the events that they reflect.*

Note, however, that membership notification events w.r.t. different processes may occur in different orders at different processes. This is illustrated in Example 1 below. Agreement on the order of notifications would require running a synchronization round for each membership change. Such a synchronization round is performed by MOSHE for groups that require it, as explained in Section 5.6.

Example 1 *Assume that a and b are two members of a group G. Assume further that the membership at both of them is $\{a, b\}$. Now, assume that c and d join group G at approximately the same time. Assume also that c is topologically close to a and that d is topologically close to b. It is highly probable that a will receive* CONGRESS*' notification about c joining G before receiving notification about d joining G, and that b will receive the notifications in the reverse order (i.e., the notification about d will be received before the notification about c at b). This implies that a will calculate the membership of G first as $\{a, b\}$, then as $\{a, b, c\}$ and finally as $\{a, b, c, d\}$, whereas b will calculate the membership first as $\{a, b\}$, then as $\{a, b, d\}$ and only then as $\{a, b, c, d\}$.*

Since the network is asynchronous and protocol messages may be delayed, membership information at distinct CONGRESS servers may differ at any given time. It has been proven that it is impossible to make strong guarantees regarding the preciseness of the membership derived from notifications at instable time periods (please see discussion in Section 5.5.1). If, however, the network stabilizes, and no new membership events occur in group G, then eventually all the members of G will have a *precise* view of the membership in G. This is formulated in the following property:

Property 5.4.2 (Eventual Preciseness) *Let G be a group, and t_0 a point in time s.t. the set of potential members of G has not changed after time t_0. Furthermore, assume that after time t_0 all the potential members of G are in the same connected component, and do not suspect each other. Then there is a time $t_1 > t_0$ s.t. at time t_1 all the potential members of G have the same calculated membership, which consists of exactly the set of potential members of G.*

Consider, for instance, Example 1 above. There, the membership eventually becomes $\{a, b, c, d\}$ at all the processes.

Two points are worth noting about the above definition:

1. If there are live potential members of G in two disjoint network components, then we would like processes in each component to have a calculated membership consisting of the set of potential members of G in the local component. It is easy to see that this requirement is fulfilled by any protocol that fulfills Property 5.4.2.

2. For simplicity's sake, we required that stability would last forever. In practice, however, the following situation holds: Let t_1 be a time by which all the events that occurred before time t_0 have been reflected by membership notification events at all the processes. If the membership of G stabilizes for only a finite time interval $[t_0, t_2]$, s.t. $t_2 > t_1$ then all the potential members of G will also have the same calculated membership. This membership will consist of exactly the set of potential members of G during the interval $[t_1, t_2]$.

The stronger guarantees are formulated and proven in [Ank97].

5.5 Semantics of the VS Membership Services

Some applications require membership services with only weak semantics, and some require strong semantics. Applications that need to consistently maintain a replicated state (e.g., coherent cache), greatly benefit from virtually synchronous communication and membership semantics. The MOSHE membership service provides such semantics for groups that explicitly request this service. We call such groups *VS groups*. The protocol that implements these semantics is the MOSHE VS membership protocol. The VS membership service, together with the reliable FIFO multicast service comprise a *virtually synchronous communication service*, which fulfills the specifications stated below. These specifications are derived from the formal specifications presented in [VKCD98].

The membership protocol encapsulates membership notifications in views. A view v is a triple consisting of: the group name, denoted $v.G$; the group membership (i.e., the list of members), denoted $v.M$, and a view identifier, denoted $v.id$. We say that the view v occurs in the group $v.G$. We omit $v.G$ where it is obvious. The view identifier is taken from a partially ordered set (in our implementation, it is an integer). We say that a process is a member of view v if it is in $v.M$.

The key features of the membership provided by MOSHE for VS groups are:

- Agreement on the order of views.
- Synchronization of multicast messages with view reports (*virtual synchrony*).

We elaborate on these features below.

5.5.1 Agreement on Views

Agreement on the order of views allows processes that continue together to perceive changes of the membership in the same order. With the CONGRESS basic service, two processes that continue together may receive membership notifications in different order, as illustrated in Example 1 in Section 5.4. The membership service uses CONGRESS incremental membership notifications in conjunction with a one round agreement protocol, which allows processes that remain connected to receive views in the same order. The MOSHE VS membership protocol guarantees the following set of properties:

Property 5.5.1 (View Identifier Local Monotony) *Processes deliver views in a monotonously increasing order of view identifiers.*

In particular, Property 5.5.1 implies that the view identifier is locally unique, i.e., a process cannot deliver two views with the same view identifier. Note that the pair $\langle V.id, V.M\rangle$ serves as a globally unique view identifier.

Property 5.5.2 (Self Inclusion) *Processes deliver only views of which they are members.*

Property 5.5.3 (Agreement on Views) *Let G be a VS group, t_0 a point in time, and S a set of clients. Assume that from time t_0 onwards:*

1. *All the clients in S are potential members of G.*
2. *All the clients in S and their servers are in the same connected component C, and the topology of C does not change.*

3. Processes in C do not suspect each other, and every process which is not in C is suspected by every process in C[2]*.*

Then there is a time $t_1 > t_0$ s.t. all the processes in S incur the same sequence of views in G after time t_1. Furthermore, if the set of potential members of G in C is exactly S after time t_0, then all the clients in S eventually deliver the same view v, s.t. $v.M = S$ and do not deliver any further views in G.

As explained in Section 3.3, perfectly precise membership services are impossible to implement in truly asynchronous environments. Therefore, we have formulated Properties 5.4.2 (Eventual Preciseness) and 5.5.3 (Agreement on Views) to guarantee preciseness of the membership service only at stable periods in which the external failure detector module does not suspect correct and connected processes. If the network is highly unstable, or if failure detection is highly unreliable, then it is possible that the membership algorithm would never be precise. If, however, the failure detector is an *eventual perfect* one (cf. Chapter 2), and the communication stabilizes, then the membership service is guaranteed to eventually be precise, and furthermore, agreed upon.

It is important to note that although our non-triviality properties (Eventual Preciseness and Agreement on Views) are guaranteed to hold only in certain runs, the conditions on these runs are *external* to the algorithm implementation, and therefore a trivial or useless protocol cannot meet these guarantees.

As noted in Section 5.4, stability does not have to hold forever. Any protocol that fulfills Property 5.5.3 must begin to precisely reflect the network situation a finite time after the network stabilizes, even if stability will not last indefinitely. (A similar argument is made w.r.t. partial synchrony in [DLS88]). In this thesis we do not analyze the actual length of time required until the membership becomes precise. This can be done, as in [FLS97], by explicitly linking the guarantees of the membership protocol to pre-determined bounds on process scheduling times and delays at the underlying network instead of using the failure detector abstraction.

5.5.2 Strong Virtual Synchrony

Virtual synchrony involves synchronizing multicast communication with membership notifications. In this programming model, group multicast send and receive events occur within the context of views. We say that a multicast send (receive) event e in group G occurs at process p in view v if v was the latest view that p received in group G before e.

[2]This requirement is fulfilled if the failure detector is eventually perfect. Please see [CT96, DFKM96, DFKM97].

The MOSHE VS membership protocol preserves the following properties of the reliable FIFO communication service: Message Integrity (Property 2.2.1) and No Duplication (Property 5.2.2). In addition, the MOSHE VS membership protocol guarantees the set of properties specified below.

The following two properties are liveness properties:

Property 5.5.4 (Self Delivery) *A message sent by a process is eventually delivered by that process, unless the process suffers a crash failure or leaves the group.*

Property 5.5.5 (Termination of Delivery) *If a process p sends a message in a view v in G, and process q is in $v.M$, then one of the following holds:*

- *q delivers this message, or*
- *p eventually delivers a new view in G, or*
- *p crashes, or*
- *q leaves the group.*

The next property is part of the strong virtual synchrony model.

Property 5.5.6 (Synchronous Delivery) *Every message is delivered within the view in which it was sent.*

The synchronous delivery property can be relaxed in various ways, which are not in the scope of this chapter. Please see [FvR95, SM98, VKCD98].

In addition, MOSHE provides the Virtual Synchrony property for groups that require it. This property is perhaps the most well known property of GCSs, to the extent that it engendered the whole Virtual Synchrony model.

Property 5.5.7 (Virtual Synchrony) *Any two processes undergoing the same two consecutive views in a group G deliver the same set of messages in G within the former view.*

The MOSHE VS guarantees are very similar to the semantic of [VKCD98], and to the *Extended Virtual Synchrony* semantic described in [MAMSA94]. We have removed the causal order, total order, and safe delivery properties from these semantics, in the belief that these are optional properties that may be built on top of this service (e.g., see [CHD98, Cho97]). There are other group membership specifications in the literature, such as [FvR95], [BDM97] and [DMS95]. These differ in various details, but have much in common with the semantic that is used here.

The total ordering protocol in Chapter 6 exploits the MOSHE VS guarantees.

5.6 Implementation of the MOSHE VS Membership Protocol

In this section we describe an overview of the implementation of the MOSHE VS membership protocol. The MOSHE membership algorithm maintains the list of members in each group. The algorithm is invoked in response to requests from clients to join or leave groups, and in response to network events. MOSHE uses CONGRESS incremental updates to propagate knowledge of network changes and of membership events.

5.6.1 Message Flow in MOSHE

When a client join/leave request or a client failure report is handled by a MOSHE server, the server invokes a corresponding join/leave request in CONGRESS for further dissemination among the other membership servers. Once CONGRESS generates a membership notification reflecting this event, MOSHE starts a synchronization round, in order to achieve virtual synchrony and agreement on the view.

The membership server computes a proposed view, as explained in Section 5.6.3 below, and emits *proposals* reflecting it to all of its clients which are members of the group. The clients echo the proposals in *flush* messages, which acknowledge their participation in this view. The flush messages are propagated to all the membership servers. The clients also multicast the flush messages to the other members of the group in order to synchronize view delivery with the multicast message flow, as explained in Section 5.6.2 below.

After the proposals are emitted, we say that the view is *pending* until flush messages arrive from all of its members. If a new incremental membership notification arrives while the view is pending, then new proposals are sent only to those newly joined clients to whom proposals for this view were not previously sent.

Once the MOSHE server receives flush messages from all the members of the pending view, it emits a view message to all of its clients (which are members of the group). The clients synchronize the view with the flush messages, as explained in Section 5.6.2 below.

Example 2 *In Figure 5.2 we illustrate the events that occur in the protocol when process a is joining group G. Initially, the membership of group G is $\{b, c\}$. The protocol is invoked when process a issues its join request.*

The join request (denoted by (1) Join(a, G) in Figure 5.2), is propagated using CONGRESS. *All the* MOSHE *servers learn of it via the* CONGRESS *membership notification, denoted by (2) MN(a, G).*

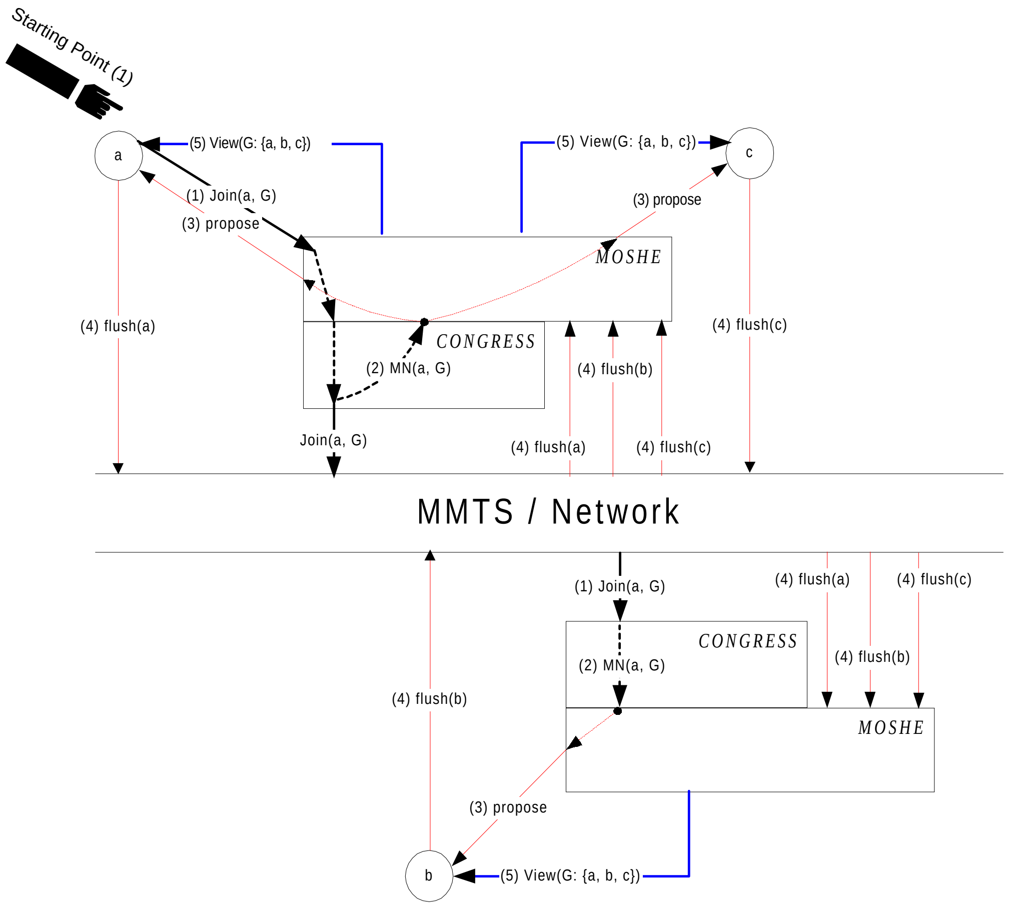


Figure 5.2: Events occurring in MOSHE when process a is joining VS group G.

Upon receiving this event, the MOSHE *servers generate proposals for a new membership to their clients. In Figure 5.2, these are denoted by (3) propose. The clients respond by multicasting (4) flush messages to the servers and clients.*

Once the MOSHE *membership server receives flush messages from all the members of the new view, it issues a view message, (denoted by (5) View(G:* $\{a, b, c\}$*)).*

5.6.2 Supporting Virtual Synchrony

Virtual synchrony requires synchronization among the clients: In order to fulfill Properties 5.5.6 (Synchronous Delivery) and 5.5.7 (Virtual Synchrony) the clients have to synchronize their multicast messages with membership events. Such synchronization necessarily incurs a delay period in which messages may not be transmitted [FvR95].

The synchronization mechanism is based on the flush messages described above. The purpose of flush messages is to synchronize views with the multicast message flow. Therefore, flush messages are multicast via the MMTS, and serve as *place holders* which denote where (in the message flow) the previous view ends. After sending a flush message, the clients do not send any new messages until the new view is reported.

Once the MOSHE membership server receives flush messages from all the members of the new view, it issues a view message. The view also contains a view identifier as explained in Section 5.6.3 below. The clients synchronize the view with the flush messages: Messages that were sent before the flush are delivered in the previous view. Recall that we assume that the MMTS provides FIFO multicast services; hence, messages sent before the flush message are delivered before the flush message. This way, every message is delivered in the view in which it was sent.

5.6.3 Computing the Proposed View

A proposed view (for a group G) consists of the proposed set of members, a proposed view identifier, and a proposed set of suspects (or leavers).

The membership server computes the initial members and suspects sets of the proposed view by applying the CONGRESS incremental membership notification to the membership of the current view. When a CONGRESS incremental membership notification reports that a process is leaving a group[3], the process is removed from the members set and added to the suspects set. In case a join notification arrives, the joiner is added to the members set and the suspects set is empty.

Consider Example 2 above: There, the membership of group G was $\{b, c\}$ and the incremental notification reported that a had joined. The proposed set of members is therefore $\{a, b, c\}$, and the proposed set of suspects is empty.

The proposed view identifier is computed by incrementing the latest known view identifier by one. For example, if the latest view was $\langle G, \{b, c\}, 3\rangle$, then the new proposed view is $\langle\{a, b, c\}, \{\}, 4\rangle$. The proposed view is sent in the proposal, and echoed in the corresponding flush message.

If a new incremental notification arrives while the view is pending, then the pending view is re-computed by aggregating the incremental notification to the pending view. When a join notification arrives, the joiner is added to the members set unless it is already in the suspects set[4]. New proposals are then sent only to those newly joined clients to whom proposals were not

[3]This can occur either because the process crashed or because it has requested to leave the group.

[4]If the joiner is in the suspects set, the notification is buffered to be handled after the current pending view will be delivered.

previously sent. This is illustrated by the following example:

Example 3 *Consider Example 1 where two processes, c and d, concurrently try to join group G. Assume that clients a and c are served by the membership server M_1, and b and d by M_2, and assume that at both servers the current view of group G is $\langle G, \{a, b\}, 3\rangle$. Consider the case in which membership server M_1 first receives the notification that c is joining, and then issues proposals for the view $\langle G, \{a, b, c\}, \{\}, 4\rangle$ to a and c. At the same time, server M_2 receives the notification that d is joining, and then issues proposals for the view $\langle G, \{a, b, d\}, \{\}, 4\rangle$ to b and d.*

In the next stage, M_1 receives the notification that d is joining, and aggregates it to the pending view, which now becomes $\langle G, \{a, b, c, d\}, \{\}, 4\rangle$. Server M_1 checks if it has to send new proposals: Since the only new member (d) is not a client of M_1, no new proposals have to be emitted. Similarly, the aggregated pending view at server M_2 becomes $\langle G, \{a, b, c, d\}, \{\}, 4\rangle$, and M_2 does not emit new proposals.

Similarly, if a leave notification is received during the synchronization round, the server removes the leaving client from the members set of the pending view, adds it to the suspects set, and no longer waits for a flush message from this client. In order to prevent blocking, the servers should eventually either receive a flush message from every member of the view, or receive a notification that the member has failed. This is guaranteed to happen since we assume that the failure detector fulfills *strong completeness*.

Note that it is possible for a flush message that reflects a CONGRESS membership notification to arrive before (or even without) the membership notification[5]. In such cases, the incremental change reflected in the flush message is also aggregated into the pending view: The suspects set becomes the union of the suspect sets, and the new members set consists of the members of the union of the members sets, except for those who are in the suspects set.

Once flush messages arrive from all the members of the pending view, the server sends the new view to the clients. The view membership is the members set of the pending view, and the view identifier is chosen to be the maximum of the proposed view identifiers among the collected flush messages.

Consider Example 3 above: There, the servers wait until they receive flush messages from all four members before they send the new view to their clients. Since all the proposals contain the proposed view identifier 4, this is the view identifier for the new view. Thus, two separate CONGRESS incremental membership notifications are aggregated and reflected in one view: $\langle G, \{a, b, c, d\}, 4\rangle$.

[5]This can occur, for example, in case of a false suspicion at some of the processes.

In case two previously disconnected servers become connected, it is possible that they may send proposals for the same view with different view identifiers. This is illustrated in the example below.

Example 4 *There are two membership servers, M_1 and M_2, which are disconnected due to a network failure. At M_1 the view of group G is $\langle\{a, b\}, 3\rangle$, while at M_2 the view is $\langle\{c, d\}, 5\rangle$. There is a difference in the view identifiers due to a couple of membership changes that occurred at M_2's network component while M_1 and M_2 were disconnected.*

Now, the network failure is mended and all the processes in the system reconnect. M_1 receives a CONGRESS *membership notification that reflects the join of c and d, and M_2 receives a* CONGRESS *membership notification that reflects the join of a and b. M_1 emits the proposal $\langle G, \{a, b, c, d\}, \{\}, 4\rangle$ whereas M_2 emits the proposal $\langle G, \{a, b, c, d\}, \{\}, 6\rangle$. These proposed views are echoed in the flush messages. Once all the flush messages are collected, both servers report of the view $\langle G, \{a, b, c, d\}, 6\rangle$, since* 6 *is the maximum view identifier among the collected flush messages.*

5.6.4 Recovery from Server Failures

The MOSHE service is fault tolerant: If a MOSHE server crashes, its clients are transparently migrated to another MOSHE server, and the application program is unaware of this change.

When a client receives a server failure report, it tries to reconnect to an alternative server[6]. The live servers also receive a report of the server's failure via CONGRESS. When a live MOSHE server receives such a report, it waits for the failed server's clients to connect to it during a predefined time interval, called the *reconnection time interval.*

After the reconnection time interval is over, the servers exchange among themselves the list of reconnected clients, and issue a leave event for those clients that did not succeed to reconnect. Since the system is asynchronous, it is possible that a client will succeed to reconnect to a server only after the reconnection time interval is over. In this case, when the client reconnects the server notifies it that the reconnect is too late. The client then re-joins all the groups that it was previously a member of.

During the migration period, some messages that were in transit between the client and server may have been lost. We now describe how the protocol recovers from these message losses:

Recovery from a lost proposal or flush message After the reconnection time interval is over, each server emits proposals for pending views to the reconnected clients. The clients echo

[6]The alternative server may be located using CORBA services, or using a list of alternative servers that the client holds. For details please see [Now98].

these proposals in flush messages as usual. The servers ignore duplicate flush messages, and ignore flush messages which pertain to views that have already been cleared.

Recovery from a lost view report When a client connects to a new server it emits a join request for each group that corresponds to a pending view (a pending view is one for which the client received a proposal and has not received a view report yet). If the server receives a join request from a client that is included in the group membership, it assumes that the client had lost the view report, and re-sends it.

Recovery from a lost join or leave The client re-issues join/leave requests for every group that it tried to join/leave but did not receive a proposal reflecting this request. In order to keep track of these groups, the client also needs to receive proposals for groups that it is leaving. Such proposals are not echoed in flush messages.

5.7 Comparison with other membership algorithms

MOSHE is a scalable, one-round membership algorithm for wide area networks. We now compare MOSHE with related work.

Our design separates the maintenance of membership from the actual group multicast: membership is not maintained by every client but only by dedicated membership servers which are not concerned with the actual communication among clients in the groups. MOSHE extends CONGRESS and provides an interface for virtually synchronous communication semantics. Unlike Maestro [BFHR98], MOSHE does not wait for responses from clients asserting that virtual synchrony was achieved before delivering views.

Existing group communication systems that were designed for use in a WAN evolved from previous work on group communication systems for use in a LAN [DM96, MMSA$^+$96]. These systems leverage the idea that all WANs are interconnected LANs. These systems first run the original algorithm in each LAN, and then run another algorithm among the LANs, merging the individual memberships into one membership. This merged membership is then disseminated to all of the group members. Thus, these algorithms overcome the problem of remote failure detection by having the failure detection done at the LAN level. However, these algorithms are inherently multi-round, since an additional round is added to the algorithm run on each LAN. For example, the Totem multiple ring algorithm [MMSA$^+$96] takes two rounds per ring[7] plus an extra round for

[7]A ring is the logical representation of a LAN in Totem.

multiple rings [MMSA$^+$96].

"Light-weight" group membership algorithms [DM96, GBCvR93, RGS$^+$96, BFHR98] employ a client-server approach to both virtual synchrony and membership maintenance. In these algorithms, there are two levels of membership, *heavy-weight* and *light-weight*. The servers are part of the heavy-weight membership, and they use virtually synchronous communication among them. The clients are part of the light-weight membership. Most light-weight group membership services, e.g., [DM96, GBCvR93], do not preserve the semantics of the underlying heavy-weight membership services.

In these systems, when there is a membership change, the servers first compute the heavy-weight membership, and then map it to several light-weight process groups. As with the approach taken by us, this approach is scalable in the number of clients, since the membership algorithm involves reaching agreement among the servers only. However, computing the light-weight group membership requires additional communication after the heavy-weight membership algorithm is complete.

Unlike light-weight group membership algorithms, MOSHE only computes the process-level group membership, hence additional message rounds for computing the light-weight membership are not necessary. Furthermore, our service provides clients with full virtual synchrony semantics.

Light-weight group membership services have another important advantage: they scale well in the number of groups maintained, since they maintain the membership for several groups at the same time. Since in our design the same membership servers maintain the membership of all of the groups, MOSHE servers can also handle membership changes concerning several groups at the same time. Indeed, our implementation of the algorithm also possess this feature: if there are concurrent notifications concerning multiple groups, the membership server handles all of these groups together, and bundles the messages corresponding to different groups into a single message.

Thus, our algorithm provides the full semantics of heavy-weight group membership along with the scalability and flexibility of a light-weight group membership, all for the cost of a single communication round in the common case.

The only other single round membership algorithm that we are aware of is the one-round algorithm in [CS95]. This algorithm terminates within one round in case of a single process crash or join, but in case of network events that affect multiple processes, the algorithm may take a linear number of rounds, where in each round a token revolves around a virtual ring consisting of all of the processes in the system. Thus, the latency until the membership is complete and stable is $O(n^2\delta)$

where δ is the maximum message delay at stable times. Thus, this membership algorithm is not suitable for WANs, where δ tends to be big and typical network events are partitions and merges.

5.8 Advanced Group Membership Services

The client-server design of the membership service allows us to support a variety of advanced services without adding complexity to the clients and hence without paying a performance penalty. MOSHE provides advanced services such as hierarchical organization of groups, secure groups and group policies.

An important innovation of our membership service is the support for hierarchical directory services. MOSHE maintains a hierarchy of groups: a group may be a *sub-group* of a parent group. A parent group may contain a number of sub-groups. This concept is useful for applications containing a number of logically related groups, e.g., a conferencing application with several discussion groups.

MOSHE also supports secure group services: It implements authentication and authorization mechanisms that determine when a user is authorized to perform actions in a group, (e.g., create a sub-group for a specified group, query which sub-groups a group has, or join a group). Furthermore, the membership service may maintain two membership sets for each process group: *active members* who may provide input in the group, and *passive members* who receive messages sent to the group but cannot send messages to the group.

Thus, the authentication mechanism allows users to determine *policies* which restrict the ability of processes to become (active/passive) members of the group. The policies are declared when the group is created. If no policy is declared, then the policies are inherited from the parent group.

More sophisticated policies may be also imposed, e.g., restricting the number of members in a group, or even the properties of the members. For example, a cosmopolitan conference over the Internet may allow only two members from each country to participate in the discussion. If, due to a membership policy, a user's join request may not be currently fulfilled but may possibly be fulfilled later, MOSHE allows the user to *block* until the join will become possible.

Chapter 6

Totally Ordered Broadcast*

This chapter presents an algorithm for Totally Ordered Broadcast in the face of network partitions and process failures, using an underlying group communication service as a building block. The algorithm always allows a majority (or quorum) of connected processes in the network to make progress (i.e., to order messages), if they remain connected for sufficiently long, regardless of past failures. Furthermore, the algorithm always allows processes to initiate messages, even when they are not members of a majority component. These messages are disseminated to other processes using a gossip mechanism. Thus, messages can eventually become totally ordered even if their initiator is never a member of a majority component. The algorithm guarantees that when a majority is connected, each message is ordered within at most two communication rounds, if no failures occur during these rounds.

6.1 Introduction

Totally Ordered Broadcast is a powerful service for the design of fault tolerant applications, e.g., consistent cache, distributed shared memory and replication [Sch90, Kei94]. This chapter presents the *COReL (Consistent Object Replication Layer)* algorithm for Totally Ordered Broadcast in the face of network partitions and process failures. The algorithm is most adequate for dynamic networks where failures are transient. *COReL* uses an underlying *totally ordered group communication service (TO-GCS)* as a building block.

COReL multicasts messages to all the connected members using the underlying TO-GCS. Once messages are delivered by the TO-GCS and logged on stable storage (by *COReL*), they are acknowledged. Acknowledgments are piggybacked on regular messages. When a majority is connected, messages become totally ordered once they are acknowledged by all the members of the connected majority. Thus, the *COReL* algorithm guarantees that when a majority is connected, each message is ordered within two communication rounds at the most, if no failures occur during these rounds[1].

*This chapter is based on a paper by Keidar and Dolev [KD96].

[1]By "no failures occur" we implicitly mean that the underlying membership service does not report of failures.

The algorithm incurs low overhead, no "special" messages are needed, all the information required by the protocol is piggybacked on regular messages.

Processes using *COReL* are always allowed to initiate messages, even when they are not members of a majority component. By carefully combining message ordering within a primary component and gossiping of messages exchanged in minority components, messages can eventually become totally ordered even if their initiator is never a member of a majority component.

The protocol presented in this chapter uses a simple majority rule to decide which network component can become the primary one. Alternatively, one could use a quorum system [PW95], which is a generalization of the majority concept. A quorum system is collection of sets (quorums) such that any two sets intersect. Using such a quorum system, a network component can become the primary one if it contains a quorum. The concept of quorums may be further generalized to allow dynamic adjustment of the quorum system. In the next chapter, we present a dynamic voting protocol for maintaining the primary component in the system; we demonstrate how this protocol may be used in conjunction with *COReL*.

6.1.1 The Problem

The *Atomic Broadcast* [HT93] problem deals with consistent message ordering. Informally, Atomic Broadcast requires that all the correct processes will deliver all the messages to the application in the same order and that they eventually deliver all messages sent by correct processes, furthermore, all the *correct* processes deliver any message that is delivered by a correct processes.

In our model two processes may be detached, and yet both are considered correct. In this case, obviously, Atomic Broadcast as defined above is unsolvable (even if the communication is synchronous) [FKM$^+$95]. We define a variant of Atomic Broadcast for partitionable networks: We guarantee that if a majority of the processes form a connected component then these processes eventually deliver all messages sent by any of them, in the same order. We call this service *Totally Ordered Broadcast*.

It is well-known that in a fully asynchronous failure-prone environment, agreement problems such as Consensus and Atomic Broadcast are not solvable [FLP85, CT96], and it is impossible to implement an algorithm with the above guarantee (please see [FKM$^+$95]). Therefore, we augment the model with an *eventual perfect failure detector*, as defined in Section 2.3.

The term *delivery* is usually used for delivery of totally ordered messages by the Atomic Broadcast algorithm to its application, but also for delivery of messages by the GCS to its application (which in our case is the Totally Ordered Broadcast algorithm). To avoid confusion, in the rest of this chapter we will use the term delivery *only* for messages delivered by the GCS to our algorithm. When discussing the Totally Ordered Broadcast algorithm, we say that the algorithm *totally orders a message* when the algorithm decides that this message is the next message in the total order, instead of saying that the algorithm "delivers" the message to its application.

6.1.2 Related Work

Group communication systems often provide totally ordered group communication services. Isis [BSS91], Horus [vRBM96], Totem [AMMS$^+$95, MMSA$^+$96], Transis [DM96, CHD98, DKM93], Amoeba [KT96], RMP [WMK95], Delta-4 [Pow91] are only some of the group communication systems that support totally ordered group communication.

To increase availability, GCSs detect failures and extract faulty members from the membership. When processes reconnect, the GCS does not recover the states of reconnected processes. This is where the *COReL* algorithm comes in: it extends the order achieved by the GCS to a global total order.

The majority-based Consensus algorithms of [DLS88, Lam89, DPLL97, CT96, DFKM96] are guaranteed to terminate under conditions similar to those of *COReL*, i.e., at periods at which the network is stable and message delivery is timely, or when failure detectors are eventually accurate. Atomic Broadcast is equivalent to Consensus [CT96]; Atomic Broadcast may be solved by running a sequence of Consensus decisions [CT96, Lam89, DPLL97].

In [MHS89], the Paxos multiple Consensus algorithm of Lamport [Lam89] is used for a replicated file system. The replication algorithm suggested in [MHS89] is centralized, and thus highly increases the load on one server, while our protocol is decentralized and symmetric.

Another advantage of using *COReL* over running a sequence of Consensus algorithms is that *COReL* essentially pipelines the sequence of Consensus decisions. While Consensus algorithms involve special rounds of communication dedicated to exchanging "voting" messages of the protocol, in our approach all the information needed for the protocol is piggybacked on regular messages. Furthermore, *COReL* does not maintain the state of every Consensus invocation separately, the information about all the pending messages is summarized in common data structures. This allows faster recovery from partitions, when *COReL* reaches agreement on all the recovered messages simultaneously.

The Atomic Broadcast algorithm of [CT96] conserves special "voting" messages by reaching agreement on the order of sets of messages instead of running Consensus for every single message. However, this increases the latency of message ordering and still requires some extra messages.

The total ordering protocol in [Ami95, ADMSM94] resembles *COReL*; it also exploits a group communication service to overcome network partitions. Like *COReL*, it uses a majority-based scheme for message ordering. It decreases the requirement for end-to-end acknowledgments, at the price of not always allowing a majority to make progress.

Recently, Fekete *et al.* [FLS97] have studied the *COReL* algorithm (following its publication in [KD96]) using the I/O automata formalism. They have presented both the specifications and the implementation using I/O automata. They presented the liveness guarantees in terms of timed automata at periods during which the underlying network is stable and timely, rather than using the failure detector abstraction as we do here. They made simplifications to the protocol which make it simpler to present, but also less efficient.

The Total protocol [MMSA93] also totally orders messages in the face of process crashes and network partitions. However, it incurs a high overhead: The maximum number of communication rounds required is not bounded, while our algorithm requires two communication rounds to order a message if no failures occur during these rounds.

6.2 The System Architecture

COReL is an algorithm for Totally Ordered Broadcast. *COReL* is designed as a high-level service atop a *totally ordered group communication service (TO-GCS)* which provides totally ordered group multicast and membership services, and is omission fault free within connected network components.

All the copies of *COReL* are members of one group. Each copy of *COReL* uses TO-GCS to send messages to the members of its group; all the members of the group deliver (or receive) the message.

After a group is created, the group undergoes view changes when processes are added or are taken out of the group. The membership service reports these changes to *COReL* through views, as explained in Section 5.5. Views are delivered among the stream of regular messages. We say that a send (receive) event e occurs at process p in view v (or in the context of v) if v was the latest view that p received before e.

COReL uses a group membership service with the guarantees described in Section 5.5, augmented with a total ordering protocol such as [CHD98], as depicted in Figure 6.1.

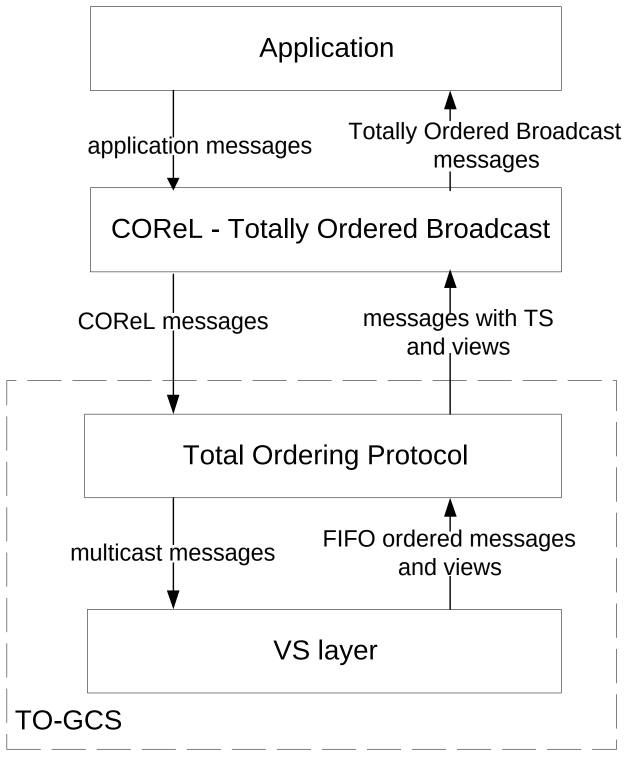

Application
application messages
Totally Ordered Broadcast messages
COReL - Totally Ordered Broadcast
COReL messages
messages with TS and views
Total Ordering Protocol
multicast messages
FIFO ordered messages and views
VS layer
TO-GCS

Figure 6.1: The layer structure of *COReL*.

6.2.1 Properties of the TO-GCS

TO-GCS extends the VS service described in Chapter 5 to provide total order of message delivery. The TO-GCS totally orders messages within each component, using a total ordering protocol such as ATOP [CHD98, Cho97], or All-Ack [DM95, Mal94].

When using ATOP over the VS service described in Chapter 5, the TO-GCS preserves all of its membership properties: View Identifier Local Monotony (Property 5.5.1), Self Inclusion (Property 5.5.2) and Agreement on Views (Property 5.5.3).

The TO-GCS also preserves the following properties of the VS service: Message Integrity (Property 2.2.1), No Duplication (Property 5.2.2), Self Delivery (Property 5.5.4), Termination of Delivery[2] (Property 5.5.5) and Virtual Synchrony (Property 5.5.7).

In addition, the TO-GCS fulfills the following properties:

[2]The ATOP [CHD98, Cho97] algorithm preserves Self Delivery and Termination of Delivery only if every live process sends infinitely many messages. This can be achieved by augmenting the service with a liveness mechanism which periodically sends "I-am-alive" messages when the process is idle.

Property 6.2.1 *A logical* timestamp (TS) *is attached to every message when it is delivered. Every message has a unique TS, which is attached to it at all the processes that deliver it. The TS total order preserves the* causal *partial order. The TO-GCS delivers messages at each process in the TS order.*

Among processes that do not remain connected we would like to guarantee agreement to some extent. If two processes become disconnected, we do not expect to achieve full agreement on the set of messages they delivered in the context of v_1 before detaching. Instead, we require that they agree on a subset of the messages that they deliver in v_1, as described below.

Let processes p and q be members of v_1. Assume that p delivers a message m before m' in v_1, and that q delivers m', but without delivering m. This can happen only if p and q became disconnected (from Properties 6.2.1 and 5.5.7, they will not both be members of the same next view). In Property 6.2.2 we require that if q delivers m' without m, then no message m'' sent by q, after delivering m', can be delivered by p in the context of v_1.

Property 6.2.2 *Let p and q be members of view v. If p delivers a message m before m' in v, and if q delivers m' and later sends a message m'', such that p delivers m'' in v, then q delivers m before m'.*

These properties are fulfilled by the ATOP [CHD98, Cho97] and All-Ack [DM95, Mal94] protocols when used in conjunction with the VS membership service described in Chapter 5.

6.3 Problem Definition: The Guarantees of *COReL*

Safety

COReL fulfills the following two safety properties:

Property 6.3.1 *At each process, messages become totally ordered in an order which is a prefix of some common global total order. I.e., for any two processes p and q, and at any point during the execution of the protocol, the sequence of messages totally ordered by p is a prefix of the sequence of messages totally ordered by q, or vice versa.*

Property 6.3.2 *Messages are totally ordered by each process in an order preserving the causal partial order.*

In addition, *COReL* preserves the following properties of underlying TO-GCS service: Message Integrity (Property 2.2.1) and No Duplication (Property 5.2.2).

Liveness

COReL guarantees that if a majority of the processes form a permanently connected component, and the failure detector is an eventual perfect one (cf. Section 2.3), then these processes eventually totally order all messages sent by any of them.

6.4 The *COReL* Algorithm

We present the *COReL* algorithm for reliable multicast and total ordering of messages. The *COReL* algorithm is used to implement long-term replication services using a TO-GCS service as described above. *COReL* guarantees that all messages will reach all processes in the same order. It always allows members of a connected primary component to order messages. The algorithm is resilient to both process failures and network partitions.

6.4.1 Reliable Multicast

When the network partitions, messages are disseminated in the restricted context of a smaller view, and are not received at processes which are members of other components. The participating processes keep these messages for as long as they might be needed for retransmission. Each process logs (on stable storage) every message that it receives from the TO-GCS. A process acknowledges a message after it is written to stable storage. The acknowledgments (*ACKs*) may be piggybacked on regular messages. Note that it is important to use application level ACKs in order to guarantee that the message is logged on stable storage. If the message is only ACKed at the TO-GCS level, it may be lost if the process crashes.

When network failures are mended and previously disconnected network components remerge, a Recovery Procedure is invoked; the members of the new view exchange messages containing information about messages in previous components and their order. They determine which messages should be retransmitted and by whom.

When a process crashes, a message that it sent prior to crashing may be lost. When a process recovers from such a crash, it needs to recover such messages. Therefore, messages are stored (on stable storage) when they are received by the application (before the application send event is complete).

6.4.2 Message Ordering

Within each component messages are ordered by the TO-GCS layer, which supplies a unique timestamp (TS) for each message when it delivers the message to *COReL*. When *COReL* receives the message, it writes the message on stable storage along with its TS. Within a majority component *COReL* orders messages according to their TS. The TS is globally unique, even in the face of partitions, and yet *COReL* sometimes orders messages in a different total order: it orders messages from majority component before (causally concurrent) messages with a possibly higher TS from minority components (otherwise it wouldn't always allow a majority to make progress). Note that both the TS order and the order provided by *COReL* preserve the causal partial order.

When a message is retransmitted, the TS that was given when the original transmission of the message was received is attached to the retransmitted message, and is the only timestamp used for this message (the new TS generated by the TO-GCS during retransmission is ignored).

We use the notion of a *primary component* to allow members of one network component to continue ordering messages when a partition occurs. For each process, the *primary component bit* is set iff this process is currently a member of a primary component. In Section 6.4.5 we describe how a majority of the processes may become a primary component. Messages that are received in the context of a primary component (i.e., when the primary component bit is set) may become totally ordered according to the following rule:

Order Rule 1 *Members of the current primary component PM are allowed to totally order a message (in the global order) once the message was acknowledged by all the members of PM.*

If a message is totally ordered at some process p according to this rule, then p knows that all the other members of the primary component received the message, and have written it on stable storage. Furthermore, the algorithm guarantees that all the other members already have an obligation to enforce this decision in any future component, using the *yellow message mechanism* explained below.

COReL maintains a local message queue $\mathcal{MQ}$, that is an ordered list of all the messages that this process received from the application or TO-GCS. After message m was received by *COReL* at process p, and p wrote it on stable storage (in its $\mathcal{MQ}$) we say that p *has* the message m. Messages are uniquely identified through a pair $< sender, counter >$. This pair is the *message id*.

Incoming messages within each component are inserted at the end of the local $\mathcal{MQ}$, thus $\mathcal{MQ}$ reflects the order of the messages local to this component. Messages are also inserted to the $\mathcal{MQ}$

(without a TS) when they are received from the application. Once Self Delivery occurs, these messages are tagged with the TS and moved to their proper place in the $\mathcal{MQ}$. When components merge, retransmitted messages from other components are inserted into the queue in an order that may interleave with local messages (but never preceding messages that were ordered already).

The Colors Model

COReL builds its knowledge about the order of messages at other processes. We use the colors model defined in [AAD93] to indicate the knowledge level associated with each message, as follows:

green: Knowledge about the message's global total order. A process marks a message as green when it knows that all the other members of the primary component know that the message is yellow. Note that this is when the message is totally ordered according to Order Rule 1. The set of green messages at each process at a given time is a prefix of $\mathcal{MQ}$. The last green message in $\mathcal{MQ}$ marks the *green line.*

yellow: Each process marks as yellow messages that it received and acknowledged in the context of a primary component, and as a result, might have become green at other members of the primary component. The yellow messages are the next candidates to become green. The last yellow message in $\mathcal{MQ}$ marks the *yellow line.*

red: No knowledge about the message's global total order. A message in $\mathcal{MQ}$ is *red* if there is no knowledge that it has a different color. Yellow messages precede all the red messages in $\mathcal{MQ}$. Thus, $\mathcal{MQ}$ is divided into three zones: a green prefix, then a yellow zone and a red suffix.

As explained in [AAD93, Kei94], it is possible to provide the application with red messages if weak consistency guarantees are required. For example, eventually serializable data services [PL91, FGL^{+}96, AAD93] deliver messages to the application before they are totally ordered. Later, the application is notified when the message becomes *stable* (green in our terminology). Messages become stable at the same order at all processes. The advantage of using *COReL* for such applications is that with *COReL* messages become stable even whenever a majority is connected, while with the implementations presented in [PL91, FGL^{+}96, AAD93], messages may become stable only after they are received by all the processes in the system.

When a message is marked as green it is totally ordered. If a member of a primary component PM marks a message m as green according to Order Rule 1 then for all the other members of PM, m is yellow or green. Since two majorities always intersect, and every primary component contains a majority, in the next primary component that will be formed at least one member will have m as yellow or green. When components merge, members of the last primary component enforce all the green and the yellow messages that they have before any concurrent red messages. Concurrent red messages from distinct components are interleaved according to the TS order.

6.4.3 Notation

We use the following notation:

- $\mathcal{MQ}^p$ is the $\mathcal{MQ}$ of process p.
- $\mathit{Prefix}(\mathcal{MQ}^p, m)$ is the prefix of $\mathcal{MQ}^p$ ending at message m.
- $\mathit{Green}(\mathcal{MQ}^p)$ is the green prefix of $\mathcal{MQ}^p$.
- We define *process p knows of a primary component PM* recursively as follows:
  1. If a process p was a member of PM then p *knows* of PM.
  2. If a process q *knows* of PM, and p recovers the state of q[3], then p *knows* of PM.

6.4.4 Invariants of the Algorithm

The order of messages in $\mathcal{MQ}$ of each process always preserves the causal partial order. Messages that are totally ordered are marked as green. Once a message is marked as green, its place in the total order may not change, and no new message may be ordered before it. Therefore, at each process, the order of green messages in $\mathcal{MQ}$ is never altered. Furthermore, the algorithm totally orders messages in the same order at all processes, therefore the different processes must agree on their green prefixes.

The following properties are *invariants* maintained by each step of the algorithm:

Causal
- If a process p has in its $\mathcal{MQ}$ a message m that was originally sent by process q, then for every message m' that q sent before m, $\mathcal{MQ}^p$ contains m' before m.
- If a process p has in its $\mathcal{MQ}$ a message m that was originally sent by process q, then for every message m' that q had in its $\mathcal{MQ}$ before sending m, $\mathcal{MQ}^p$ contains m' before m.

No Changes in Green New green messages are appended to the end of $\mathit{Green}(\mathcal{MQ}^p)$, and this is the only way that $\mathit{Green}(\mathcal{MQ}^p)$ may change.

[3] p recovers the state of q when p completes running the Recovery Procedure for a view that contains q.

Agreed Green The processes have compatible green prefixes: for every pair of processes p and q running the algorithm, and for every $Green(\mathcal{MQ}^p)$, (at every point in the course of the algorithm), and every $Green(\mathcal{MQ}^q)$, one of $Green(\mathcal{MQ}^p)$ and $Green(\mathcal{MQ}^q)$ is a prefix of the other.

Yellow If a process p marked a message m as green in the context of a primary component PM, and if a process q *knows* of PM, then:

1. Process q has m marked as yellow or green.
2. $Prefix(\mathcal{MQ}^q, m) = Prefix(\mathcal{MQ}^p, m)$.

In Appendix A.1 we formally prove that these invariants hold in *COReL*, and thus prove the correctness of *COReL*.

6.4.5 Handling View Changes

View Change Handler for View v:

- Unset the primary component bit.
- Stop handling regular messages, and stop sending regular messages.
- If v contains new members, run the Recovery Procedure described in Section 6.4.5.
- If v is a majority, run the algorithm for establishing a new primary component, described in Section 6.4.5.
- Continue handling and sending regular messages.

Figure 6.2: View change handler.

The main subtleties of the algorithm are in handling view changes. Faults can occur at any point in the course of the protocol, and the algorithm ensures that even in the face of cascading faults, no inconsistencies are introduced. To this end, every step taken by the handler for view changes must maintain the invariants described in Section 6.4.4.

When merging components, messages that were passed in the more restricted context of previous components need to be disseminated to all members of the new view. Green and yellow messages from a primary component should precede messages that were concurrently passed in other components. All the members of the new view must agree upon the order of all past messages. To this end, the processes run the Recovery Procedure.

If the new view v introduces new members, the Recovery Procedure is invoked in order to bring all the members of the new view to a common state. New messages that are delivered in the context of v are not inserted into $\mathcal{MQ}$ before the Recovery Procedure ends, and thus the **Causal** invariant is not violated. The members of v exchange *state messages*, containing information about messages in previous components and their order. In addition, each process reports of the last primary component that it knows of, and of its green and yellow lines. Every process that receives all the state messages knows exactly which messages every other member has, and the messages that not all the members have are retransmitted.

In the course of the Recovery Procedure, the members agree on the green and yellow lines. The new green line is the *maximum* of the green lines of all the members: Every message that one of the members of v had marked as green, becomes green for all the members. The members that *know* of the latest primary component, PM, determine the new yellow line. The new yellow line is the *minimum* of the yellow lines of the members that know of PM. If some message m is red for a member that knows of PM, then by the **Yellow** invariant, it was not marked as green by any member of PM. In this case if any member had marked m as yellow, it changes m back to red. A detailed description of the Recovery Procedure is presented in Section 6.4.5.

After reaching an agreed state, the members of a majority component in the network may practice their right to totally order new messages. They must order all the yellow messages first, before new messages, and before red messages form other components, in order to be consistent with decisions made in previous primary components.

If the new view is a majority, the members of v will try to establish a new primary component. The algorithm for establishing a new primary component is described in Section 6.4.5. All committed primary components are sequentially numbered. We refer to the primary component with sequential number i as PM_i.

When a view change is delivered, the handler described in Figure 6.2 is invoked. In the course of the run of the handler, the primary component bit is unset, regular messages are blocked, and no new regular messages are initiated.

Establishing a Primary Component

A new view, v, is established as the new primary component, if v is a majority, after the retransmission phase described in Section 6.4.5. The primary component is established in a three-phase agreement protocol, similar to Three Phase Commit protocols [Ske82, KD98]. The three phases are required in order to allow for recovery in case failures occur in the course of the establishing process. The three phases correlate to the three levels of colors in $\mathcal{MQ}$.

In the first phase all the processes multicast a message to notify the other members that they **attempt** to establish the new primary component. In the second phase, the members **commit** to establish the new primary component, and mark all the messages in their $\mathcal{MQ}$ as yellow. In the **establish** phase, all the processes mark all the messages in their $\mathcal{MQ}$ as green and set the primary component bit to TRUE. A process marks the messages in its $\mathcal{MQ}$ as green only when it knows that all the other members marked them as yellow. Thus, if a failure occurs in the course of the protocol, the **Yellow** invariant is not violated. If the TO-GCS reports of a view change before the process is over – the establishing is aborted, but none of its effects need to be undone. The primary component bit remains unset until the next successful establish process.

Establishing a New Primary Component
If v contains new members, the Recovery Procedure is run first. If v is a majority, all members of a view v try to establish it as the new primary component:

- Compute: $New\_Primary =$ $\max_{i \in C}(Last\_Attempted\_Primary_i) + 1$.
  The members of v now try to establish it as $PM_{New\_Primary}$.
- Attempt to establish $PM_{New\_Primary}$:
  Set *Last_Attempted_Primary* to *New_Primary* on stable storage, and send an attempt message, to notify the other members of the attempt.
- Wait for attempt messages from all members of v. When these messages arrive, do the following in one atomic step:
  1. Commit to the view by setting *Last_Committed_Primary* to *New_Primary* on stable storage.
  2. Mark all the messages in the $\mathcal{MQ}$ that are not green as yellow.

  Send a commit message, to notify the other members of the commitment.
- Wait for commit messages from all members of v, and then *establish* v, by setting to TRUE the primary component bit. Mark as green all the messages in $\mathcal{MQ}$.
- If the transport layer reports of a view change before the process is over – the establishing is aborted, but its effects are not undone.

Figure 6.3: Establishing a new primary component.

Each process maintains the following variables:

Last_Committed_Primary is the number of the last primary component that this process has committed to establish.

Last_Attempted_Primary is the number of the last primary component that this process has attempted to establish. This number may be higher than the number of the last component actually committed to, in the case of failures.

The algorithm for establishing a new primary component is described in Figure 6.3.

Recovery Procedure

Retransmission Rule *If process p has messages m and m′ such that m′ is ordered after m in p's messages queue, then during Step 7 of the Recovery Procedure:*

- *If p has to retransmit both messages then it will retransmit m before m′.*
- *If p has to retransmit m′ and another process q has to retransmit m then p does not retransmit m′ before receiving the retransmission of m.*

Figure 6.4: Retransmission rule.

If the new view, v, introduces new members, then each process that delivers the view change runs the following protocol:

Recovery Procedure for process p and view v

1. Send state message including the following information:
   - *Last_Committed_Primary.*
   - *Last_Attempted_Primary.*
   - For every process q, the id of the last message that p received from q[4].
   - The id of the latest green message (green line).
   - The id of the latest yellow message (yellow line).
2. Wait for state messages from all the other processes in $v.M$.
3. Let: $Max\_Committed = \max_{p \in v.set} Last\_Committed\_Primary_p$.

   Let *Representatives* be the members that have:
   $Last\_Committed\_Primary = Max\_Committed$.

   The *Representatives* advance their green lines to include all messages that any member of v had marked as green, and retreat their yellow lines to include only messages that all

[4]Note that this is sufficient to represent the set of messages that p has, because the order of messages in $\mathcal{MQ}^p$ always preserves the *causal* order.

of them had marked as yellow, and in the same order. For example, if process p has a message m marked as yellow, while another member with $Last\_Committed\_Primary = Max\_Committed$ has m marked as red, or does not have m at all, then p changes to red m along with any messages that follow m in $\mathcal{MQ}^p$.

4. If all the members have the same last committed primary component, (i.e., all are *Representatives*), go directly to Step 7.

   A unique representative from the group of *Representatives* is chosen deterministically.

   Determine (from the state messages) the following sets of messages:

   component_stable is the set of messages that all the members of v have.

   component_ordered is the set of messages that are green for all the members of v.

   priority are yellow and green messages that the representative has.

5. Retransmission of priority messages:

   The chosen representative computes the maximal prefix of its $\mathcal{MQ}$ that contains component_ordered messages only. It sends the set of priority messages in its $\mathcal{MQ}$ that follow this prefix. For component_stable messages, it sends only the header (including the original ACKs), and the other messages are sent with their data and original piggybacked ACKs.

   Members from other view insert these messages into their $\mathcal{MQ}$s, in the order of the retransmission, following the green prefix, and ahead of any non_priority messages[5].

6. If $Last\_Committed\_Primary_p < Max\_Committed$; do the following in one atomic step:

   - If p has yellow messages that were not retransmitted by the representative, change these messages to red, and reorder them in the red part of $\mathcal{MQ}$ according to the TS order.
   - Set $Last\_Committed\_Primary$ to $Max\_Committed$ (on stable storage).
   - Set the green and yellow lines according to the representative; the yellow line is the last retransmitted message.

7. Retransmission of red messages:

   Messages that not all the members have, are retransmitted. Each message is retransmitted by at most one process. The processes that need to retransmit messages send them, with their original ACKs, in an order maintaining the Retransmission Rule described in Figure 6.4.

   Concurrent retransmitted messages from different processes are interleaved in $\mathcal{MQ}$ according to the TS order of their original transmissions.

Note: If the TO-GCS reports of a view change before the protocol is over, the protocol is immediately restarted for the new view. The effects of the non-completed run of the protocol do not need to be undone.

[5]Note that it is possible for members to already have some of these messages, and even in a contradicting order (but in this case, not as green messages). In this case they adopt the order enforced by the representative.

After receiving all of the retransmitted messages, if v is a majority then the members try to establish a new view. (The algorithm is described Section 6.4.5).

If the view change reports only of process faults, and no new members are introduced, the processes need only establish the new view and no retransmissions are needed. This is due to the fact that, from Property 5.5.7 of the TO-GCS, all the members received the same set of messages until the view change.

Chapter 7

Dynamic Voting for Primary Components*

Fault tolerant distributed systems often use quorum systems in order to guarantee consistency; a quorum system is collection of sets (quorums) such that any two sets intersect. Traditionally, the same quorum system is used throughout the system's life time, and when new processes join, the system is re-started with a new quorum system. Many new applications, e.g., conferencing applications and interactive games, wish to allow users to freely join and leave, without restarting the entire system. The dynamic voting paradigm allows such systems to change the quorum system on the fly and define quorums adaptively, accounting for the changes in the set of participants. Furthermore, dynamic voting was shown to lead to more available services than any other paradigm for maintaining a primary component in unreliable networks.

Nonetheless, implementing dynamic voting bears subtleties. In fact, many of the suggested dynamic voting protocols may lead to inconsistencies which stem from concurrent existence of two disjoint supposedly primary components. Other protocols severely limit the availability in case failures occur while the protocol is trying to form a new quorum system.

This chapter presents a robust and efficient dynamic voting protocol for unreliable asynchronous networks. The protocol consistently maintains the primary component in a distributed system. The protocol allows the system to make progress in cases of repetitive failures in which previously suggested protocols block. The protocol is simple to implement.

7.1 Introduction

Many fault tolerant distributed systems (e.g., ISIS [BvR94], Phoenix [MS94], Consul [MPS91b, MPS93] and xAMp [RV92]) use the primary component paradigm to allow a subset of the processes to function when failures occur. A majority (or quorum) of the processes is often chosen to be the primary component. In unreliable networks this is problematic: Repeated failures may cause majorities to further split up, leaving the system without a primary component. To overcome this

*This chapter is based on a paper by Yeger Lotem, Keidar and Dolev [YLKD97].

problem, the dynamic voting paradigm was introduced; dynamic voting was exploited in the ISIS system as early as 1985 [Bir].

A "classical" (static) quorum system is a collection of sets (quorums) such that any two sets intersect. The dynamic voting paradigm, on the other hand, defines quorums adaptively: When a partition occurs, a new and possibly smaller quorum may be chosen such that each newly formed quorum contains a majority of the previous one, but does not necessarily intersect all the previous quorums. Stochastic models analysis [JM90], simulations [PL88], and empirical results [AW96] show that dynamic voting leads to more available services than any other paradigm for maintaining a primary component.

Another important benefit of the dynamic voting paradigm is its flexibility to support a dynamically changing set of processes. With emerging world-wide communication technology, new applications wish to allow users to freely join and leave. Using dynamic voting, such systems can dynamically adapt to the changes in the set of participants.

In this chapter we present a robust and efficient protocol for maintaining a primary component using dynamic voting in an asynchronous environment, where processes and communication links may fail. By recording historical information, our protocol allows the system to make progress in certain cases in which previously suggested protocols either block, require a cold start of the entire system, or lead to inconsistencies. Furthermore, our protocol is simple to implement.

Previously suggested dynamic voting protocols were presented as part of replication and transaction management algorithms [DB85, PL88, EAD91, JM90, Her86, Jaj87, Ami95] or as part of group communication systems [RB91, MS94]. We decouple the primary component maintenance from both the group communication mechanism and the application, and focus solely on maintaining the primary component. Our algorithm may be incorporated in any distributed application that makes progress in a primary component, e.g., replication algorithms [Kei94, EAD91], transaction management [KD98], and infrastructure systems like the ISIS toolkit [BvR94]. In Section 7.7 I demonstrate how it may be incorporated within the *COReL* algorithm presented in Chapter 6.

If a failure occurs during an execution of the protocol while a new primary component is being formed, some previously suggested dynamic voting protocols (e.g., [JM90, Ami95]) block until all the members of the last primary component become reconnected. Blocking until all the members reconnect significantly reduces the availability, especially in failure-prone environments (for which dynamic voting is most suitable) and in applications with a dynamic set of participants, where a waited upon process might have voluntarily left the system. In contrast, our protocol requires only a majority of the members that attempted to form the last primary component to become reconnected in order to make progress in such cases. Note that the analyses of the availability of dynamic voting do not take the possibility of blocking into consideration, and therefore the actual availability of protocols prone to blocking is lower than expected.

Unlike the dynamic voting protocol of ISIS [RB91], our protocol recovers from situations in which the primary component was transiently lost (e.g., when the primary component partitions into three minority groups which later reconnect) without requiring a cold start of the entire system. In ISIS only members of the primary component are considered alive; members of minority components must "commit suicide" and restart under a new identity. Therefore, the primary component cannot be reconstructed after it is lost, and a cold start is required. In approach, processes may be active even when they are not members of a primary component. This feature is exploited in the algorithm presented in Chapter 6, where processes may issue updates even when they are members of non-primary components.

The challenge in designing consistent dynamic voting protocols is in coping with failures that occur while the processes are trying to form a new primary component. Careless handling of such cases may lead to concurrent existence of two disjoint primary components: Once a process detaches, it is impossible for other processes to know whether it received a specific message before its detachment, or not. Some past protocols (e.g., [DB85, PL88, EAD91]) lead to inconsistent results in such cases, as demonstrated by the following typical scenario:

Scenario 1
- *The system consists of five processes: a, b, c, d and e. The system partitions into two components: a, b, c and d, e.*
- *a, b and c try to form a new primary component. To this end, they exchange messages.*
- *a and b form the primary component $\{a, b, c\}$, assuming that process c does so too. However, c detaches before receiving the last message, and therefore is not aware of this primary component. a and b remain connected, while c connects with d and e.*
- *a and b notice that c detached, therefore form a new primary component $\{a, b\}$ (a majority of $\{a, b, c\}$).*
- *Concurrently, c, d and e form the primary component $\{c, d, e\}$ (a majority of $\{a, b, c, d, e\}$).*
- *The system now contains two live primary components, which may lead to inconsistencies.*

Our protocol overcomes the difficulty demonstrated in the scenario above by maintaining another level of knowledge. The protocol guarantees that if a and b succeed in forming $\{a, b, c\}$, then c is aware of this possibility. From c's point of view, the primary components $\{a, b, c\}$ is ambiguous: It might have or might have not been formed by a and b. In general, every process records, along with the last primary component it formed, later primary components that it attempted to form but detached before actually forming them. These ambiguous attempts are taken into account in later attempts to form a primary component.

Some previously suggested protocols avoid inconsistencies by running two phase commit ([JM90, Her86]), or similar mechanisms ([Ami95]) that cause processes to block when their latest primary component is ambiguous. This imposes limitations on the system's progress, as demonstrated in the following typical scenario:

Scenario 2

- *The system consists of five processes: a, b, c, d and e. The system partitions into two components: a, b, c and d, e.*
- *a, b and c try to form a new primary component. To this end, they run a two phase commit protocol of which a is the coordinator.*
- *a crashes. b and c detach before the two phase commit ends. They remain blocked.*
- *b and c re-connect with d and e, but they cannot form a primary component.*

The problem is especially severe in environments with a dynamic set of participants: If a has permanently left, the system remains blocked forever. With our protocol, the system may remain blocked only if a majority of the members that attempted to form the quorum leave the system. Specifically, when running our protocol in the scenario above, b and c do not block. Instead, they "remember" that they made an ambiguous attempt with a, namely $\{a, b, c\}$. Since the group $\{b, c, d, e\}$ contains a majority of $\{a, b, c\}$, it is an eligible new primary component.

In [MS94], a three phase Consensus protocol [CT96] is employed in order to allow a majority to resolve ambiguous attempts. This induces a high overhead that makes the protocol infeasible for use in practice: When a majority of the previous primary component reconnects, [MS94] requires at least five communication rounds in order to resolve the previous attempt and form a new primary component. Our protocol avoids such excessive communication by using pipelining: The status of past ambiguous attempts is resolved while new primary components are being formed. Thus, when a majority of the previous primary component reconnects, only two communication rounds are required in order to form a new one. Our protocol is required to record several ambiguous attempts in case failures cascade.

Unfortunately, recording all ambiguous attempts is not feasible: The number of ambiguous attempts a process might need to record may be exponential in the number of participating processes. In [YLKD97], we consider a simple mechanism that records only the latest ambiguous attempt, and show that it does not work; we demonstrate that in order to preserve consistency it may be necessary to consider an ambiguous attempt even if it is followed by an exponential number of ambiguous attempts. Taking a huge number of attempts into consideration limits the possibility of progress in the system, and may cause the system to block. An important contribution of our work is in providing a simple "garbage collection" mechanism for reducing the number of attempts that a process needs to record to at most n, where n is the number of processes in the system. Practically, the number of attempts a process may need to consider is expected to be small. Thus, our protocol achieves a good balance between the historical data it stores, the restrictions on the ability to make progress in the system and the number of communication rounds.

The main criticism of the dynamic voting paradigm is that there can be situations where almost all of the processes in the system are connected but cannot form a new quorum because of the potential existence of a past surviving quorum held by a single process. To prevent such situations, our protocol allows users to set a lower bound, $\mathit{Min\_Quorum}$, on the size of quorums. This way, every component containing more than $n - \mathit{Min\_Quorum}$ members (where n is the number of processes in the system, and $\mathit{Min\_Quorum} < n/2$) can always form a quorum, regardless of past events in the system. We developed a novel mechanism for providing this feature in environments that allow new processes to join on the fly.

7.2 The Model

The processes are connected by an asynchronous communication network as described in Chapter 2.

The initial primary component in the system consists of a core group of processes, $\mathcal{W}_0$, that is known to all the processes in $\mathcal{W}_0$. The core group is typically the initial configuration on which the system manager runs a protocol (e.g., the sites running a distributed database). The set of all the processes that may run the protocol is unknown to any of the processes in advance, and thus the configuration may change dynamically. Processes that do not belong to $\mathcal{W}_0$ are aware of the fact that they are not members of the core group.

7.2.1 The Membership Service

Maintaining the primary component is typically decoupled into two separate problems: first, determining the set of connected processes, and second, deciding whether a set of processes is the primary component. Like other dynamic voting protocols, we focus on solving the latter problem, assuming a separate mechanism that solves the former.

Our dynamic voting protocol assumes a membership mechanism no stronger than those assumed in [DB85, Jaj87, PL88, JM90, EAD91, Ami95]. Each process is equipped with an underlying membership module, e.g., [ADKM92a, AMMS$^+$93, EMS95, CS95]. When the membership module senses failures or recoveries, it reports to the process of the new membership, i.e., the set of processes that are currently assumed to be connected.

Our only requirement of the membership service is that every message is received in the view in which it was sent (cf. Property 5.5.6 of the membership service in Chapter 5). This can be achieved either by refraining from sending messages while a membership change takes place or by discarding old messages that arrive after a membership change. In order to discard messages from previous memberships, the protocol needs to provide a locally unique view identifier (cf. Property 5.5.1 of the membership service in Chapter 5), with which all the messages sent in this membership will be tagged. These requirements are fulfilled by simple and efficient membership protocols, e.g., the protocol in Chapter 5 or the one round membership protocol of [CS95], which terminate after one communication round.

As shown in [CHTCB96], it is impossible to reach agreement upon the current membership in an asynchronous system. Hence, we do not assume that the membership reports accurately reflect the network situation, nor is the membership reported atomically to all the processes. The dynamic voting protocol we present is correct (i.e., guarantees a total order on primary components) regardless of whether the membership mechanism is live and accurate or not. The liveness of the protocol (its ability to form new primary components when the network situation changes) depends on the accuracy and liveness of this membership mechanism.

7.3 Problem Definition

In this chapter we present a primary component maintenance service, that allows a group of processes to form a primary component in a consistent way. Such a service is required to impose a total order on all the primary components formed in the system. When using a static quorum system, the order is easily provided using the following property: "every two primary components intersect". Unfortunately, dynamic quorum systems do not possess this property. Instead, a total order on primary components is defined by extending the causal order on components that do intersect.

Let P and P' be two primary components. If $j \in P \cap P'$, and j participates in both of these primary components, i.e., attempts to form both, then j participates in one of P and P' before the other. A process does not participate in two quorums concurrently. If j participates in P first, we denote the transitive closure of this relation by: $P \prec P'$. The requirement from a dynamic paradigm for maintaining primary components is that $\prec$ is a total order. Since a process is a member of at most one component at any given time, the total order on primary components implies that at any given time there is at most one live and connected primary component in the system.

7.4 The Primary Component Protocol

We present a protocol for maintaining the primary component in an asynchronous system. Initially, the primary component in the system is the core group, $\mathcal{W}_0$. Whenever a membership change is reported, the notified members invoke a new session of the protocol, trying to form a new primary component. If they succeed, then at the end of the session they form a new primary component P, which persists until the next membership change. Each process independently invokes the protocol once it receives the membership message.

In this section, we present a simplified version of the protocol in which the members of the core group, $\mathcal{W}_0$, have a special status: every quorum in the system must contain a threshold of members from $\mathcal{W}_0$. In Section 7.6 we modify the protocol to eliminate this special status.

7.4.1 Dynamic Quorums

Our protocol uses dynamic voting to determine when a group of processes is eligible to be the next primary component in the system. Originally, dynamic voting was implemented by allowing a majority of the previous primary component to become the new primary component [DB85]. Dynamic linear voting [Jaj87], optimizes this scheme by breaking ties between groups of equal size using a linear order, $\mathcal{L}$, imposed on all the potential processes in the system. We extend dynamic linear voting with another parameter: *Min_Quorum*, the minimum quorum size allowed in the system.

We define a predicate $Next\_Quorum(S,T)$, that is TRUE iff T can become the new primary component in the system, given that the previous primary component was S. Formally, $Next\_Quorum(S,T)$ is TRUE iff:

1. $|T \cap \mathcal{W}_0| \geq Min\_Quorum$, and

2. (a) $|T \cap S| > |S|/2$, or

   (b) $|T \cap S| = |S|/2$ and $\exists p \in T \cap S$ such that $\forall q \in S \setminus T \;\; \mathcal{L}(p) > \mathcal{L}(q)$, or

   (c) $|T \cap \mathcal{W}_0| > |\mathcal{W}_0| - Min\_Quorum$.

Requirements 2(a) and 2(b) describe dynamic linear voting. Requirement 1 sets a threshold on the minimum quorum size allowed in the system. The complementing requirement, 2(c), allows a large group of processes to become a primary component regardless of the system's history. However, it still requires every quorum in the system to contain at least *Min_Quorum* members of $\mathcal{W}_0$. In Section 7.6 we relax this restriction, and require, instead, that a quorum will contain *Min_Quorum* processes.

It is easy to see that the *Next_Quorum* predicate has the following properties:

1. If $Next\_Quorum(S,T)$ then $S \cap T \neq \emptyset$.
2. If $Next\_Quorum(S,T)$ and $Next\_Quorum(S,T')$ then $T \cap T' \neq \emptyset$.

We extend the definition of the *Next_Quorum* predicate so that $Next\_Quorum(\infty,T)$ is FALSE for every set T.

7.4.2 Variables and Notation

The protocol is conducted in sessions, and the sessions are numbered. A session S of the protocol is identified by its membership, $S.M$, and session number, $S.N$. Each process p maintains the following variables:

$Is\_Primary_p$ is a boolean variable that is TRUE iff the current membership is the primary component in the system. If $p \in \mathcal{W}_0$, then it is initialized to TRUE, and otherwise to FALSE.

$Session\_Number_p$ is the current session number. This variable is initialized to 0.

$Last\_Primary_p$ is the last primary component that process p formed (i.e., the membership and *Session_Number* of the session in which the last primary component was formed). If $p \in \mathcal{W}_0$ then it is initialized to $(\mathcal{W}_0, 0)$ and otherwise to $(\infty, -1)$.

$Ambiguous\_Sessions_p$ is the set of ambiguous sessions that process p attempted to form after p participated in $Last\_Primary_p$. For each ambiguous session (or attempt) S in this set, p maintains an associative array (which acts like a function) $S.A$. For every $q \in S.M$: if p knows that q formed S then $S.A(q) = 1$; if p knows that q did not form S then $S.A(q) = -1$ and otherwise $S.A(q) = 0$. The set of ambiguous sessions is initially empty.

$Last\_Formed_p$ is an associative array. For each process q that p participated in a session with, $Last\_Formed_p(q)$ is the last session (membership and number) that p formed and q was a member of. Initially, if $p, q \in \mathcal{W}_0$ then $Last\_Formed_p(q)$ is $(\mathcal{W}_0, 0)$. Otherwise, it is $(\infty, -1)$.

We use the following notation:

- $\mathcal{M}$ is the membership as reported in the membership message that invoked the current session of the protocol. The membership is a set of processes.
- $\mathit{Max\_Session}$ is: $\max_{p \in \mathcal{M}}(\mathit{Session\_Number}_p)$.
- $\mathit{Max\_Primary}$ is: $\mathit{Last\_Primary}_p$ $s.t.$ $\mathit{Last\_Primary}_p.N = \max_{q \in \mathcal{M}}(\mathit{Last\_Primary}_q.N)$.
- $\mathit{All\_Ambiguous\_Sessions}$ is: $\bigcup_{p \in \mathcal{M}}(A \in \mathit{Ambiguous\_Sessions}_p \text{ s.t. } A.N > \mathit{Max\_Primary}.N)$.

In order to simplify notations, we extend the definition of the $\mathit{Next\_Quorum}$ predicate to sessions. For a pair of sessions $S1, S2$, we define $\mathit{Next\_Quorum}(S1, S2)$ as: $\mathit{Next\_Quorum}(S1.M, S2.M)$.

7.4.3 The Protocol

Whenever a membership change is reported, the notified members invoke a new session of the protocol. Each session of the protocol is conducted in three steps: In the first step the connected processes exchange information about past sessions.[1] The second step is the attempt step. Each process uses the information it received in the first step to make an independent decision whether the current membership is eligible to be a primary component. If it is, the member attempts to form the session: it computes the session number, records the session and sends an attempt message to the rest of the members. In the last step, the processes form the new primary component: They declare the session as a primary component, and no longer record preceding sessions. The primary component formed in this step remains the primary component in the system until another membership change occurs.

If a process receives a membership message in the course of a session, it aborts the session and invokes a new session. Once the membership stabilizes, sessions are no longer aborted. If the expected messages fail to arrive from some of the members, then the primary component protocol is blocked until a membership change is reported.

Intuitively, the purpose of the attempt step is to guarantee that if a process p forms a session $\mathcal{F}$,[2] then all the other members of $\mathcal{F}$ recorded $\mathcal{F}$ as an ambiguous session. Thus, if some members of $\mathcal{F}$ detach before the last step, they will take $\mathcal{F}$ into account in future attempts to form a primary component.

[1]In case the membership protocol involves message exchange among the members, this information can be piggybacked onto the membership protocol messages, thus no extra communication round is needed.

[2]As a convention, we use the notation $\mathcal{F}$ to denote formed sessions.

```
1. Set $Is\_Primary$ to FALSE.
   Send your $Session\_Number$, $Ambiguous\_Sessions$, $Last\_Primary$ and $Last\_Formed$ to all the
   members of $\mathcal{M}$.

2. Attempt step: Upon receiving this information from all members of $\mathcal{M}$:
   • Update $Ambiguous\_Sessions$ according to the learning rules described in Figure 7.3.
   • Apply the resolution rules described in Figure 7.2.
   • Compute $Max\_Session$, $Max\_Primary$, and $All\_Ambiguous\_Sessions$.
   • if ($Next\_Quorum(Max\_Primary.M, \mathcal{M})$ and
     $((\forall S \in All\_Ambiguous\_Sessions)\ \ Next\_Quorum(S.M, \mathcal{M})))$
     then "attempt the session:"
       – Set $Session\_Number$ to $Max\_Session+1$.
       – Append to $Ambiguous\_Sessions$ the session $S = (\mathcal{M}, Session\_Number)$, with
         $S.A(q) = 0$ for every $q \in S.M$ s.t. $q \neq p$, and $S.A(p) = -1$.
       – Send an attempt message to every member of $\mathcal{M}$.
     else terminate this session with $Is\_Primary$=FALSE.

3. Form step: Upon receiving an attempt message from all members of $\mathcal{M}$ set:
   • $Last\_Primary = (\mathcal{M}, Session\_Number)$, and
   • $Ambiguous\_Sessions = \emptyset$, and
   • $Is\_Primary$=TRUE, and
   • $\forall q \in \mathcal{M}\ \ Last\_Formed_p(q) = Last\_Primary$.
```

Figure 7.1: A session of the protocol executed by process p.

In order to avoid recording an exponential number of ambiguous sessions, our protocol employs a "garbage collection" mechanism that reduces the number of ambiguous sessions recorded concurrently to be at most the number of processes in the system. A process deletes ambiguous sessions when it resolves their status, i.e., discovers whether they were formed by any member or not. In order to resolve a session, a process needs to learn about the session status at other members. The rules for learning and resolving ambiguous sessions are described in Section 7.4.4. These rules are employed during the attempt step. The protocol is formally described in Figure 7.1.

In each step of the protocol, when a process changes any of its private variables, it must write the change to a stable storage before responding to the message that caused the change. If the storage is destroyed because of a disk crash, the process may recover with its $Last\_Primary = (\infty, -1)$. This may limit the availability, but does not affect the correctness.

The Resolution Rules:

Adoption If $p \in \mathcal{F}.M$ and $Last\_Primary_p.N < \mathcal{F}.N$, and p learns that session $\mathcal{F}$ was formed by at least one of its members then:
Process p sets $Last\_Primary_p$ to $\mathcal{F}$ and $\forall q \in \mathcal{F}.M$ p sets $Last\_Formed_p(q) = \mathcal{F}$.

Deletion If p learns that an ambiguous session S was not formed by any of its members, or if p learns that a session $\mathcal{F}$, where $\mathcal{F}.N \geq S.N$ and $p \in \mathcal{F}.M$, was formed by at least one of its members, then:
Process p deletes S from $Ambiguous\_Sessions_p$.

Figure 7.2: The resolution rules.

7.4.4 Resolving Ambiguous Sessions

A process can either adopt or delete ambiguous sessions upon resolving their status. The resolution is based on determining whether an ambiguous session was formed by any of its members. If an ambiguous session was formed by some member, then the other members adopt this session: They declare the session as a primary component. On the other hand, if an ambiguous session was not formed by any of its members, then it is safe to delete it from *Ambiguous_Sessions*. Furthermore, if a process adopts a session S, (as described above), then it no longer records ambiguous sessions that precede S. The resolution rules are formally summarized in Figure 7.2.

Process p learns the status of process q w.r.t. session S during a later session S', where $p, q \in S.M \cap S'.M$, if during S' p executes the attempt step. Process p learns accordingly:

- If $Last\_Formed_q(p).N = S.N$ then p learns that process q formed session S.
- If $Last\_Formed_q(p).N < S.N$ then p learns that process q did not form session S.

Process p learns that session S was not formed by any of its members if:

- p doesn't form S, and learns from all the other session members that they did not form session S either, or
- There exist a later session S' and a process q, where $p, q \in S.M \cap S'.M$, such that during S', q does not consider S to be ambiguous or formed. Formally:
  - $Last\_Primary_q.N < S.N$ or ($Last\_Primary_q.N = S.N$ and $Last\_Primary_q.M \neq S.M$), and
  - $S \notin Ambiguous\_Sessions_q$, and
  - p executes the attempt step of the protocol during S'.

Figure 7.3: Learning rules.

The resolution rules require a process p to learn whether an ambiguous session that p records was formed by any of its members. This is achieved by collecting the session status from other session members during the first step of future sessions of the protocol. Process p applies the information it gathered to its *Ambiguous_Sessions* set during the attempt step.

A process p learns that a session S was formed by at least one of its members upon discovering that such a member exists. A process p learns that a session S was not formed by any of its members in one of two ways: First, by discovering that every member of S did not form S. Second, by discovering that another session member, q, does not consider S to be ambiguous, although q did not form S or any other session with a session number greater than S. This implies that either q did not even attempt to form S, or q already learned that none of the members of S formed S. The formal rules of learning are described in Figure 7.3.

7.5 Evaluating the Efficiency

Our protocol assumes a simple underlying membership protocol, which may be conducted in one communication round (e.g., [CS95]). Each session of our protocol is conducted in two communication rounds, one of which may be conducted by piggybacking information on the messages sent by the membership protocol. Thus, in each session of the protocol, a total of $2n$ messages are multicast by all the processes, where n is the number of processes participating in this session.

The protocol presented in Section 7.4 is symmetric: Processes multicast messages to all other processes. Such a protocol is efficient assuming a hardware broadcast/multicast mechanism. For networks in which efficient multicast is not available, it is straightforward to convert our protocol to work in a centralized fashion by appointing a coordinator for each session. The coordinator may be chosen deterministically, for example, the first member of the current membership in lexicographical order. In the centralized protocol, each process sends two messages to the coordinator, and the coordinator multicasts two messages to the other processes. Hence, a total of $4n$ point to point messages are sent. Once the membership stabilizes, our protocol terminates within one session, in which it resolves past ambiguous attempts and also forms a new primary component (if possible). As in the symmetric version of the protocol, the first round of messages may be piggybacked on membership protocol messages.

Below, we prove that a process records concurrently at most n ambiguous sessions in the worst case. In practice, the number of ambiguous sessions is expected to be small, since whenever a new primary component is successfully formed, all the ambiguous sessions are discarded.

In Lemma 7.5.4 we prove that if a process p attempts to form two ambiguous sessions with a process q, then during the later session p can learn q's status w.r.t. the former session. Note that after p learns a session's status as recorded by every session member, p can resolve the status of a session. Therefore, in case p cannot resolve a session's status, there is at least one session member with which p does not share a later attempt. This property linearly bounds the number of unresolved ambiguous sessions a process records concurrently, as we formally prove in Theorem 7.5.1.

Lemma 7.5.1 *At each process, the value of Session_Number is increased whenever the process attempts to form a session. Session_Number does not change at any other time.*

Proof: Immediate from the protocol. □

Lemma 7.5.2 *If a member p of a session $\mathcal{F}$ sets $Last\_Primary_p$ to $\mathcal{F}$ during session $\mathcal{F}$, then all members q of $\mathcal{F}$ appended $\mathcal{F}$ to $Ambiguous\_Sessions_q$ during this session.*

Proof: Process p sets $Last\_Primary_p$ to $\mathcal{F}$ during session $\mathcal{F}$ in Step 3 of the protocol, only if p received attempt messages from all members q of $\mathcal{F}$ indicating that they successfully executed Step 2 of the protocol during this session, i.e., appended $\mathcal{F}$ to $Ambiguous\_Sessions_q$. □

Lemma 7.5.3 *Process p sets $Last\_Primary_p$ to a session $\mathcal{F}$ during a session S, if either $\mathcal{F} = S$, or there exists a process q such that q set $Last\_Primary_q$ to $\mathcal{F}$ during $\mathcal{F}$.*

Proof: According to the protocol, a process p sets $Last\_Primary_p$ to $\mathcal{F}$ in one of two cases:

1. During session $\mathcal{F}$ in Step 3 of the protocol.
2. During a later session than $\mathcal{F}$, upon learning that another member q of $\mathcal{F}.M$ set $Last\_Formed_q(p)$ to $\mathcal{F}$ before p. Let r be the first member of $\mathcal{F}.M$ who set $Last\_Formed_r(p)$ to $\mathcal{F}$, then r set $Last\_Formed_r(p)$ to $\mathcal{F}$ by the first case, hence, during $\mathcal{F}$. □

Lemma 7.5.4 *Let p be a process and A_1, A_2 two ambiguous sessions, such that $A_1.N < A_2.N$ and both A_1 and A_2 are in $Ambiguous\_Sessions_p$. If there exists a process q such that $q \in A_1.M \cap A_2.M$, then p learned whether q formed session A_1 before p attempted to form session A_2.*

Proof: By induction on the difference $A_2.N - A_1.N$.

- Base case: $A_2.N - A_1.N = 1$. According to the protocol, a process attempts to form a session in Step 2 of the protocol, after the process received the *Last_Formed* arrays from all session members and applied the learning rules as follows:

  1. If $\mathit{Last\_Formed}_q(p).N < A_1.N$ then p learned that q did not form A_1.
  2. If $\mathit{Last\_Formed}_q(p).N = A_1.N$ then p learned that q formed A_1.

  Notice that $\mathit{Last\_Formed}_q(p).N > A_1.N$ is impossible; Otherwise, q formed $\mathit{Last\_Formed}_q(p)$, and therefore by Lemmas 7.5.3 and 7.5.2 process p attempted to form $\mathit{Last\_Formed}_q(p)$. Hence, by Lemma 7.5.1, $A_1.N < \mathit{Last\_Formed}_q(p).N < A_2.N$, in contradiction with *Session_Number* being an integer.

- General case: We assume the induction hypothesis holds for $A_2.N - A_1.N < k$, and prove for $A_2.N - A_1.N = k$. Since $A_2 \in \mathit{Ambiguous\_Sessions}_p$, p received $\mathit{Last\_Formed}_q(p)$ during session A_2, and learned as follows:

  1. If $\mathit{Last\_Formed}_q(p).N \leq A_1.N$ then, as in the base case, p learned whether q formed A_1.
  2. Otherwise, $\mathit{Last\_Formed}_q(p).N > A_1.N$. Hence, there exists a formed session F_i such that:
     - $A_1.N < F_i.N < A_2.N$, and
     - $p, q \in F_i.M$, and
     - q formed F_i.

     Since q formed F_i, then by Lemmas 7.5.3 and 7.5.2 $F_i \in \mathit{Ambiguous\_Sessions}_p$ upon ending F_i. Moreover, $F_i.N - A_1.N < k$. Hence, from the induction hypothesis, p learned whether q formed A_1 before attempting to form F_i, hence before attempting to form A_2. $\square$

Theorem 7.5.1 *Process p records concurrently at most $n - \mathit{Min\_Quorum} + 1$ ambiguous sessions, where n is the number of processes that participate in an execution of the protocol.*

Proof: Assume to the contrary that p records in $\mathit{Ambiguous\_Sessions}_p$ $n - \mathit{Min\_Quorum} + 2$ ambiguous sessions concurrently, $A_1, \ldots, A_{n-\mathit{Min\_Quorum}+2}$, such that $(\forall 1 \leq i < n - \mathit{Min\_Quorum} + 2)$ $A_i.N < A_{i+1}.N$[3]. Since A_i is still ambiguous, p did not learn whether some member of A_i formed

[3]By Lemma 7.5.1, the order requirement is always fulfilled.

it or not. Hence, by Lemma 7.5.4, there is at least one member of A_i that is not a member of any session $A_j \in \mathit{Ambiguous\_Sessions}_p$ for $j > i$. Consequently, for each i, there are at least i processes that do not participate in any session A_j where $j > i$. In particular, after recording sessions $A_1, \ldots, A_{n-\mathit{Min\_Quorum}+1}$, there are at least $n - \mathit{Min\_Quorum} + 1$ members that are not members of $A_{n-\mathit{Min\_Quorum}+2}$. Therefore $A_{n-\mathit{Min\_Quorum}+2}.M$ consists of less than $\mathit{Min\_Quorum}$ members, and $A_{n-\mathit{Min\_Quorum}+2}$ is not a legal session, a contradiction. □

7.6 Dynamically Changing Quorum Requirements

The definition of *Next_Quorum* presented in Section 7.4.1 requires every quorum to contain at least *Min_Quorum* members of $\mathcal{W}_0$ in order to always allow a group of more than $|\mathcal{W}_0| - \mathit{Min\_Quorum}$ members of $\mathcal{W}_0$ to make progress (provided that $\mathit{Min\_Quorum} > |\mathcal{W}_0|/2$). This requirement restricts the availability if some members of $\mathcal{W}_0$ leave the system. In this section we eliminate the special status of the members of $\mathcal{W}_0$: we always allow a group of more than $n - \mathit{Min\_Quorum}$ processes to make progress, where n is the "current" number of processes in the system. We present a novel mechanism for providing this feature, in environments which allow new processes to join on the fly.

Allowing n to change dynamically is subtle because the truth value of *Next_Quorum* changes with time. For example, $\mathit{Next\_Quorum}(S, T)$ may be initially TRUE because T contains more than $|\mathcal{W}_0| - \mathit{Min\_Quorum}$ members of $\mathcal{W}_0$, but later, as the set of participants increases, the truth value of $\mathit{Next\_Quorum}(S, T)$ may become FALSE. Therefore, n must be increased with care, and new processes may not immediately be taken into account. New processes are inserted into the "set of participants" using the two new variables described below:

$\mathcal{W}$ is the set of processes taken into account in the new *Min_Quorum* requirement. For members of $\mathcal{W}_0$, $\mathcal{W}$ is initialized to $\mathcal{W}_0$. For other processes, $\mathcal{W}$ is initially empty. New processes are inserted into this group when they participate in a formed session.

$\mathcal{A}$ is the set of processes that have not been admitted into $\mathcal{W}$ yet. $\mathcal{A}$ is initialized to the empty set if $p \in \mathcal{W}_0$, and otherwise to contain p itself.

These variables are used to evaluate the *Next_Quorum* predicate. Below we describe how these variables are maintained in the course of the primary component protocol (cf. Section 7.4). At the beginning of each step in a session S of the protocol, every process p executes the following operations:

1. In the first step, p sends $\mathcal{W}_p$ and $\mathcal{A}_p$ to every member of S.
2. *The Attempt Step:* Upon receiving responses from every member of S, p updates $\mathcal{W}_p$ and $\mathcal{A}_p$ as follows:
   (a) Set $\mathcal{W}_p$ to $\bigcup_{q \in S.M} \mathcal{W}_q$.
   (b) Set $\mathcal{A}_p$ to $(\bigcup_{q \in S.M} \mathcal{A}_q) \setminus \mathcal{W}_p$.

   The *Min_Quorum* requirement in the definition of the *Next_Quorum* predicate is evaluated as follows:

   - S is eligible to be a new primary component only if $|S.M \cap \mathcal{W}_p| \geq Min\_Quorum$.
   - If $|S.M \cap (\mathcal{W}_p \cup \mathcal{A}_p)| > |\mathcal{W}_p \cup \mathcal{A}_p| - Min\_Quorum$, then for every session S' that p records, the truth value of $Next\_Quorum(S', S)$ is TRUE, regardless of the system history.

   The first item replaces Requirement 1 in the definition of the *Next_Quorum* predicate in Section 7.4.1. The second item replaces Requirement 2(c) in the same definition.
3. *The Form Step:* Upon receiving an attempt message from every member of S:
   (a) Set $\mathcal{W}_p$ to $\mathcal{W}_p \cup (\mathcal{A}_p \cap S.M)$.
   (b) Set $\mathcal{A}_p$ to $\mathcal{A}_p \setminus S.M$.

This mechanism allows the system to adjust the quorum requirements in the protocol to the dynamically changing set of processes. We prove the correctness of the resulting protocol in the next section.

Jajodia and Mutchler [JM89] suggest a similar idea in their hybrid algorithm. The hybrid approach combines dynamic voting in large quorums with static voting in quorums of size three, ruling out quorums consisting of a single process. The resulting hybrid algorithm works as follows: Dynamic voting is employed in quorums that consist of at least four members. In a quorum Q of size three, static voting is employed: subsequent quorums contain at least two members of Q. The protocol returns to the dynamic voting scheme once a quorum containing at least four processes is formed.

The hybrid approach of [JM89] is comparable with the mechanism presented above when *Min_Quorum* is chosen to be two. In this case, neither approach is strictly better than the other. Note that the hybrid algorithm of [JM89] applies the hybrid approach to the algorithm of [JM90] which uses two phase commit to avoid inconsistencies.

7.7 Using Dynamic Voting in Conjunction with *COReL*

Our dynamic voting protocol may be used in conjunction with a variety of applications and services that make progress in primary components. The *COReL* protocol is an example of such a service. We now explain how our dynamic voting protocol may be exploited in conjunction with *COReL*.

The steps of the dynamic voting protocol should be performed together with steps of *COReL*, as follows:

1. The first step (information exchange) can be either piggybacked on membership messages or on messages sent by the Recovery Procedure of *COReL* described in Section 6.4.5.

2. The Attempt Step of the dynamic voting protocol should be performed as part of the Attempt Step of *COReL*'s algorithm for establishing a new primary component, described in Section 6.4.5.

3. The Form Step of the dynamic voting protocol should be performed as part of the Commit Step of *COReL*'s algorithm for establishing a new primary component, described in Section 6.4.5.

Part III

Theoretical Foundations

Chapter 8

Majority-Resilient Atomic Commit*

This chapter presents a new atomic commitment protocol, *enhanced three phase commit* (*E3PC*), that *always* allows a quorum in the system to make progress. Previously suggested quorum-based protocols (e.g., the quorum-based *three phase commit (3PC)* [Ske82]) allow a quorum to make progress in case of one failure. If failures cascade, however, and the quorum in the system is "lost" (i.e., at a given time no quorum component exists), a quorum can later become connected and still remain blocked. With E3PC, a connected quorum never blocks. E3PC is based on the quorum-based 3PC [Ske82], and it does not require more time or communication than 3PC. This protocol can be exploited in a replicated database setting, making the database *always* available to a majority of the sites.

8.1 Introduction

Reliability and availability of loosely coupled distributed database systems are becoming requirements for many installations, and fault tolerance is becoming an important aspect of distributed systems design. When sites crash, or when communication failures occur, it is desirable to allow as many sites as possible to make progress. A common way to increase the availability of data and services is *replication*. If data are replicated in several sites, they can still be available despite site and communication-link failures. Protocols for transaction management in distributed and replicated database systems need to be carefully designed in order to guarantee database consistency. This chapter presents a novel *atomic commitment protocol* (*ACP*) that *always* allows a majority (or quorum) to make progress. This protocol can be exploited in a replicated database setting, making the database *always* available to a majority of the sites.

In distributed and replicated database systems, when a transaction spans several sites, the database servers at all sites have to reach a common decision regarding whether the transaction should be committed or not. A mixed decision results in an inconsistent database, while a unani-

*This chapter is based on a paper by Keidar and Dolev [KD98].

mous decision guarantees the *atomicity* of the transaction (provided that the local server at each site can guarantee local atomicity of transactions). To this end an *atomic commitment protocol*, such as *two phase commit* (*2PC*) [Gra78] is invoked. The atomic commit problem and the two phase commit protocol are described in Section 8.3. Two phase commit is a *blocking* protocol: if the coordinator fails, all the sites may remain blocked indefinitely, unable to resolve the transaction.

To reduce the extent of blocking, Skeen suggested the quorum-based three phase commit (3PC) protocol, which maintains consistency in spite of network partitions [Ske82]. In case of failures, the algorithm uses a *quorum* (or *majority*)-based recovery procedure that allows a quorum to resolve the transaction. If failures cascade, however, and the quorum in the system is "lost" (i.e., at a certain time no quorum component exists), *a quorum of sites can become connected and still remain blocked.* Other previously suggested quorum-based protocols (e.g., [CR83, CK85]) also allow a quorum to make progress in case of one failure, while if failures cascade, a quorum can later become connected and still remain blocked. To our knowledge, the only previously suggested ACP that *always* allows a quorum to make progress is the ACP that we construct in [Kei94]. The protocol in [Kei94] is not straightforward; it uses a replication service as a building block, while the protocol presented in this chapter is easy to follow and self-contained.

This chapter presents the *enhanced three phase commit* (*E3PC*) protocol, which is an enhancement of the quorum-based 3PC [Ske82]. E3PC maintains consistency in the face of site failures and network partitions: sites may crash and recover, and the network may partition into several *components* and remerge. E3PC *always* allows a quorum to make progress: At any point in the execution of the protocol, if a group G of sites becomes connected, and this group contains a *quorum* and no subsequent failures occur for sufficiently long, then all the members of G eventually reach a decision. Furthermore, every site that can communicate with a site that has already reached a decision will also, eventually, reach a decision. An operational site that is not a member of a connected quorum may be *blocked*, i.e., may have to wait until a failure is repaired in order to resolve the transaction. This is undesirable but cannot be avoided; Skeen proved that every protocol that tolerates network partitions is bound to be blocking in certain scenarios [SS83].

E3PC achieves higher availability than 3PC simply by carefully maintaining two additional counters and with no additional communication. The principles demonstrated in this chapter can be used to increase the resilience of a variety of distributed services, e.g., replicated database systems, by ensuring that a quorum will always be able to make progress. Other protocols that use two counters in order to allow a majority to make progress are given in Chapters 6 and 7 and in [MHS89, CT96, Lam89, DLS88].

Numerous database replication schemes that are based on quorums have been suggested [Gif79, Her86, Her87, EASC85, EAT89]. These algorithms use *quorum systems* to determine when data objects are accessible. In order to guarantee the atomicity of transactions, these algorithms use an ACP and therefore are bound to block when the ACP they use blocks. Thus, with previously suggested ACPs, these approaches do not *always* allow a connected majority to update the database. Using E3PC these protocols can be made more resilient. Section 8.6 describes in detail how E3PC may be incorporated into *accessible copies* protocols [EASC85, EAT89], in order to make the database always available to a quorum.

E3PC uses a *perfect* failure detector (cf. Section 2.3): Every site has accurate information regarding which sites are connected to it. Section 8.7 discusses the ability of E3PC to work with unreliable failure detectors. In this case, the protocol solves the *weak atomic commit* problem [Gue95].

The rest of this chapter is organized as follows: Section 8.2 presents the computation model. Section 8.3 provides general background on the atomic commitment problem. The quorum-based three phase commit protocol [Ske82] is described in Section 8.4, and enhanced three phase commit is described in Section 8.5. Section 8.6 describes how E3PC can be exploited in replicated database systems. Section 8.7 describes the protocol's behavior with an *unreliable* failure detector. The discussion in Section 8.8 concludes this chapter. In Appendix A.4 the correctness of E3PC is formally proven.

8.2 The Model

Our protocol is applicable in an asynchronous message-passing environment, as described in Chapter 2. The set of sites running the protocol is fixed and is known to all the sites. Failures are detected using a perfect fault detector, defined in Section 2.3. This assumption is weakened in Section 8.7.

8.3 Background – Distributed Transaction Management

This section provides general background on the atomic commit problem and protocols.

8.3.1 Problem Definition

A distributed transaction is composed of several subtransactions, each running on a different site. The database manager at each site can unilaterally decide to ABORT the local subtransaction, in which case the entire transaction must be aborted. If all the participating sites agree to COMMIT their subtransaction (vote **Yes** on the transaction) and no failures occur, the transaction should be committed. It is assumed that the local database server at each site can atomically execute the subtransaction once it has agreed to COMMIT it.

In order to ensure that all the subtransactions are consistently committed or aborted, the sites run an *atomic commitment protocol* such as *two phase commit.* The requirements of atomic commitment (as defined in Chapter 7 of [BHG87]) are as follows:

AC1: **Uniform Agreement:** All the sites that reach a decision reach the same one.

AC2: A site cannot reverse its decision after it has reached one.

AC3: **Validity:** The COMMIT decision can be reached only if all sites voted **Yes**.

AC4: **Non-triviality:** If there are no failures and all sites voted **Yes**, then the decision will be to COMMIT.

AC5: **Termination:** At any point in the execution of the protocol, if all existing failures are repaired and no new failures occur for sufficiently long, then all sites will eventually reach a decision.

8.3.2 Two Phase Commit

The simplest and most renowned ACP is *two phase commit* [Gra78]. Several variations of 2PC have been suggested (e.g., presume abort and presume commit [MLO86]), the simplest version is centralized – one of the sites is designated as the *coordinator.* The coordinator sends a transaction (or request to prepare to commit) to all the participants. Each site answers by a **Yes** ("ready to commit") or by a **No** ("abort") message. If any site votes No, all the sites abort. The coordinator collects all the responses and informs all the sites of the decision. In absence of failures, this protocol preserves atomicity. Between the two phases, each site *blocks*, i.e., keeps the local database locked, waiting for the final word from the coordinator. If a site fails before its vote reaches the coordinator, it is usually assumed that it had voted No. If the coordinator fails in the first phase, all the sites remain blocked indefinitely, unable to resolve the last transaction. The centralized version of 2PC is depicted in Figure 8.1.

Commit protocols may also be described using state diagrams [SS83]. The state diagram for 2PC is shown in Figure 8.1. The circles denote states; final states are double-circled. The arcs

Coordinator	**Participant**
Transaction is received: Send sub-transactions.	
	Sub-transaction is received: Send reply – **Yes** or **No**.
If all sites respond **Yes**: Send COMMIT. If some site voted **No**: Send ABORT.	
	COMMIT or ABORT is received: Process accordingly.

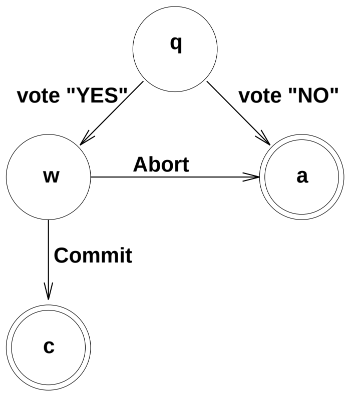


Figure 8.1: The centralized two phase commit protocol.

represent state transitions, and the *action* taken (e.g., message sent) by the site is indicated next to each arc. In this protocol, each site (either coordinator or participant) can be in one of four possible states:

q : INITIAL state – A site is in the initial state until it decides whether to unilaterally abort or to agree to commit the transaction.

w : WAIT state – In this state the coordinator waits for votes from all of the participants, and each participant waits for the final word from the coordinator. This is the "uncertainty period" for each site, when it does not know whether the transaction will be committed or not.

c : COMMIT state – The site knows that a decision to commit was made.

a : ABORT state – The site knows that a decision to abort was made.

The states of a commit protocol may be classified along two orthogonal lines. In the first dimension, the states are divided into two disjoint subsets: The *committable* states and the *non-committable* states. A site is in a committable state only if it knows that all the sites have agreed to proceed with the transaction. The rest of the states are *non-committable.* The only committable state in 2PC is the COMMIT state. The second dimension distinguishes between *final* and *non-final* states. The *final* states are the ones in which a decision has been made and no more state transitions are possible. The final states in 2PC are COMMIT and ABORT.

8.3.3 Quorums

In order to reduce the extent of blocking in replication and atomic commit protocols, *majority* votes or *quorums* are often used. A *quorum system* is a generalization of the majority concept. E3PC, like Skeen's quorum-based three phase commit protocol [Ske82], uses a quorum system to decide when a group of connected sites may resolve the transaction. To enable maximum flexibility the quorum system may be elected in a variety of ways (e.g., weighted voting [Gif79]). The quorum system is *static*; it does not change in the course of the protocol.

The predicate $Q(S)$ is TRUE for a given subset S of the sites iff S is a quorum. The requirement from this predicate is that for any two sets of sites S and S' such that $S \cap S' = \emptyset$, at most one of $Q(S)$ and $Q(S')$ holds, i.e., every pair of quorums intersect. For example, in the simple majority quorum system $Q(S)$ is TRUE iff $|S| > n/2$, where n is the total number of sites running the protocol. Numerous quorum systems that fulfill these criteria were suggested. An analysis of the availability of different quorum systems may be found in [PW95].

For further flexibility, it is possible to set different quorums for commit and abort (this idea was presented in [Ske82]). In this case, a *commit quorum* of connected sites is required in order to commit a transaction, and an *abort quorum* is required to abort. For example, to increase the probability of commit in the system, one can assign smaller quorums for commit and larger ones for abort.

In this case, the quorum system consists of two predicates: $Q_C(G)$ is TRUE for a given group of sites G iff G is a commit quorum, and $Q_A(G)$ is TRUE iff G is an abort quorum. The requirement from these predicates is that for any two groups of sites G and G' such that $G \cap G' = \emptyset$, at most one of $Q_C(G)$ and $Q_A(G')$ holds, i.e., every commit quorum intersects every abort quorum.

8.3.4 The Extent of Blocking in Commit Protocols

The 2PC protocol is an example of a *blocking* protocol: operational sites sometimes wait on the recovery of failed sites. Locks must be held in the database while the transaction is blocked. Even though blocking preserves consistency, it is highly undesirable because the locks acquired by the blocked transaction cannot be relinquished, rendering the data inaccessible by other requests. Consequently, the availability of data stored in reliable sites can be limited by the availability of the weakest component in the distributed system.

Skeen *et al.* [SS83] proved that there exists no non-blocking protocol resilient to network partitioning. When a partition occurs, the best protocols allow no more than one group of sites to continue while the remaining groups block. Skeen suggested the quorum-based three phase commit protocol, which maintains consistency in spite of network partitions [Ske82]. This protocol is blocking in case of partitions; it is possible for an operational site to be blocked until a failure is mended. In case of failures, the algorithm uses a *quorum* (or *majority*)-based recovery procedure that allows a quorum to resolve the transaction. If failures cascade, however, *a quorum of sites can become connected and still remain blocked.* Skeen's quorum-based commit protocol is described in Section 8.4.

Since completely non-blocking recovery is impossible to achieve, further research in this area concentrated on minimizing the number of blocked sites when partitions occur. Chin *et al.* [CR83] define *optimal termination protocols (recovery procedures)* in terms of the average number of sites that are blocked when a partition occurs. The average is over all the possible partitions, and all the possible states in the protocol in which the partitions occurs. The analysis deals only with states in the basic commit protocol and ignores the possibility for cascading failures (failures that occur during the recovery procedure). It is proved that any ACP with optimal recovery procedures takes at least three phases and that the quorum-based recovery procedures are optimal.

In [Kei94] we construct an ACP that always allows a connected majority to proceed, regardless of past failures. To our knowledge, no other ACP with this feature was suggested. The ACP suggested in [Kei94] uses a reliable replication service as a building block and is mainly suitable for replicated database systems. This chapter presents a novel commitment protocol, *enhanced three phase commit*, which always allows a connected majority to resolve the transaction (if it remains connected for sufficiently long). E3PC does not require complex building blocks, such as the one in [Kei94], and is more adequate for partially replicated or non-replicated distributed database systems; it is based on the quorum-based three phase commit [Ske82].

8.4 Quorum-Based Three Phase Commit

This section describes Skeen's quorum-based commit protocol [Ske82]. E3PC is a refinement of 3PC, and therefore we elaborate on 3PC before presenting E3PC. The basic *three phase commit* is described in Section 8.4.1, and the recovery procedure is described in Section 8.4.2. In Section 8.4.3 it is shown that with 3PC a connected majority of the sites can be blocked. A simplified version of 3PC that uses the same quorums for commit and abort is presented.

8.4.1 Basic Three Phase Commit

The 3PC protocol is similar to two phase commit, but in order to achieve resilience, another non-final "buffer state" is added in 3PC, between the WAIT and the COMMIT states:

Coordinator	**Participant**
Transaction is received: Send sub-transactions to participants.	
	Sub-transaction is received: Send reply – **Yes** or **No**.
If all sites respond **Yes**: Send PRE-COMMIT. If any site voted **No**: Send ABORT.	
	PRE-COMMIT received: Send ACK to coordinator.
Upon receiving a quorum of **ACKs**: Send COMMIT. Otherwise: Block (wait for more votes or until recovery)	
	COMMIT or ABORT is received: Process the transaction accordingly.

Figure 8.2: The quorum-based three phase commit protocol.

pc : PRE-COMMIT state – this is an intermediate state before the commit state and is needed to allow for recovery. In this state the site is still in its "uncertainty period."

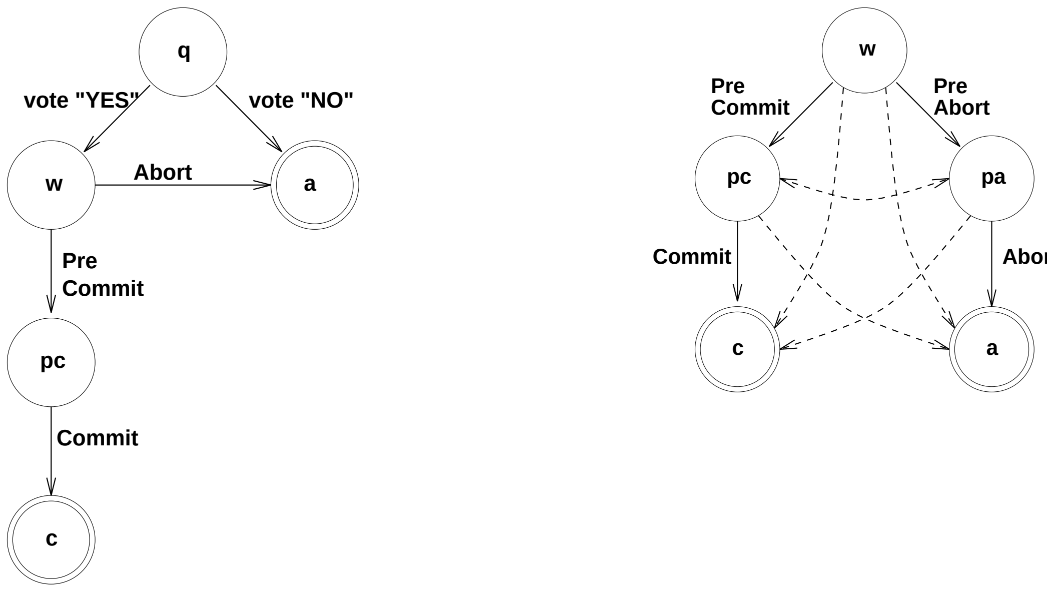
The quorum-based 3PC is described in Figure 8.2, and a corresponding state diagram is depicted in Figure 8.3(a). The COMMIT and PRE-COMMIT states of 3PC are *committable* states; a site may be in one of these states only if it knows that all the sites have agreed to proceed with the transaction. The rest of the states are *non-committable.* In each step of the protocol, when the sites change their state, they must write the new state to *stable storage* before replying to the message that caused the state change.

8.4.2 Recovery Procedure for Three Phase Commit

When a group of sites detect a failure (a site crash or a network partition) or a failure repair (site recovery or merge of previously disconnected network components), they run the recovery procedure in order to try to resolve the transaction (i.e., commit or abort it). The recovery procedure consists of two phases: first elect a new coordinator, and next attempt to form a quorum that can resolve the transaction.

A new coordinator may be elected in different ways (e.g., [GM82]). In the course of the election, the coordinator hears from all the other participating sites. If there are failures (or recoveries) in the course of the election, the election can be restarted.

The new coordinator tries to reach a decision whether the transaction should be committed or



(a) The Basic Three Phase Commit (b) The Recovery Procedure

Figure 8.3: Three phase commit and the recovery procedure.
q: INITIAL state; **w:** WAIT; **pc:** PRE-COMMIT; **c:** COMMIT; **pa:** PRE-ABORT; **a:** ABORT.

not and tries to form a quorum for its decision. The protocol must take the possibility of failures and failure repairs into account and, furthermore, must take into account the possibility of two (or more) different coordinators existing concurrently in disjoint network components. In order to ensure that the decision will be consistent, a coordinator must explicitly establish a quorum for a COMMIT or an ABORT decision. To this end, in the recovery procedure, another state is added:

pa : PRE-ABORT state. Dual state to PRE-COMMIT.

The recovery procedure is described in Figure 8.4. The state diagram for the recovery procedure is shown in Figure 8.3(b). The dashed lines represent transitions in which this site's state was not used in the decision made by the coordinator. Consider for example the following scenario: site p_1 reaches the PRE-ABORT state during an unsuccessful attempt to abort. The network then partitions, and p_1 remains blocked in the PRE-ABORT state. Later, a quorum (that does not include p_1) is formed, and another site, p_2, decides to COMMIT the transaction (this does not violate consistency, since the attempt to abort has failed). If now p_1 and p_2 become connected, the coordinator must decide to COMMIT the transaction, because p_2 is COMMITTED already. Therefore, p_1 makes a transition from PRE-ABORT to COMMIT.

After collecting the states from all the sites, the coordinator tries to decide how to resolve the transaction. If any site has previously committed or aborted, then the transaction is immediately

1. Elect a new coordinator, r.
2. The coordinator, r, collects the states from all the connected sites.
3. The coordinator tries to reach a decision, as described in Figure 8.5. The decision is computed using the states collected so far. The coordinator multicasts a message reflecting the decision.
4. Upon receiving a PRE-COMMIT or PRE-ABORT each participant sends an ACK to r.
5. Upon receiving a quorum of ACKs for PRE-COMMIT or PRE-ABORT, r multicasts the corresponding decision: COMMIT or ABORT.
6. Upon receiving a COMMIT or an ABORT message: Process the transaction accordingly.

Figure 8.4: The quorum-based recovery procedure for three phase commit.

Collected States	Decision
$\exists$ ABORTED	ABORT
$\exists$ COMMITTED	COMMIT
$\exists$ PRE-COMMITTED $\wedge$ Q(sites in WAIT and PRE-COMMIT states)	PRE-COMMIT
Q(sites in WAIT and PRE-ABORT states)	PRE-ABORT
Otherwise	BLOCK

Figure 8.5: The decision rule for the quorum-based recovery procedure.

committed or aborted accordingly. Otherwise, the coordinator attempts to establish a quorum. A COMMIT is possible if at least one site is in the PRE-COMMIT state and the group of sites in the WAIT state together with the sites in the PRE-COMMIT state form a quorum. An ABORT is possible if the group of sites in the WAIT state together with the sites in the PRE-ABORT state form a quorum. The decision rule is summarized in Figure 8.5.

8.4.3 Three Phase Commit Blocks a Quorum

In this section it is shown that in the algorithm described above, it is possible for a quorum to become connected and still remain blocked. In our example, there are three sites executing the transaction: p_1, p_2, and p_3. The quorum system used is a simple majority: every two sites form a quorum. Consider following the scenario depicted in Figure 8.6:

p_1 is the coordinator. All the sites vote **Yes** on the transaction. p_1 receives and processes the votes, but p_2 and p_3 detach from p_1 before receiving the PRE-COMMIT message sent by p_1.

p_2 is elected as the new coordinator. It sees that both p_2 and p_3 are in the WAIT state and therefore sends a PRE-ABORT message, according to the decision rule. p_3 receives the PRE-ABORT message, acknowledges it, and then detaches from p_2.

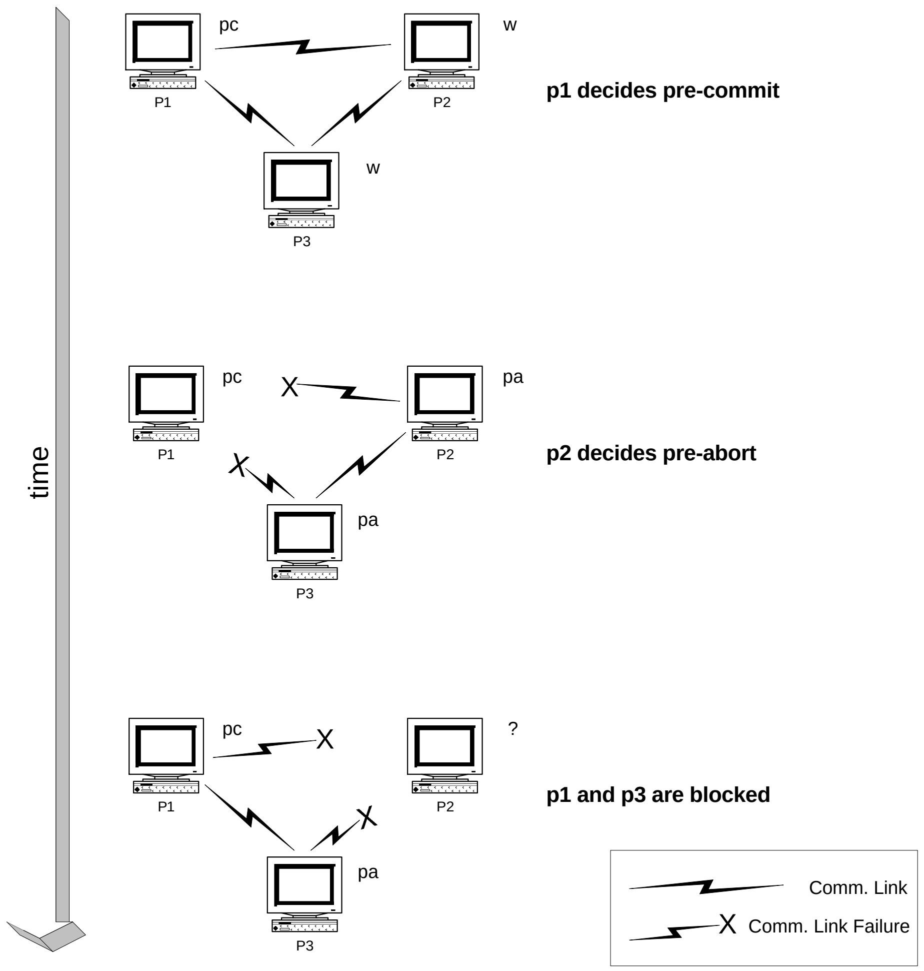


Figure 8.6: Three phase commit blocks a quorum.

Now, p_3 is in the PRE-ABORT state, while p_1 is in the PRE-COMMIT state. If now p_1 and p_3 become connected, then according to the decision rule, they remain BLOCKED, even though they form a quorum.

Analysis

In this example, it is actually safe for p_1 and p_3 to decide PRE-ABORT, because none of the sites could have committed, but it is *not* safe for them to decide PRE-COMMIT, because p_3 cannot know whether p_2 has aborted or not.

Observe that p_3 decided PRE-ABORT "after" p_1 decided PRE-COMMIT, and therefore the PRE-COMMIT decision made by p_1 is "stale", and no site has actually reached a COMMIT decision following it, because otherwise, it would have been impossible for p_2 to reach a PRE-ABORT decision.

The 3PC protocol does not allow a decision in this case, because the sites have no way of knowing which decision was made "later." Had the sites known that the a PRE-ABORT decision was made "later," they could have decided PRE-ABORT again and would have eventually ABORTED the transaction. E3PC provides the mechanism for doing exactly this.

8.5 The E3PC Protocol

This section presents a three phase atomic commitment protocol, *enhanced three phase commit*, with a novel quorum-based recovery procedure that always allows a quorum of sites to resolve the transaction, even in the face of cascading failures. The protocol is based on the quorum-based three phase commit protocol [Ske82]. E3PC does not require more communication or time than 3PC; the improved resilience is achieved simply by maintaining two additional counters, which impose a *linear order* on quorums formed in the system.

Initially, the basic E3PC is invoked. If failures occur, the sites invoke the recovery procedure and elect a new coordinator. The new coordinator carries on the protocol to reach a decision. If failures cascade, the recovery procedure may be reinvoked an arbitrary number of times. Thus, one *execution* of the protocol (for one transaction) consists of one *invocation* of the *basic E3PC* and of zero or more *invocations* of the recovery procedure.

Section 8.5.1 describes how E3PC enhances 3PC. The recovery procedure for E3PC is described in Section 8.5.2. In Section 8.5.3 it is shown that E3PC does not block a quorum in the example of Section 8.4.3. Section 8.5.4 outlines the correctness proof for E3PC. First, a simplified version of E3PC that uses the same quorums for commit and abort is presented. In Section 8.5.5 a more general version of E3PC, which uses different quorums for commit and abort is described.

8.5.1 E3PC: Enhancing Three Phase Commit

The basic E3PC is similar to the basic 3PC, the only difference being that E3PC maintains two additional counters, as follows: In each invocation of the recovery procedure, the sites try to elect a new coordinator. The coordinators elected in the course of an execution of the protocol are sequentially numbered: A new "election number" is assigned in each invocation of the recovery procedure. Note that there is no need to elect a new coordinator in each invocation of the basic 3PC or E3PC; the re-election is needed only in case failures occur. The coordinator of the basic E3PC is assigned "election number" one, even though no elections actually take place. The following two counters are maintained by the basic E3PC and by the recovery procedure:

Last_Elected - The number of the last election that this site took part in. This variable is updated when a new coordinator is elected. This value is initialized to *one* when the basic E3PC is invoked.

Last_Attempt - The election number in the last attempt this site made to commit or abort. The coordinator changes this variable's value to the value of *Last_Elected* whenever it makes a decision. Every other participant sets its *Last_Attempt* to *Last_Elected* when it moves to the PRE-COMMIT or to the PRE-ABORT state, following a PRE-COMMIT or a PRE-ABORT message from the coordinator. This value is initialized to *zero* when the basic E3PC is invoked.

These variables are logged on stable storage. The second counter, *Last_Attempt*, provides a *linear order* on PRE-COMMIT and PRE-ABORT decisions; e.g., if some site is in the PRE-COMMIT state with its *Last_Attempt* $= 7$, and another site is in the PRE-ABORT state with its *Last_Attempt* $= 8$, then the PRE-COMMIT decision is "earlier" and therefore "stale," and the PRE-ABORT decision is safe. The first counter, *Last_Elected*, is needed to guarantee the uniqueness of the *Last_Attempt*, [1] i.e., that two different attempts will not be made with the same value of *Last_Attempt* (cf. Lemma A.4.3 in Appendix A.4).

Notation

The following notation is used:

- $\mathcal{P}$ is the group of sites that are live and connected, and which take part in the election of the new coordinator.
- *Max_Elected* is $\max_{p\in\mathcal{P}}$(*Last_Elected* of p).
- *Max_Attempt* is $\max_{p\in\mathcal{P}}$(*Last_Attempt* of p).
- *Is_Max_Attempt_Committable* is a predicate that is TRUE iff all the members that are in non-final states and whose *Last_Attempt* is equal to *Max_Attempt* are in a committable state (i.e., in the PRE-COMMIT state). Formally, *Is_Max_Attempt_Committable* is TRUE iff $\forall_{p\in\mathcal{P}}$(*Last_Attempt* of $p =$ *Max_Attempt* $\wedge p$ is in a non-final state $\rightarrow p$ is in a committable state)

8.5.2 Quorum-Based Recovery Procedure

As in 3PC, the recovery procedure is invoked when failures are detected and when failures are repaired. Sites cannot "join" the recovery procedure in the middle, instead, the recovery procedure must be reinvoked to let them take part.

[1] The value of *Last_Elected* is not guaranteed to be unique, two elections may be made with the same value of *Last_Elected*, in case the first election with this number did not terminate successfully at all the members. Also note that the same coordinator can not be chosen with the same election number twice.

1. Elect a new coordinator r. The election is non-blocking, it is restarted in case of failure. In the course of the election, r hears from all the other sites their values of *Last_Elected* and *Last_Attempt* and determines *Max_Elected* and *Max_Attempt*. r sets *Last_Elected* to *Max_Elected*+1 and notifies the sites in $\mathcal{P}$ of its election, and of the value of *Max_Elected*.
2. Upon hearing *Max_Elected* from r, set *Last_Elected* to *Max_Elected*+1 and send local state to the coordinator r.
3. The coordinator, r collects states from the other sites in $\mathcal{P}$, and tries to reach a decision as described in Figure 8.8. The decision is computed using the states collected so far; the subset of sites from which r received the state so far is denoted by $\mathcal{S}$. Upon reaching a decision other than BLOCK, r sets *Last_Attempt* to *Last_Elected*, and multicasts the decision to all the sites in $\mathcal{P}$.
4. Upon receiving a PRE-COMMIT or PRE-ABORT each participant sets its *Last_Attempt* to *Last_Elected* and sends an ACK to r.
5. Upon receiving a quorum of ACKs for PRE-COMMIT (PRE-ABORT), r multicasts the decision: COMMIT (ABORT).
6. Upon receiving a COMMIT (ABORT) message from r: process the transaction accordingly.

Figure 8.7: The recovery procedure for E3PC.

All the messages sent by the protocol carry the election number (*Last_Elected*) and process id of the coordinator. Thus, it is possible to know in which invocation of the protocol each message was sent. A site that hears from a new coordinator ceases to take part in the previous invocation that it took part in and no longer responds to its previous coordinator. Messages from previous invocations are ignored. Thus, a site cannot concurrently take part in two invocations of the recovery procedure. Furthermore, if a site responds to messages from the coordinator in some invocation, it necessarily took part in the election of that coordinator.

The recovery procedure for E3PC is similar to the quorum-based recovery procedure described in Section 8.4.2. As in 3PC, in each step of the recovery procedure, when the sites change their state, they must write the new state to stable storage before replying to the message that caused the state change. The recovery procedure is described in Figure 8.7. The possible state transitions in E3PC and its recovery procedure are the same as those of 3PC, depicted in Figure 8.3; the improved performance in E3PC results from the decision rule, which allows state transitions in more cases.

In Step 3 of the recovery procedure, r collects the states from the other sites in $\mathcal{P}$ and tries to reach a decision. The sites are blocked until r receives enough states to allow a decision. It is possible to reach a decision before collecting the states from all the sites in $\mathcal{P}$; e.g., when a final state is received, a decision can be made. It is also possible to reach a decision once states are collected from a quorum, if one of the quorum members has *Last_Attempt*=*Max_Attempt*. The subset of $\mathcal{P}$ from which r received the state so far is denoted by $\mathcal{S}$; r constantly tries to compute the decision using the states in $\mathcal{S}$, whenever new states arrive and until a decision is reached. The decision rule is described below. If the decision is not BLOCK, r changes *Last_Attempt* to *Last_Elected*, and multicasts the decision to all the sites in $\mathcal{P}$.

Decision Rule

Collected States	Decision
$\exists$ ABORTED	ABORT
$\exists$ COMMITTED	COMMIT
Is_Max_Attempt_Committable $\wedge Q(\mathcal{S})$	PRE-COMMIT
$\neg$*Is_Max_Attempt_Committable* $\wedge Q(\mathcal{S})$	PRE-ABORT
Otherwise	BLOCK

Figure 8.8: The decision rule for E3PC.

The coordinator collects the states from the live members of $\mathcal{P}$ and applies the following decision rule to the subset $\mathcal{S}$ of sites from which it received the state.

- If there exists a site (in $\mathcal{S}$) that is in the ABORTED state – ABORT.
- If there exists a site in the COMMITTED state – COMMIT.
- If *Is_Max_Attempt_Committable* is TRUE, and $\mathcal{S}$ is a quorum – PRE-COMMIT.
- If *Is_Max_Attempt_Committable* is FALSE and $\mathcal{S}$ is a quorum – PRE-ABORT.
- Otherwise – BLOCK.

The decision rule is summarized in Figure 8.8. It is easy to see that with the new decision rule, if a group of sites is a quorum, it will never be blocked.

8.5.3 E3PC does not Block a Quorum

In E3PC, if a group of sites forms a quorum, it will never be blocked. This is obvious from the decision rule: if some site has previously committed (aborted), then the decision is COMMIT (ABORT). Otherwise, a decision can always be made according to the value of *Is_Max_Attempt_Committable*.

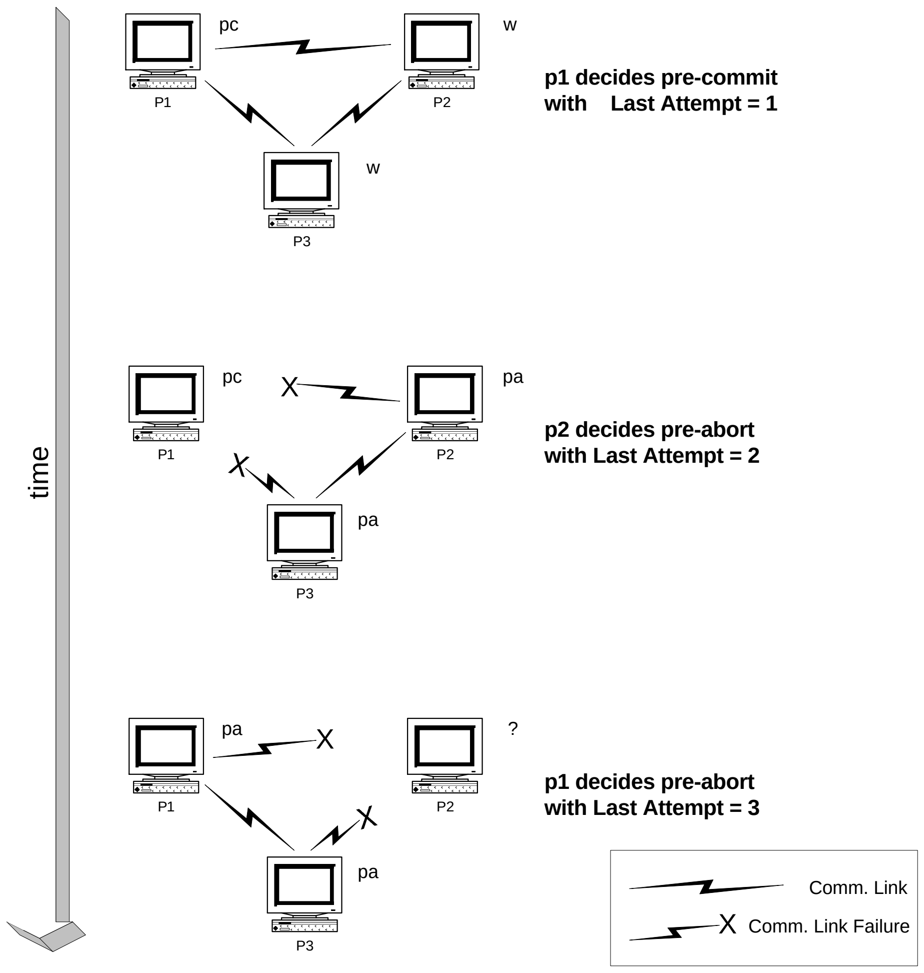

pc
w
P1
P2
w
P3
p1 decides pre-commit with Last Attempt = 1
pc
pa
P1
P2
pa
P3
p2 decides pre-abort with Last Attempt = 2
time
pa
?
P1
P2
pa
P3
p1 decides pre-abort with Last Attempt = 3
Comm. Link
Comm. Link Failure

Figure 8.9: E3PC does not block a quorum.

We now demonstrate that E3PC does not block with the scenario of Section 8.4.3 (in which Skeen's quorum-based 3PC does block). In this example, there are three sites executing the transaction - p_1, p_2, and p_3 - and the quorum system is a simple majority: every two sites form a quorum. The following scenario, depicted in Figure 8.9, was considered:

- Initially, p_1 is the coordinator. All the sites vote **Yes** on the transaction. p_1 receives and processes the votes, but p_2 and p_3 detach from p_1 before receiving the PRE-COMMIT message sent by p_1. Now $Last\_Attempt_{p_1}$ is 1 while $Last\_Attempt_{p_2} = Last\_Attempt_{p_3} = 0$, and the value of *Last_Elected* is one for all the sites.

- p_2 is elected as the new coordinator, and the new *Last_Elected* is two. It sees that both p_2 and p_3 are in the WAIT state and therefore sends a PRE-ABORT message, according to the decision rule, and moves to the PRE-ABORT state while changing its *Last_Attempt* to two. p_3 receives the PRE-ABORT message, sets its *Last_Attempt* to two, sends an acknowledgment, and detaches from p_2.

- Now, p_3 is in the PRE-ABORT state with its value of *Last_Attempt* $= 2$, while p_1 is in the PRE-COMMIT state with its *Last_Attempt* $= 1$. If now p_1 and p_3 become connected, then, according to the decision rule, they decide to PRE-ABORT the transaction, and they do *not* remain blocked.

8.5.4 Correctness of E3PC

In Appendix A.4 it is formally proven that E3PC fulfills the requirements of atomic commitment described in Section 8.3.1. This section outlines the proof.

First it is proven that two contradicting attempts (i.e., PRE-COMMIT and PRE-ABORT) cannot be made with the same value of *Last_Attempt* (Lemma A.4.3). This is true due to the fact that every two quorums intersect and that a quorum of sites must increase *Last_Elected* before a PRE-COMMIT or a PRE-ABORT decision. Moreover, *Last_Attempt* is set to the value of *Last_Elected*, which is higher than the previous value of *Last_Elected* of all the participants of the recovery procedure. Next, it is proven that the value of *Last_Attempt* at each site increases every time the site changes state from a committable state to a non-final non-committable state, and vice versa (Lemma A.4.5).

Using the two lemmas above the following is proven (Lemmas A.4.6 and A.4.8): If the coordinator reaches a COMMIT (ABORT) decision upon receiving a quorum of ACKs for PRE-COMMIT (PRE-ABORT) when setting its *Last_Attempt* to i, then for every $j \geq i$ no coordinator will decide PRE-ABORT (PRE-COMMIT) when setting its *Last_Attempt* to j. These lemmas are proven by induction on $j \geq i$; it is shown, by induction on j, that if some coordinator r sets its *Last_Attempt* to j in Step 3 of the recovery procedure, then *Is_Max_Attempt_Committable* is TRUE (FALSE) in this invocation of the recovery procedure, and therefore, the decision is PRE-COMMIT (PRE-ABORT).

One concludes that if some site running the protocol COMMITS the transaction, then no other site ABORTS the transaction.

8.5.5 Using Different Quorums for Commit and Abort

This section describes how to generalize E3PC to work with different quorums for commit and abort. Commit and abort quorums are described in Section 8.3.3. The following changes need to be made in the protocol:

Collected States	Decision
$\exists$ ABORTED	ABORT
$\exists$ COMMITTED	COMMIT
Is_Max_Attempt_Committable $\wedge Q_C(\mathcal{S})$	PRE-COMMIT
$\neg$*Is_Max_Attempt_Committable* $\wedge Q_A(\mathcal{S})$	PRE-ABORT
Otherwise	BLOCK

Figure 8.10: The decision rule for E3PC with commit and abort quorums.

1. In the second phase of the basic E3PC, the coordinator waits for a commit quorum of ACKs before sending PRE-COMMIT.

2. In Step 5 of the recovery procedure, the coordinator needs to wait for a commit quorum of ACKs in order to PRE-COMMIT, and for ACKs from an abort quorum in order to PRE-ABORT.

3. Likewise, the decision rule is slightly changed to require a commit quorum in order to PRE-COMMIT (in case *Is_Max_Attempt_Committable* is TRUE) and an abort quorum in order to PRE-ABORT (if *Is_Max_Attempt_Committable* is FALSE). The resulting decision rule is shown in Figure 8.10.

It is easy to see from the new decision rule that if a group of processes is both a commit quorum and an abort quorum, it does not remain blocked.

The correctness proof of the general version of E3PC is similar to the correctness proof of E3PC presented in this chapter; it uses the property that every commit quorum intersects every abort quorum in order to prove that two contradicting attempts (i.e., PRE-COMMIT and PRE-ABORT) cannot be made with the same value of *Last_Attempt*. The formal proof may be found in [KD94].

8.6 Replicated Database Systems

In replicated database systems, the sites continuously execute transactions. When the network partitions, it is often desirable to allow a quorum of the sites to access the database, but it is usually undesirable to allow sites in two disjoint network components to concurrently update the same data. Numerous replication schemes that are based on quorums have been suggested [Gif79, Her86, Her87, EASC85, EAT89]. In order to guarantee the atomicity of transactions, these algorithms use an ACP and therefore are bound to block when the ACP they use blocks. We propose to use E3PC in conjunction with these protocols in order to make the database *always* available to a quorum.

The same quorum system should be used to determine when the data are accessible to a group of sites as for the atomic commitment protocol. In a fully replicated database, a group of sites needs to be a quorum of the total number of sites in order to access the database. Hence, in order to resolve a transaction using the E3PC recovery procedure, a group of sites needs to be a quorum of the total number of sites and not just of the sites that invoked E3PC for the specific transaction.

If the data are *partially replicated*, then for each item accessed by this transaction, a quorum of the sites it resides on is required. In order to resolve a transaction using the E3PC recovery procedure, a group of sites needs to contain a quorum for each item accessed by this transaction.

There is a subtle point to consider with this solution: sites that did not take part in the basic E3PC for this transaction may take part in the recovery procedure. The local databases at such sites are not up-to-date, since they do not necessarily reflect the updates performed by the current transaction. Therefore, these sites need to recover the database state from other sites during the merge and before taking part in the recovery procedure. In the *accessible copies* protocols [EASC85, EAT89], this is done every time the view changes. In this case, we suggest using the view change as the failure detector for E3PC; thus, the recovery procedure is always invoked following a view change, after all the participating sites have reached an up-to-date state. Below, we describe in detail how E3PC may be incorporated into *accessible copies* protocols.

8.6.1 Using E3PC with Accessible Copies Protocols

Accessible copies protocols [EASC85, EAT89] maintain a *view* of the system to determine when data are accessible: A data item can be read/written within a view (component) only if a majority of its read/write votes are assigned to copies that reside on sites that are members of this view. This majority of votes is the "accessibility threshold for the item," not to be confused with read and write quorums used within the current view. In order to guarantee the *atomicity* of each transaction, these protocols use an ACP. I propose to use E3PC as this ACP using these accessibility thresholds as its quorum system. This way the sites that succeed in resolving the previous transaction are also allowed to access the database in new transactions.

A group of sites is considered a quorum (in E3PC) if and only if it contains a majority of the votes of each item accessed by this transaction. A connected quorum of the sites may invoke a transaction and access the data. When the sites running the transaction wish to commit it, they run E3PC for the transaction. The basic E3PC may be invoked by a *subset* of the sites, the members of the current view. The views maintained by the accessible copies protocol are used as *failure detectors* for E3PC; when the view changes, the recovery procedure is invoked.

In the course of the view change protocol, each site executes an *update_transaction* in order to recover the most up-to-date values of each data item. If the *update_transaction* is aborted, the view change is aborted; a successful view change implies that the "newly joined" sites have successfully performed the updates and thus have given up their right to unilaterally abort the transaction. When the recovery procedure is invoked with sites that did not take part in the basic E3PC for the current transaction, these sites are considered to be in the *wait* state with their *Last_Elected*= 1 and *Last_Attempt*= 0, as if they had voted **Yes** on the transaction, and detached.

- The basic E3PC may be invoked by a *subset* of the sites, the members of the current view.
- E3PC uses view changes as its failure detector, i.e., every time the view changes, the recovery procedure is invoked.
- When the recovery procedure is invoked with "newly joined" sites that did not take part in the basic E3PC, the "newly joined" sites are considered to be in the *wait* state with their *Last_Elected*= 1 and *Last_Attempt*= 0.

Figure 8.11: E3PC adjusted to the accessible copies protocol.

Figure 8.11 summarizes the adjustments made in E3PC to make it suitable for the accessible copies protocol. With this protocol, the database is always available to a quorum of connected sites. We know of no previous database replica control protocol with this feature.

8.7 Failure Detectors and Weak Atomic Commit

The E3PC protocol presented above uses a *perfect* failure detector: Every site has accurate information regarding which sites are connected to it. This assumption is not practical: in asynchronous systems, it is not always possible to tell failed sites from very slow ones. In practice, systems use unreliable mechanisms, e.g., timeout, in order to detect failures. Such mechanisms may make mistakes and *suspect* that a *correct (connected)* site is *faulty (disconnected).*

Can the perfect failure detection assumption be relaxed? Guerraoui [Gue95] proves that the Atomic Commit Problem, as defined in Section 8.3.1, cannot be solved without a perfect failure detector; the non-triviality requirement (AC4) is too strong. He defines the *weak atomic commit problem* by changing the non-triviality requirement of atomic commit as follows:

Non-Triviality: If all sites voted **Yes**, and no site is ever *suspected*, then the decision will be to

COMMIT.

The other requirements of atomic commit are unchanged. The weak atomic commit problem can be solved with non-perfect failure detectors.

Can the weak atomic commit problem be solved in a fully asynchronous environment that is not augmented with any failure detector? Unfortunately, the answer to this question is no. In a fully asynchronous environment, reaching Consensus[2] is impossible [FLP85], in the sense that every protocol that reaches agreement is bound to have an infinite run. In particular, using any failure detector that can be implemented in such an environment, e.g., a time-out mechanism, E3PC does not fulfill the termination (AC5) requirement. However, when the protocol does terminate, the rest of the requirements of weak atomic commit are preserved.

We have seen that in order to solve weak atomic commit, the model must be augmented with some failure detector. An *eventual perfect* failure detector (formally defined in Section 2.3) may suspect correct sites, but there is a time after which correct sites are no longer suspected. Using such a failure detector, E3PC solves the weak atomic commit problem. E3PC terminates once a quorum of sites becomes connected and no failures or suspicions occur for sufficiently long. In a practical system, this assumption is likely to be fulfilled.

However, *eventual perfect* failure detectors are not the weakest ones which may be used; [CT96] and [DFKM96] define weaker classes of failure detectors. Chandra *et al.* [CHT92] prove that the weakest possible failure detector to solve Consensus in a crash failure model is the *eventual weak* failure detector. Intuitively, an eventual weak failure detector may make mistakes and suspect correct sites, but there is a time after which there is some correct site that is not suspected by any other site that is connected to it. Guerraoui and Schiper [GS95] present a solution to the weak atomic commit problem in an environment without network partitions, using an *eventual weak* failure detector. Their protocol may be adapted to work in an environment with network partitions, using the technique presented in [DFKM96]. This technique yields a protocol that is less efficient (requiring more communication) than E3PC.

8.8 Discussion

In this chapter, we demonstrate a general technique for constructing resilient algorithms that always allow progress in a majority component, using a simple and well-known protocol (namely, 3PC). The

[2]Guerraoui [Gue95] proves that the weak atomic commit problem is reducible to Consensus.

underlying concept of E3PC is the use of two counters to convey information among a sequence of majority components. This technique is exploited in the *COReL* algorithm presented in Chapter 6 and in the dynamic voting protocol in Chapter 7.

We demonstrate how the three phase commit [Ske82] protocol can be made more resilient simply by maintaining two additional counters and by changing the decision rule. The new protocol, E3PC, *always* allows a quorum of connected sites to resolve a transaction: At any point in the execution of the protocol, if a group G of sites becomes connected and this group contains a quorum of the sites, and no subsequent failures occur for sufficiently long, then all the members of G eventually reach a decision. Furthermore, every site that can communicate with a site that already reached a decision will also, eventually, reach a decision. We have shown that 3PC does not possess this feature: if the quorum in the system is "lost" (i.e., at a certain time no quorum component exists), a quorum can later become connected and still remain blocked.

E3PC does not require more communication or time than 3PC; the improved resilience is achieved simply by maintaining two additional counters. The information needed to maintain the counters is piggybacked on messages that are sent in 3PC as well as in E3PC: the values of *Last_Elected* and *Last_Attempt* are attached to messages used to elect a new coordinator.

We discuss how E3PC can be extended to work in an environment with unreliable failure detectors. In this case, the protocol solves the *weak* atomic commitment problem.

E3PC may be used in conjunction with quorum-based replication protocols, such as [Gif79, Her86, Her87, EASC85, EAT89], in order to make the database always available to a quorum. I demonstrate how E3PC may be incorporated in *accessible copies* protocols [EASC85, EAT89]; with the new protocol, the database is always available to a quorum of connected sites.

Appendix

Appendix A

Correctness Proofs

A.1 Correctness Proof of *COReL*

We now prove the correctness of the *COReL* algorithm. In Section A.1.1 we prove that the order of messages in $\mathcal{MQ}$ of each process always preserves the causal partial order, and thus, the total order determined by the algorithm at each process preserves the causal partial order. We conclude that at each process messages become both red and green in causal order, and Property 6.3.2 holds.

In Section A.1.2 we prove that messages are totally ordered in the same order at all the processes, and hence, Property 6.3.1 holds. The proof is based on the order imposed on committed primary components. We prove that if a message m is marked as green by some process p in the context of some primary component, then in all the later primary components, all the members will agree with p on m's order. In other words, every primary component preserves the order determined by previous primary components.

Finally, in Section A.1.3 we prove the liveness of *COReL*.

Notation

We denote by p **commits** j the event that p, as a member of PM_j, commits to PM_j when trying to establish it as the new primary component. We denote by p **adopts** j the event that in Step 6 of the Recovery Procedure p sets its $Last\_Committed\_Primary$ to j according to the representative's $Last\_Committed\_Primary$.

We denote by $v_p(v)$ the event that process p receives the view v.

The Epochs Model

The processes running the protocol may be viewed as state machines; they react to messages that they receive by the TO-GCS. The *event* $e(m,p)$ is the reaction of process p when it receives the message m. The event may include internal state changes as well as transmission of messages by p.

A *history* of the protocol is a set of events, partially ordered by the *causal* partial order. In Chapter 2 we define the causal order of messages motivated by Lamport's [Lam78] definition of the order of events in a distributed system. We generalize the definition of causal order to events and views as follows:

- The **causal order of events** is defined as follows:
  $e(m,p) \xrightarrow{cause} e(m',q)$ if $m \xrightarrow{cause} m'$.
- The **causal order of views** is defined as follows: View v causally precedes view v' if $(\exists p)(\exists q)$ $v_p(v) \xrightarrow{cause} v_q(v')$.

A history is divided into *epochs*:

Definition:

- An event $e(m,p)$ *happens in* $epoch_i(p)$ if e occurs when $Last\_Committed\_Primary_p = i$ and e does not change $Last\_Committed\_Primary_p$.
- The event in which $Last\_Committed\_Primary_p$ is changed to i is the first event in $epoch_i(p)$. Note that $Last\_Committed\_Primary_p$ may change in two types of events, when p commits i, or when p adopts i.

We say that $epoch_i(p)$ *is empty* if in the history of p, $Last\_Committed\_Primary_p$ was never i.

A.1.1 Causal Order

We first prove that the order determined by the algorithm at each process preserves the causal partial order.

Claim A.1.1 *Messages are received at each process in an order preserving the causal partial order, and the order of the messages in $\mathcal{MQ}$ of each process always preserves the causal partial order.*

Proof: Messages are ordered at each process' $\mathcal{MQ}$ when they are first delivered to it. This order may be altered only in the course of a view change protocol when a representative of a primary component enforces its order over the TS order. The proof is by induction on the steps of the protocol in which a message is transmitted or reordered in $\mathcal{MQ}$.

When a regular message is first transmitted to the members of the current view we assume that the TO-GCS delivers it in TS order, (which preserves the causal order), and without missing causally preceding messages (in the context of the same view). The messages are inserted into each $\mathcal{MQ}$ in this order.

In a new view, v, regular messages are sent only after the Recovery Procedure ends, and therefore are received by each process after all the messages that were sent in views that causally precede v.

We now show that during a view change the order in which retransmitted messages are delivered at each process, and placed in $\mathcal{MQ}$ preserves causality.

We assume (by induction) that for each member p of a view v that receives the view v, the order of messages in $\mathcal{MQ}^p$ preserves the causal partial order when the view change occurs, and we show that this property still holds throughout the Recovery Procedure.

By the Retransmission Rule the retransmission order preserves the order of the messages in $\mathcal{MQ}$ of the retransmitting process. From our assumption, this order preserves the causal partial order. Therefore, from the assumption on the TO-GCS, the delivery order of retransmitted messages preserves the causal partial order.

Retransmitted messages are inserted into $\mathcal{MQ}$ according to the TS order with the exception of one special case: messages that are retransmitted by the chosen representative, r, in Step 5 of the Recovery Procedure are inserted into $\mathcal{MQ}$ at the receiving end ahead of non_priority messages, i.e., before any messages that r didn't have marked as yellow or green. This does not violate causality, since, from the inductive assumption on $\mathcal{MQ}^r$, the non_priority messages do not causally precede any priority message in $\mathcal{MQ}^r$. Thus, in all cases, retransmitted messages are inserted into $\mathcal{MQ}$ in causal order.

If m' precedes m in $\mathcal{MQ}^p$ and p reorders m to follow m' in Step 5 of the Recovery Procedure then the representative, r, either had m' and not m, or m' before m. Therefore, by the inductive assumption on $\mathcal{MQ}^r$, m' does not causally follow m. □

Corollary A.1.2 *Red messages are delivered to the application in an order that preserves the causal partial order.*

Proof: Follows from Claim A.1.1 and the fact that messages become red in the order that they are inserted into $\mathcal{MQ}$. □

Theorem A.1.1 *The total order of messages computed at each process extends the causal order.*

Proof: Follows from Claim A.1.1 and the fact that the ordered messages are a prefix of $\mathcal{MQ}$. □

A.1.2 Total Order

We now prove that messages are totally ordered in the same order at all the processes. The proof is based on the order imposed on committed primary components. We prove that if a message m is marked as green by some process p in the context of some primary component, then in all the later primary components, all the members will agree with p on m's order. In other words, every primary component preserves the order determined by previous primary components. The proof is by induction on the committed primary components.

Claim A.1.3 *If p and q committed to primary components with number i, PM_i^p and PM_i^q respectively, then PM_i^p and PM_i^q are the same.*

Proof: Assume the contrary. Since every two primary components intersect there is a process r that is a member of both PM_i^p and PM_i^q. Since both views were committed to, r attempted to establish both. W.l.o.g. r attempted to establish PM_i^p before attempting to establish PM_i^q, then when trying to establish PM_i^q, r had at least i as the number of the *Last_Attempted_Primary*, and therefore the *New_Primary* suggested for the new view is greater than i, which contradicts the assumption. □

Henceforth we will refer to *primary component number* i as PM_i.

Claim A.1.4 *If p adopts j then there exists a process q s.t. q commits j.*

Proof: A process sets its *Last_Committed_Primary* to j either when committing to j, or when adopting from another process that has its *Last_Committed_Primary* set to j. Therefore, there must be one process that commits to j, in order to *start* the chain. □

Claim A.1.5 *For each process p, the value of $Last\_Committed\_Primary_p$ does not decrease.*

Proof: In the protocol, a process may change its *Last_Committed_Primary* in two cases:

- In Step 6 of the Recovery Procedure, when adopting the value of the representative's *Last_Committed_Prima*
  In this case the *Last_Committed_Primary* of p does not decrease, otherwise p would have been chosen as the representative.

- When committing to a new primary component PM_j. Assume that immediately before committing to PM_j $Last\_Committed\_Primary_p = i$. We consider two cases:
  - If p committed to i, then before committing to PM_i, p attempted i, therefore, $Last\_Attempted\_Primary_p$ i when $v_p(PM_j)$ occurs.
  - Otherwise, p adopted i, and from Claim A.1.4, some member q of PM_i has committed to PM_i. Since q committed to PM_i all the members of PM_i have attempted to to establish it, and, furthermore, all the members of PM_i set their $Last\_Attempted\_Primary$ to i causally before any process committed to i. Since every two primary components intersect, there exists a process r that is a member of both PM_i and PM_j; r's attempt to establish PM_i causally precedes the event that p sets its $Last\_Committed\_Primary$ to i. Therefore, when r starts to run the Recovery Procedure in which p commits to PM_j, $Last\_Attempted\_Primary_r \geq i$, and this is the value that r sends in the state message.

  In both cases, the number j, of the new primary that the members try to establish $> i$.

□

Corollary A.1.6 *If events e and e' happen at p in epochs i and i' respectively, and if $i < i'$ then e happens before e'. Note: the order on e and e' is well defined since they both happen at process p.*

Proof: Immediate from Claim A.1.5. □

Claim A.1.7 *If process p totally orders a message m according to Order Rule 1 in the context of PM_i then all the members of PM_i have committed to PM_i. Therefore, epoch i is not empty for these processes.*

Proof: In order to totally order m according to Order Rule 1 in the context of PM_i, p has to establish PM_i (and set to TRUE the primary component bit). Before establishing PM_i, p waits for all the members of PM_i to send a commit message, therefore all the members of PM_i have committed to PM_i. □

We now proceed to the main part of the proof. In Claims A.1.9 through A.1.13 we prove, by induction, that the following proposition holds in $epoch_j(q)$ for each process q and for all j:

Proposition A.1.8 for process q in epoch j:
Assume that a message m was marked as green by some process p_m in the context of PM_{i_m} according to Order Rule 1, and that i_m is the first primary component in which some process marked m as green according to Order Rule 1. Then:

- *If $j > i_m$ and if $epoch_j(q)$ is non-empty: q has m marked as yellow or green, and* $\text{Prefix}(\mathcal{MQ}^q, m) = \text{Prefix}(\mathcal{MQ}^{p_m}, m)$ *in $epoch_j$.*
- *If $j = i_m$, and if $epoch_j(q)$ is non-empty then starting at a certain point in $epoch_j(q)$, q has m marked as yellow or green and* $\text{Prefix}(\mathcal{MQ}^q, m) = \text{Prefix}(\mathcal{MQ}^{p_m}, m)$.

Definition:

We say that *there are no contradictions in the green zone when view v is received* if :

- For each member p of view v, $v_p(v)$ occurs.
- Let *Representatives* be the set of members of v with the highest *Last_Committed_Primary* (as denoted in Step 3 of the Recovery Procedure). For each member p of v, and for each message m that p has marked as green in $\mathcal{MQ}^p$ before $v_p(v)$ occurs:
  - All the *Representatives* have m marked as yellow or green.
  - Process p agrees with all the *Representatives* on the ordered prefix of its $\mathcal{MQ}$ that ends at m.

Claim A.1.9 *Assume that* there are no contradictions in the green zone when the view v is received. *Let m be a message that all the* Representatives *have as yellow or green and agree on* $\text{Prefix}(\mathcal{MQ}, m)$ *when they receive the view v.*

Then, at the end of Step 7 of the Recovery Procedure all the members of v that execute Step 7 have identical $\mathcal{MQ}$s, and the message order in their $\text{Prefix}(\mathcal{MQ}, m)$ *is the same as the order in* $\text{Prefix}(\mathcal{MQ}, m)$ *of the Representatives when $v_p(v)$ occurred. The message order in the $\mathcal{MQ}$s is not altered until the end of this instance of the Recovery Procedure.*

Proof: We first show that all the members agree on a prefix of their $\mathcal{MQ}$s that contains all the messages that were marked as green by at least one of the members before this instance of the Recovery Procedure.

Let m be a message that all the *Representatives* have as yellow or green and agree on *Prefix*$(\mathcal{MQ}, m)$ when they receive the view v.

In Step 3 of the Recovery Procedure all the *Representatives* mark m as yellow or green, and the order of messages in their $\mathcal{MQ}$s is not altered. At the end of this step, all the *Representatives* agree on the prefix of green and yellow messages in their $\mathcal{MQ}$s. From the assumption, this prefix is consistent with the order in the green prefixes of all the other members of v.

In Step 5 of the Recovery Procedure there are two cases to consider:

- If m is component_ordered, i.e., m is green for all the members of v, then, from the assumption, all the members have identical prefixes ending at m. In this case, m is not retransmitted, m remains green and its order isn't altered.
- Otherwise, m (or its header) is retransmitted by the representative. Since no member had green messages in an order that contradicts *Prefix*$(\mathcal{MQ}, m)$ of the *Representatives*, all the members adopt the order of the representatives on a prefix of their $\mathcal{MQ}$s that contains m.

In Step 6 all the members of v mark m as green or yellow, (according to the representative's color) At the end of this step, all the members have the same set of green and yellow messages in their $\mathcal{MQ}$s, and in the same order.

We have shown that at the end of Step 6 all the members of v that execute this step agree on a prefix of their $\mathcal{MQ}$s that contains all the messages that were marked as green by one of the members before this instance of the Recovery Procedure. Furthermore, they agree on the prefix of green and yellow messages in their $\mathcal{MQ}$s. We now show that at the end of Step 7 all the members of v that execute Step 7 have identical $\mathcal{MQ}$s. And indeed, before this step they agree on the order of all green and yellow messages. During this step, all the members retransmit all the red messages, and insert them into their $\mathcal{MQ}$s. Thus, all the processes have the same set of red messages in their $\mathcal{MQ}$s. The red messages in each $\mathcal{MQ}$ are ordered according to the timestamp order of their original transmissions. Therefore, at the end of this step, all the members of v have identical $\mathcal{MQ}$s, and the message order is consistent with the order in the *Prefix*$(\mathcal{MQ}, m)$ of the *Representatives* when $v_p(v)$ occurred.

No new messages arrive until the end of the Recovery Procedure, and the message order in each $\mathcal{MQ}$ is not altered until the end of this instance of the Recovery Procedure. □

Claim A.1.10 *Assume that for each member p of the view v, $v_p(v)$ occurs (i.e., p receives the view v), and let j be the highest value of Last_Committed_Primary$_p$ among members of v at the time*

$v_p(v)$ occurs. If Proposition A.1.8 holds for each member p of v for every epoch $i \leq j$ when $v_p(v)$ occurs, then throughout this execution of the Recovery Procedure, Proposition A.1.8 holds for all the members of v for every epoch $i \leq j$.

Proof: Let m be a message that was first marked as green according to Order Rule 1 in the context of PM_{i_m} (by some process p_m) , and assume that $j > i_m$. From the assumption, all the *Representatives* (members with *Last_Committed_Primary*= j) have m as yellow or green when they start to run the protocol, and their *Prefix*$(\mathcal{MQ}, m)$ are identical to *Prefix*$(\mathcal{MQ}^{p_m}, m)$. Therefore, in Step 3 of the Recovery Procedure, none of the members changes m to red, and all the messages in their *Prefix*$(\mathcal{MQ}, m)$ are yellow or green.

Furthermore, if any other member p of v has m marked as green, then *Prefix*$(\mathcal{MQ}^p, m)$ = *Prefix*$(\mathcal{MQ}^{p_m}, m)$

In Step 5 of the Recovery Procedure there are two cases to consider:

- If m is component_ordered, i.e., m is green for all the members of v, and all the members have identical prefixes ending at m, then m is not retransmitted, m remains green and its order isn't altered.

- Otherwise, m (or its header) is retransmitted by the representative in Step 5, and marked as green or yellow (according to the representative's color) in Step 6 of the protocol, by every member that sets its *Last_Committed_Primary* to j. No member changes it to red at this step, and no member had it marked as green in contradicting order. Thus, at the end of this step of the Recovery Procedure all the members have m as yellow or green and their *Prefix*$(\mathcal{MQ}, m)$ are identical.

We have shown that m is not changed to red by any of the members of v in the course of the Recovery Procedure. Therefore, if for a process p, *Last_Committed_Primary*$_p \geq i_m$ when p started to run the protocol, Proposition A.1.8 still holds for p at any point in the course of the protocol.

It is now left to show for the case that *Last_Committed_Primary*$_p$ is initially smaller than i_m, but is changed in the course of the protocol. In this case, p adopts j in Step 6 of the Recovery Procedure.

And indeed, if p sets its *Last_Committed_Primary* to j on stable storage in Step 6 of the protocol, then, from the discussion above p had already received m before Step 6 of the protocol, and in Step 6, p marks it as green or yellow, according to its color at the representative, and adopts the order of messages in $\mathit{Prefix}(\mathcal{MQ}, m)$ of the representative, when changing its $\mathit{Last\_Committed\_Primary}$ to j.

Every process that reaches the end of Step 6 of the Recovery Procedure, has its $\mathit{Last\_Committed\_Primary} \geq i_m$, and has m marked as yellow or green, and its $\mathit{Prefix}(\mathcal{MQ}, m)$ is identical to $\mathit{Prefix}(\mathcal{MQ}^{p_m}, m)$. This is not altered in later steps of the protocol. □

Claim A.1.11 *If at a certain point in $epoch_j(q)$, q has m marked as yellow or green and* $\text{Prefix}(\mathcal{MQ}^q, m) = \text{Prefix}(\mathcal{MQ}^{p_m}, m)$*, then Proposition A.1.8 holds for q in $epoch_j$, i.e., this property is not altered in later steps of the protocol.*

Proof: The color of messages may change from yellow to red only in the course of the Recovery Procedure, and the order of messages in $\mathcal{MQ}$ may only be altered in the course of the Recovery Procedure. Therefore, it is sufficient to show that Proposition A.1.8 is invariant under the Recovery Procedure, i.e., that if it holds for all the members when they start to run the Recovery Procedure, then it also holds for all the members throughout the execution of the Recovery Procedure.

If not all the members of v receive the view v, then not all the members of v run the Recovery Procedure, therefore the other members of v receive another view without going past Step 2 of this instance of the protocol. In this case, no messages are changed to red and the $\mathcal{MQ}$s are not altered.

Therefore, we may restrict our discussion to instances of the Recovery Procedure for views v s.t. all the members of v receive the view v. The conclusion in this case is derived from Claim A.1.10.
□

Claim A.1.12 *If Proposition A.1.8 holds for all the processes in every epoch k s.t. $k < j$, and if q commits j, then Proposition A.1.8 holds for q in epoch j.*

Proof: If q commits j then q is a member of PM_j, and all the members of PM_j have attempted to establish it, therefore all of them received the view PM_j, and all of them reached the end of Step 7 of this instance of the Recovery Procedure.

Let k be $\max_{p \in PM_j} \mathit{Last\_Committed\_Primary}_p$ when $v_p(PM_j)$ occurs. For every message m that one of the members p has green, there exists a first primary component PM_{i_m} in the context of which m was marked as green according to Order Rule 1, and $i_m \leq k$. From the assumption, all the *Representatives* (members with $\mathit{Last\_Committed\_Primary} = k$) have m marked as yellow or green, and agree with p on $\mathit{Prefix}(\mathcal{MQ}, m)$. Therefore, there are no contradictions in the green zone when PM_j is received.

Let m be a message that was first marked as green (according to Order Rule 1) in the context of PM_{i_m} by some process p'. We first show that at a certain point in $epoch_j(q)$, q has m marked as yellow or green and $Prefix(\mathcal{MQ}^q, m) = Prefix(\mathcal{MQ}^{p_m}, m)$. We consider three cases:

- If $j > i_m$, since every two primary components intersect, there exists at least one process that is a member of both PM_j and PM_{i_m} and from Claim A.1.7, this process committed to PM_{i_m}, therefore, $k \geq i_m$.

  Let *Representatives* be the following set:
  $\{p \in PM_j : Last\_Committed\_Primary_p = k$ when $v_p(PM_j)$ occurs $\}$.
  From the assumption Proposition A.1.8 holds for all the representatives in k, thus, Proposition A.1.8 holds for them when $v_p(PM_j)$ occurs. Therefore, all the representatives have m marked as yellow or green and the prefixes of their $\mathcal{MQ}$s that end at m are identical when $v_p(PM_j)$ occurs. Furthermore, none of the other members have green messages in contradicting order.

  Therefore, from Claim A.1.9, at the end of Step 7 of the Recovery Procedure all the members of PM_j have identical $\mathcal{MQ}$s, and the message order is consistent with the order in $Prefix(\mathcal{MQ}^{p'}, m)$ when $v_{p'}(PM_j)$ occurred. The $\mathcal{MQ}$s are not altered until the end of this instance of the Recovery Procedure.

- If $j = i_m$ and some member, p, of PM_j had m in its $\mathcal{MQ}$ when $v_p(PM_j)$ occurred, then m is marked as green by the members of PM_j that *establish* it.

  Every process that completes the execution of this instance of the Recovery Procedureand *establishes* PM_j marks as green all the messages in its $\mathcal{MQ}$. If some process *establishes* PM_j then all the other members committed to j, and marked all these messages as yellow. From the above discussion, their $\mathcal{MQ}$s are identical at this point.

- We have shown that for every process that runs this instance of the Recovery Procedure, Proposition A.1.8 holds in the course of the run of the protocol, and when the protocol ends.

  It is left to show that if m is marked as green according to Order Rule 1 in the context of PM_j by some process p_m, and p_m first received m in the context of PM_j, then q has m marked as yellow or green at some point in epoch j and $Prefix(\mathcal{MQ}^{p_m}, m) = Prefix(\mathcal{MQ}^q, m)$. And indeed, if p_m marked m as green then it received acknowledgments for it from all the members of PM_j, all of them received m in the context of PM_j, and therefore, marked it as yellow when receiving it. From Property 6.2.2 of the TO-GCS, all the members of PM_j received the same set of messages following the view change and before m, and in the same order. These messages are ordered in the $\mathcal{MQ}$ of each process in the order they are received.

We have shown that in all cases, at a certain point in $epoch_j(q)$, q has m marked as yellow or green and $Prefix(\mathcal{MQ}^q, m) = Prefix(\mathcal{MQ}^{p_m}, m)$, it is left to show that this property *persists*, i.e., that it is not altered in later steps of the protocol. Claim A.1.11 concludes the proof. □

Claim A.1.13 *Proposition A.1.8 holds for every process q in every epoch j.*

Proof: The proof is by induction on j:

For $j = 0$, the $epoch_0$ at each process precedes committing to any primary component, therefore for all m, $i_m > j$ and Property 6.2.1 trivially holds.

We now assume that Property 6.2.1 holds for all the process in every epoch k s.t. $k < j$, and prove that it holds for all the processes in $epoch_j$. If no process has committed to PM_j, then for all the processes $epoch_j$ is empty and Proposition A.1.8 trivially holds. Otherwise, let q be a process s.t. $epoch_j(q)$ is non-empty. There are two cases to consider:

- If q commits j - then from Claim A.1.12, Proposition A.1.8 holds for q in epoch j.
- If $epoch_j(q)$ is non-empty and q does not commit to j, then q adopts j in the course of a run of the Recovery Procedurefor some view v. In this case, $j = \max_{p \in v} Last\_Committed\_Primary_p$ at the time $v_p(v)$ occurs, and q adopts j in Step 6 of the Recovery Procedure. Since q passed Step 2 of this instance of the protocol, we conclude that for each member p of the view v, $v_p(v)$ occurred (i.e., p received the view v).

  Therefore, from Claim A.1.10, Proposition A.1.8 holds for all the members of v for every epoch $i \leq j$, throughout this execution of the Recovery Procedure. In particular, Proposition A.1.8 holds at some point in $epoch_j(q)$. Claim A.1.11 concludes the proof.

□

Theorem A.1.2 *At each process, messages become totally ordered in an order which is a prefix of some common global total order. I.e., for any two processes p and q, and at any point during the execution of the protocol, the sequence of messages delivered by p is a prefix of the sequence of messages delivered by q, or vice versa.*

Proof: From the protocol, the order of green messages in $\mathcal{MQ}$ is never altered. The proof immediately follows from Claim A.1.13. □

A.1.3 Liveness of *COReL*

In this section we prove that *COReL* fulfills the liveness guarantee stated in Section 6.3.

Claim A.1.14 *If a majority of the processes form a permanently connected component, and the failure detector is an eventual perfect one, then these processes eventually receive any message previously sent by any of them and establish a primary component, following which the view change handler is not invoked.*

Proof:

Let S be a majority set of processes which form a permanently connected component. From Property 5.5.3 (which is preserved by the TO-GCS) all the members of S eventually deliver the same view v, s.t. $v.M = S$ and do not deliver any further views. From Property 5.5.5 (which is also preserved by the TO-GCS), since the members of S do not crash, do not leave the group, and do not deliver any further views, all the messages they send in the Recovery Procedure are received by all of them. In the course of the Recovery Procedure, every process retransmits messages from its $\mathcal{MQ}$ which not all the other members of S have received. Since every message a process sends is in $\mathcal{MQ}$, all the processes receive any message previously sent by any of them. Therefore, the Recovery Procedure terminates successfully and the members successfully establish a primary component. Since no further views are delivered, the view change handler is not invoked. □

Claim A.1.15 *If a majority of the processes establish a primary component, following which the view change handler is not invoked then every message sent by any of them in this primary component is eventually totally ordered by all of them.*

Proof: From Property 5.5.5, since no further view changes occur, every message sent in the new primary component is eventually delivered at every member of this component. Every process which delivers the message acknowledges it, and all the acknowledgments are also eventually delivered. Therefore, the messages become totally ordered according to Order Rule 1. □

Theorem A.1.3 *If a majority of the processes form a permanently connected component, and the failure detector is an eventual perfect one, then these processes eventually totally order all messages sent by any of them.*

Proof: From Claim A.1.14, these processes eventually receive any message previously sent by any of them and establish a primary component, following which the view change handler is not invoked. When the new primary component is established, all the previous messages become totally ordered. From Claim A.1.15, every message sent by any of them in the new primary component is also eventually totally ordered by all of them. □

A.2 Correctness Proof of the Dynamic Voting Protocol

The basic requirement from a dynamic paradigm for maintaining a primary component is to impose a total order on all the primary components formed in the system. The total order on primary components is defined by extending the causal order on components that intersect. This requirement is formally postulated in Section 7.3. We now prove the correctness of our protocol, i.e., that the total order requirement is fulfilled.

Throughout the proof we use the following notations and definitions:

- The variables: *Session_Number*, *Max_Session*, *All_Ambiguous_Sessions*, and *Max_Primary* computed during a session S, are denoted: *Session_Number*(S), *Max_Session*(S), *All_Ambiguous_Sessions*(S) and *Max_Primary*(S), respectively.
- $Sessions\_List(S)$ is defined as: $All\_Ambiguous\_Sessions(S) \cup \{Max\_Primary(S)\}$.
- The initial primary component $(\mathcal{W}_0, 0)$ is denoted: F_0. It is considered a formed session.
- In general, we shall denote a session by S, and a formed session by $\mathcal{F}$.

Lemma A.2.1 *If two members p and q of a session S attempt to form S, then p and q increment their Session_Number during session S, and $Session\_Number_p = Session\_Number_q$ upon ending the session.*

Proof: A process p attempts to form session S in Step 2 of the protocol, only if p received $Session\_Number_r$ from all members r of S during Step 1 of the protocol. During Step 2 of session S p increments $Session\_Number_p$ to $Max\_Session(S) + 1$. Since $Max\_Session(S)$ is computed over the same set of values at all processes q that also attempt to form S, $Session\_Number_p = Session\_Number_q$ upon ending the session. □

Corollary A.2.1 *During every formed session $\mathcal{F}$ every member p of $\mathcal{F}$ increments $Session\_Number_p$ to $\mathcal{F}.N$.*

Lemma A.2.2 *If two formed sessions, $\mathcal{F}_1, \mathcal{F}_2$, intersect then $\mathcal{F}_1.N \neq \mathcal{F}_2.N$.*

Proof: Let process $p \in (\mathcal{F}_1.M \cap \mathcal{F}_2.M)$. By Corollary A.2.1, p increments $Session\_Number_p$ in both $\mathcal{F}_1, \mathcal{F}_2$, w.l.g., p participates in $\mathcal{F}_1$ first. Hence $\mathcal{F}_1.N < \mathcal{F}_2.N$. □

Lemma A.2.3 *Let A be an attempt, such that there is no formed session $\mathcal{F}$ fulfilling $F_0.N < \mathcal{F}.N < A.N$, then $F_0 = Max\_Primary(A)$.*

Proof: For every process p in the system, $Session\_Number_p$ is initialized to zero. By Lemma 7.5.1 $A.N > 0$, i.e., $A.N > F_0.N$. Since A is an attempt, there exists a formed session $\mathcal{F}$ s.t. $\mathcal{F} = Max\_Primary(A)$, and $Next\_Quorum(\mathcal{F}, A)$ is TRUE. Since we extended the definition of the $Next\_Quorum$ predicate so that $Next\_Quorum(\infty, T)$ is FALSE for every set T, then $\mathcal{F} \neq (\infty, -1)$.

Assume for the sake of contradiction that $\mathcal{F} \neq F_0$. Since $\mathcal{F}$ is in particular an attempt, $\mathcal{F}.N > F_0.N$. Moreover, there exists a process p s.t. $p \in \mathcal{F}.M \cap A.M$. By Corollary A.2.1 and Lemma 7.5.1, in Step 1 of session A $Session\_Number_p \geq \mathcal{F}.N$. Thus $Session\_Number(A) > \mathcal{F}.N > F_0.N$, in contradiction to the definition of A. Hence $\mathcal{F} = F_0$. □

Lemma A.2.4 *If $Last\_Primary_p = \mathcal{F}$ then for every member q of $\mathcal{F}$ either*

- *$\mathcal{F}$ is in $Ambiguous\_Sessions_q$, or*
- *$Last\_Primary_q = \mathcal{F}$, or*
- *$Last\_Primary_q.N > \mathcal{F}.N$.*

Proof: By Lemmas 7.5.2 and 7.5.3 all members q of $\mathcal{F}$ appended $\mathcal{F}$ to $Ambiguous\_Sessions_q$. Process q deletes $\mathcal{F}$ from $Ambiguous\_Sessions_q$ upon forming a session $\mathcal{F}'$, in one of the following ways:

- q deletes $\mathcal{F}$ in step 3 of the protocol during session $\mathcal{F}'$. If $\mathcal{F}' = \mathcal{F}$, then $Last\_Primary_q = \mathcal{F}$. Otherwise, q participated in $\mathcal{F}'$ after it ended session $\mathcal{F}$. By Corollary A.2.1, $Session\_Number_q$ is incremented during session $\mathcal{F}'$, hence $\mathcal{F}'.N > \mathcal{F}.N$.

- q deletes $\mathcal{F}$ in Step 2 of the protocol. Since $\mathcal{F}$ was formed by one of its members, q may delete it only when q adopts $\mathcal{F}'$ according to the resolution rules during a later session than $\mathcal{F}'$. Hence $q \in \mathcal{F}'.M$ and $\mathcal{F}'.N \geq \mathcal{F}.N$. If $\mathcal{F}' = \mathcal{F}$, then $Last\_Primary_q = \mathcal{F}$. Otherwise, by Lemma A.2.2, $\mathcal{F}'.N > \mathcal{F}.N$.

Note that $\mathcal{F}'$ can later be deleted from $Last\_Primary_q$, when q forms a new session, but in this case the newly formed session always has a session number greater than the previous formed session held in $Last\_Primary_q$. □

Lemma A.2.5 *For every sequence of formed sessions $\mathcal{F}_1, \ldots, \mathcal{F}_k$ and a formed session $\mathcal{F}$, such that*

- $\forall i < k \quad \mathcal{F}_i.N < \mathcal{F}_{i+1}.N$, *and*
- $\forall i < k \quad \mathit{Next\_Quorum}(\mathcal{F}_i, \mathcal{F}_{i+1})$ *is* TRUE, *and*
- $\mathcal{F}.N > \mathcal{F}_k.N$

if $\mathcal{F}_1 \in \mathit{Sessions\_List}(\mathcal{F})$ *then* $\mathit{Next\_Quorum}(\mathcal{F}_k, \mathcal{F})$ *is* TRUE.

Proof : By induction on k.

- **Base case $k = 1$.**

  Since $\mathcal{F}$ is a formed session and $\mathcal{F}_1 \in \mathit{Sessions\_List}(\mathcal{F})$ then, according to the protocol, $\mathit{Next\_Quorum}(\mathcal{F}_1, \mathcal{F})$ is TRUE.

- **General case $k > 1$.**

  By the induction hypothesis, $\mathit{Next\_Quorum}(\mathcal{F}_{k-1}, \mathcal{F})$ is TRUE. Since $\mathit{Next\_Quorum}(\mathcal{F}_{k-1}, \mathcal{F}_k)$ is also TRUE then, by the properties of *Next_Quorum*, $\mathcal{F}_k.M \cap \mathcal{F}.M \neq \emptyset$.

  Let $p \in \mathcal{F}_k.M \cap \mathcal{F}.M$. Since $\mathcal{F}.N > \mathcal{F}_k.N$ and $\mathcal{F}$ is a formed session, p participated in $\mathcal{F}_k$ before participating in $\mathcal{F}$. By Lemma A.2.4, when $\mathcal{F}$ begins, there are two possibilities:

  1. $\mathcal{F}_k$ is in $\mathit{Ambiguous\_Sessions}_p$. Hence $\mathit{Next\_Quorum}(\mathcal{F}_k, \mathcal{F})$ is TRUE.
  2. p has $\mathit{Last\_Primary}_p.N \geq \mathcal{F}_k.N$. This case is not possible since $\mathcal{F}_1 \in \mathit{Sessions\_List}(\mathcal{F})$, therefore when session $\mathcal{F}$ begins, p has $\mathit{Last\_Primary}_p.N \leq \mathcal{F}_1.N < \mathcal{F}_k.N$.

□

Lemma A.2.6 *Let $\mathcal{G}$ be a formed session other than F_0. Let $\mathcal{F}$ be a formed session such that $\mathcal{F}.N = max(F.N | F$ is a formed session and $F.N < \mathcal{G}.N)$. Then the value of $\mathcal{F}.N$ is unique among formed sessions, and $\mathit{Next\_Quorum}(\mathcal{F}, \mathcal{G})$ is* TRUE.

Proof: The proof is by induction on $\mathcal{F}.N$.

- **Base case $\mathcal{F}.N = 0$.**

  By Lemma 7.5.1, and since for every process p $\mathit{Session\_Number}_p$ is initialized to zero, $\mathcal{F} = F_0$. By Lemma A.2.3 $F_0 = \mathit{Max\_Primary}(\mathcal{G})$ therefore $\mathit{Next\_Quorum}(F_0, \mathcal{G})$ is TRUE.

- **General case $\mathcal{F}.N > 0$.**

  Let $\mathcal{F}^*$ be a formed session such that $\mathcal{F}^*.N = max(F.N|F$ is a formed session and $F.N < \mathcal{F}.N)$. Then, by the induction hypothesis, the value of $\mathcal{F}^*.N$ is unique among formed sessions, and $Next\_Quorum(\mathcal{F}^*, \mathcal{F})$ is TRUE. Assume, for the sake of contradiction, that $\mathcal{F}.N$ is not unique among formed sessions, then there exists a formed session F' such that $F'.N = \mathcal{F}.N$. By the induction hypothesis $Next\_Quorum(\mathcal{F}^*, F')$ is also TRUE. By the properties of *Next_Quorum*, $\mathcal{F}$ and F' intersect, hence by Lemma A.2.2 $F'.N \neq \mathcal{F}.N$, a contradiction. Therefore $\mathcal{F}.N$ is unique among formed sessions.

  Let $Max\_Primary(\mathcal{G}) = \mathcal{F}'$. If $\mathcal{F}' = \mathcal{F}$ then $Next\_Quorum(\mathcal{F}, \mathcal{G})$ is TRUE and we are done. Otherwise, by the definition of $\mathcal{F}$ and the uniqueness of $\mathcal{F}.N$ among formed sessions, $\mathcal{F}'.N < \mathcal{F}.N$. By applying the induction hypothesis multiple times we construct a unique sequence of formed sessions, $\mathcal{F}_1, \ldots, \mathcal{F}_k$, such that:

  - $\mathcal{F}_1 = \mathcal{F}'$, and
  - $\mathcal{F}_k = \mathcal{F}$, and
  - $\forall i < k \quad \mathcal{F}_i.N < \mathcal{F}_{i+1}.N$, and
  - $\forall i < k \quad Next\_Quorum(\mathcal{F}_i, \mathcal{F}_{i+1})$ is TRUE, and
  - $\mathcal{G}.N > \mathcal{F}_k.N$

  Furthermore, $\mathcal{F}' \in Sessions\_List(\mathcal{G})$, hence by Lemma A.2.5 $Next\_Quorum(\mathcal{F}, \mathcal{G})$ is TRUE.

□

Theorem A.2.1 *The transitive closure of the causal order between intersecting formed sessions, denoted $\prec$, is a total order.*

Proof: By Lemma A.2.6, each formed session has a unique session number, hence we can define a total order on formed sessions as follows: $\mathcal{F} < \mathcal{F}'$ if $\mathcal{F}.N < \mathcal{F}'.N$. We now show that $\mathcal{F} < \mathcal{F}'$ iff $\mathcal{F} \prec \mathcal{F}'$, implying that $\prec$ is a total order on formed sessions.

$\Leftarrow$ **:** If $\mathcal{F} \prec \mathcal{F}'$, then there is a sequence of intersecting formed sessions starting at $\mathcal{F}$ and ending at $\mathcal{F}'$. Therefore, by Corollary A.2.1 all the members of each such formed session incremented their session number to the same value. Together with Lemma 7.5.1 it implies that $\mathcal{F}.N < \mathcal{F}'.N$.

$\Rightarrow$ **:** If $\mathcal{F}.N < \mathcal{F}'.N$, then by Lemma A.2.6 and the properties of *Next_Quorum*, every formed session intersects with the formed session that precedes it w.r.t. the session number. Therefore, $\mathcal{F} \prec \mathcal{F}'$. $\Box$

A.3 Correctness of the Dynamically Changing Quorum System

We now prove that the mechanism introduced in Section 7.6 is correct, i.e., that the resulting protocol fulfills the total order requirement. We note that the only difference between this protocol and the protocol presented in Section 7.4 is in the way the *Next_Quorum* predicate is evaluated.

In proving Lemmas A.2.5 and A.2.6, we relied on the property that if $Next\_Quorum(S, T)$ and $Next\_Quorum(S, T')$ then $T \cap T' \neq \emptyset$. With the dynamically changing quorum system, the correctness of this property is subtle: it depends on the times at which these expressions are evaluated. Therefore, the proofs of these lemmas no longer hold with the new mechanism. In this section we prove Lemma A.3.8, which is equivalent to Lemma A.2.5, and Lemma A.3.9 – the equivalent of Lemma A.2.6. From the latter we deduce Theorem A.3.1. In the proof, we use other lemmas that were proven in Sections 7.5 and A.2. These lemmas remain valid since their proofs do not refer to the properties of the *Next_Quorum* predicate.

Lemma A.3.1 *At each process p, $\mathcal{W}_p$ and $\mathcal{W}_p \cup \mathcal{A}_p$ are monotonically increasing.*

Proof: According to the protocol, processes are never removed from $\mathcal{W}_p$. Processes are removed from $\mathcal{A}_p$ only after they were added to $\mathcal{W}_p$. $\Box$

Lemma A.3.2 *All the members that attempt to form a session S, set their $\mathcal{W}$ and $\mathcal{A}$ variables to the same value in the Attempt Step.*

Proof: Processes attempt to form a session after receiving information messages from all other members of the session. They compute $\mathcal{W}$ and $\mathcal{A}$ using only the information in these messages. All the members receive the same set of messages, and therefore the result of the computation is the same. $\Box$

Henceforth we denote by $\mathcal{W}(S)$ and $\mathcal{A}(S)$ the values of $\mathcal{W}$ and $\mathcal{A}$ computed in the Attempt Step during session S. We denote by $\mathcal{WA}(S)$ the union $\mathcal{W}(S) \cup \mathcal{A}(S)$.

Lemma A.3.3 *If a process p formed a session $\mathcal{F}$, then at the end of session $\mathcal{F}$ $\mathcal{W}_p \cup \mathcal{A}_p = \mathcal{WA}(\mathcal{F})$.*

Proof: Since $\mathcal{F}$ is a formed session, by Lemmas 7.5.2 and 7.5.3 all the members of $\mathcal{F}$, including p, attempted to form session $\mathcal{F}$. If p formed session $\mathcal{F}$ during $\mathcal{F}$, then the only change process p could make to $\mathcal{A}_p$ and $\mathcal{W}_p$ after attempting the session was to move members from $\mathcal{A}_p$ to $\mathcal{W}_p$. Hence, after p attempted to form session $\mathcal{F}$, the value of $\mathcal{W}_p \cup \mathcal{A}_p$ does not change until the end of session $\mathcal{F}$, and, by Lemma A.3.2, is equal to $\mathcal{WA}(S)$. □

Lemma A.3.4 *If for two formed sessions $\mathcal{F}_i$ and $\mathcal{F}_j$, $\mathcal{F}_i \in$ Sessions_List$(\mathcal{F}_j)$, then $\mathcal{F}_i$ and $\mathcal{F}_j$ intersect.*

Proof: Since $\mathcal{F}_i$ in *Sessions_List*$(\mathcal{F}_j)$, there is a member p of $\mathcal{F}_j$ s.t. during $\mathcal{F}_j$, $\mathcal{F}_i$ is in $\{Last\_Primary_p\} \cup Ambiguous\_Sessions_p$. Since a process records only sessions in which it participated, process p participated in session $\mathcal{F}_i$, and therefore $\mathcal{F}_i$ and $\mathcal{F}_j$ intersect.□

Lemma A.3.5 *If two formed sessions $\mathcal{F}_i, \mathcal{F}_j$ intersect and $\mathcal{F}_i.N < \mathcal{F}_j.N$, then $\mathcal{WA}(\mathcal{F}_i) \subseteq \mathcal{WA}(\mathcal{F}_j)$.*

Proof: There exists a process p s.t. $p \in \mathcal{F}_i.M \cap \mathcal{F}_j.M$. From Lemmas 7.5.2 and 7.5.3 process p attempted to form both sessions. Therefore, by Lemmas A.3.2 and A.3.3, at the end of session $\mathcal{F}_i$, $\mathcal{W}_p \cup \mathcal{A}_p = \mathcal{WA}(\mathcal{F}_i)$, while at the end of session $\mathcal{F}_j$, $\mathcal{W}_p \cup \mathcal{A}_p = \mathcal{WA}(\mathcal{F}_j)$. From Lemma A.3.1, $\mathcal{WA}(\mathcal{F}_i) \subseteq \mathcal{WA}(\mathcal{F}_j)$. □

Lemma A.3.6 *For every formed session $\mathcal{F}$ and for every process $q \in \mathcal{W}(\mathcal{F}) \cap \mathcal{F}.M$, q is a member of some formed session $\mathcal{F}'$ s.t. $\mathcal{F}'.N < \mathcal{F}.N$.*

Proof: Recall that $\mathcal{W}(\mathcal{F})$ is $\bigcup_{p \in \mathcal{F}.M} \mathcal{W}_p$. For every process $p \in \mathcal{F}.M$, $\mathcal{W}_p$ is initialized to be either $\mathcal{W}_0 = F_0.M$ or an empty set. If $q \in \mathcal{W}_0$, then q is a member of F_0, which is the formed session with the smallest number. Otherwise, process $q \notin \mathcal{W}_0$ could have been added to $\mathcal{W}_p$ in one of two cases:

- during the Form Step of a session $\mathcal{F}'$, if $q \in \mathcal{F}'.M$. In this case, q is a member of the formed session $\mathcal{F}'$ and p participated in $\mathcal{F}'$ before participating in $\mathcal{F}$. Therefore, from Corollary A.2.1, $\mathcal{F}'.N < \mathcal{F}.N$.

- during the Attempt Step of a session S, if there exists a member r of $S.M$ such that $q \in \mathcal{W}_r$. Let s be the first process that added q to $\mathcal{W}_s$, then s added q to $\mathcal{W}_s$ by the first case, i.e., during a formed session $\mathcal{F}'$, of which q is a member. If $q \in \mathcal{F}.M$ then since the Attempt Step of $\mathcal{F}$ causally follows the Form Step of $\mathcal{F}'$, q participated in $\mathcal{F}'$ before participating in $\mathcal{F}$. Therefore, from Corollary A.2.1, $\mathcal{F}'.N < \mathcal{F}.N$. $\Box$

Definition A.3.1 Formed Sequence *is a set of formed sessions* $\{\mathcal{F}_1, \ldots, \mathcal{F}_k\}$, *such that*

- $\mathcal{F}_1 = F_0$, *and*
- $\forall i < k \quad \mathcal{F}_i.N < \mathcal{F}_{i+1}.N$, *and*
- $\forall i < k \quad \mathcal{F}_i \in Sessions\_List(\mathcal{F}_{i+1})$, *and*
- *for every formed session* $\mathcal{F}'$, *such that* $\mathcal{F}'.N < \mathcal{F}_k.N$, $\mathcal{F}'$ *is in the formed sequence.*

Lemma A.3.7 *Let* $\mathcal{FS}$ *be a formed sequence of length* k. *For every formed session* $\mathcal{F}_i \in \mathcal{FS}$, *where* $i < k$, *the following is* TRUE*:*

- $\mathcal{WA}(\mathcal{F}_i) \subseteq \mathcal{WA}(\mathcal{F}_{i+1})$, *and*
- $\mathcal{W}(\mathcal{F}_{i+1}) \cap \mathcal{F}_{i+1}.M \subseteq \mathcal{WA}(\mathcal{F}_i)$.

Proof: By the definition of $\mathcal{FS}$, $\mathcal{F}_i \in Sessions\_List(\mathcal{F}_{i+1})$. From Lemma A.3.4 $\mathcal{F}_i$ and $\mathcal{F}_{i+1}$ intersect. Therefore, from Lemma A.3.5, $\mathcal{WA}(\mathcal{F}_i) \subseteq \mathcal{WA}(\mathcal{F}_{i+1})$. From Lemma A.3.6 $\mathcal{W}(\mathcal{F}_{i+1}) \cap \mathcal{F}_{i+1}.M \subseteq \bigcup_{\mathcal{F}_j \in \mathcal{FS}, j \leq i} \mathcal{F}_j.M$. Since for every formed session $\mathcal{F}$, $\mathcal{F}.M \subseteq \mathcal{W}(\mathcal{F}) \cup \mathcal{A}(\mathcal{F})$, then by the first part of this lemma, $\mathcal{W}(\mathcal{F}_{i+1}) \cap \mathcal{F}_{i+1}.M \subseteq \mathcal{WA}(\mathcal{F}_i)$. $\Box$

Lemma A.3.8 *Let* $\mathcal{F}$ *be a formed session. Let* $\mathcal{FS}$ *be a formed sequence s.t.:*

- $\mathcal{FS}$ *contains every formed session* $\mathcal{F}'$, *where* $\mathcal{F}'.N < \mathcal{F}.N$, *and*
- $\mathcal{FS}$ *is of length* k, *and*
- $\mathcal{F}_k.N \leq \mathcal{F}.N$.
- $\mathcal{F} \notin \mathcal{FS}$.

Then $\mathcal{F}_k.N < \mathcal{F}.N$ *and* $\mathcal{F}_k \in Sessions\_List(\mathcal{F})$.

Proof : From Lemma A.3.6, $\mathcal{W}(\mathcal{F}) \cap \mathcal{F}.M \subseteq \bigcup_{\mathcal{F}_j \in \mathcal{FS}} \mathcal{F}_j.M$. Let $\mathcal{F}_i$ be a formed session with the minimal index in $\mathcal{FS}$, such that $\mathcal{W}(\mathcal{F}) \cap \mathcal{F}.M \subseteq \bigcup_{\mathcal{F}_j \in \mathcal{FS}, j \leq i} \mathcal{F}_j.M$. By Lemma A.3.7, $(\forall j < i)\mathcal{WA}(\mathcal{F}_j) \subseteq \mathcal{WA}(\mathcal{F}_i)$, and for every formed session $\mathcal{F}_j$, $\mathcal{F}_j.M \subseteq \mathcal{WA}(\mathcal{F}_j)$, together we have:

$$\mathcal{W}(\mathcal{F}) \cap \mathcal{F}.M \subseteq \mathcal{WA}(\mathcal{F}_i) \tag{A.1}$$

The minimality of $\mathcal{F}_i$ implies that there exists a process $p \in \mathcal{F}.M \cap \mathcal{F}_i.M$. If $i = k$ then by Lemma A.2.2 $\mathcal{F}_k.N < \mathcal{F}.N$. Therefore, in any case $\mathcal{F}_i.N < \mathcal{F}.N$. By Lemma A.2.4, during $\mathcal{F}$ there exists a formed session F^* in $Ambiguous\_Sessions_p \cup \{Last\_Primary_p\}$ s.t. $F^*.N \geq \mathcal{F}_i.N$. Since $Session\_Number_p$ is monotonically increasing, $F^*.N < \mathcal{F}.N$, and hence $F^* \in \mathcal{FS}$. Either $F^* \in Sessions\_List(\mathcal{F})$, or else F^* is not in $Sessions\_List(\mathcal{F})$ because $Sessions\_List(\mathcal{F})$ contains another formed session F' such that $F'.N > F^*.N$. Note that F' is also in $\mathcal{FS}$ and $F'.N < \mathcal{F}.N$. In both cases, there exists a formed session, $\mathcal{F}_j$, in $Sessions\_List(\mathcal{F}) \cap \mathcal{FS}$ s.t. $j \geq i$.

If $j = k$ then we are done. Otherwise, we show below that $\mathcal{F}_{j+1}.N < \mathcal{F}.N$ and $\mathcal{F}_{j+1} \in Sessions\_List(\mathcal{F})$. The argument below may be repeated by induction, hence we conclude that $\mathcal{F}_k.N < \mathcal{F}.N$ and $\mathcal{F}_k \in Sessions\_List(\mathcal{F})$.

Since $\mathcal{F}_j \in Sessions\_List(\mathcal{F})$, $Next\_Quorum(\mathcal{F}_j, \mathcal{F})$ is TRUE during $\mathcal{F}$. Similarly, during $\mathcal{F}_{j+1}$, $Next\_Quorum(\mathcal{F}_j, \mathcal{F}_{j+1})$ is TRUE. There are three cases to consider:

1. If both $\mathcal{F}_{j+1}$ and $\mathcal{F}$ contain a majority of $\mathcal{F}_j$ then $\mathcal{F}_{j+1}$ and $\mathcal{F}$ intersect.

2. $\mathcal{F}$ does not contain a majority of $\mathcal{F}_j$. In this case,

$$\begin{aligned} |\mathcal{F}.M \cap \mathcal{WA}(\mathcal{F})| &> |\mathcal{WA}(\mathcal{F})| - Min\_Quorum \qquad (A.2) \\ |\mathcal{F}_{j+1}.M \cap \mathcal{W}(\mathcal{F}_{j+1})| &\geq Min\_Quorum \qquad (A.3) \end{aligned}$$

   By Lemma A.3.7, $\mathcal{W}(\mathcal{F}_{j+1}) \cap \mathcal{F}_{j+1}.M \subseteq \mathcal{WA}(\mathcal{F}_j)$ and by Lemma A.3.5, $\mathcal{WA}(\mathcal{F}_j) \subseteq \mathcal{WA}(\mathcal{F})$. Therefore, we can replace $\mathcal{W}(\mathcal{F}_{j+1})$ in Equation (3) above with $\mathcal{WA}(\mathcal{F})$. By summing the resulting equation with Equation (2), we get: $|\mathcal{F}.M \cap \mathcal{WA}(\mathcal{F})| + |\mathcal{F}_{j+1}.M \cap \mathcal{WA}(\mathcal{F})| > |\mathcal{WA}(\mathcal{F})|$. Hence, $\mathcal{F}_{j+1}$ and $\mathcal{F}$ intersect.

3. $\mathcal{F}_{j+1}$ does not contain a majority of $\mathcal{F}_j$. In this case,

$$\begin{aligned} |\mathcal{F}_{j+1}.M \cap \mathcal{WA}(\mathcal{F}_{j+1})| &> |\mathcal{WA}(\mathcal{F}_{j+1})| - Min\_Quorum \qquad (A.4) \\ |\mathcal{F}.M \cap \mathcal{W}(\mathcal{F})| &\geq Min\_Quorum \qquad (A.5) \end{aligned}$$

By Lemma A.3.7, $\mathcal{WA}(\mathcal{F}_i) \subseteq \mathcal{WA}(\mathcal{F}_{j+1})$. Together with Equation (1) above, we get: $\mathcal{W}(\mathcal{F}) \cap \mathcal{F}.M \subseteq \mathcal{WA}(\mathcal{F}_{j+1})$. As above, we can replace $\mathcal{W}(\mathcal{F})$ in Equation (5) with $\mathcal{WA}(\mathcal{F}_{j+1})$, and by summing the resulting equation with Equation (4), we conclude that $\mathcal{F}_{j+1}$ and $\mathcal{F}$ intersect.

In all cases, $\mathcal{F}_{j+1}$ and $\mathcal{F}$ intersect, and by Lemma A.2.2 $\mathcal{F}_{j+1}.N < \mathcal{F}.N$. Let process q be in $\mathcal{F}_{j+1}.M \cap \mathcal{F}.M$. By Lemma A.2.4, during $\mathcal{F}$ one of the following cases exists:

- $\mathcal{F}_{j+1} \in \mathit{Ambiguous\_Sessions}_q \cup \{\mathit{Last\_Primary}_q\}$, and therefore $\mathcal{F}_{j+1} \in \mathit{Sessions\_List}(\mathcal{F})$.
- $\mathit{Last\_Primary}_q = F'$ where $F'.N > \mathcal{F}_{j+1}.N$. This is impossible since $\mathcal{F}_j \in \mathit{Sessions\_List}(\mathcal{F})$, and therefore when $\mathcal{F}$ begins $\mathit{Last\_Primary}_q.N \leq \mathcal{F}_j.N < F'.N$, a contradiction.

□

Lemma A.3.9 *Let $\mathcal{G}$ be a formed session other than F_0. Let $\mathcal{F}$ be a formed session such that $\mathcal{F}.N = max(F.N | F$ is a formed session and $F.N < \mathcal{G}.N)$. Then the value of $\mathcal{F}.N$ is unique among formed sessions, and $\mathcal{F} \in \mathit{Sessions\_List}(\mathcal{G})$.*

Proof: The proof is by induction on $\mathcal{F}.N$.

- **Base case $\mathcal{F}.N = 0$.**

  Immediate from Lemma A.2.3.

- **General case $\mathcal{F}.N > 0$.**

  Let $\mathcal{F}^*$ be a formed session such that $\mathcal{F}^*.N = max(F.N | F$ is a formed session and $F.N < \mathcal{F}.N)$. Then, by the induction hypothesis, the value of $\mathcal{F}^*.N$ is unique among formed sessions, and $\mathcal{F}^* \in \mathit{Sessions\_List}(\mathcal{F})$. Moreover, by applying the induction hypothesis multiple times we construct a formed sequence, $\mathcal{FS}$, of length k such that $\mathcal{F}_k = \mathcal{F}$ and $\mathcal{F}_{k-1} = \mathcal{F}^*$.

  Assume, for the sake of contradiction, that $\mathcal{F}.N$ is not unique among formed sessions. Therefore, there exists another formed session $\mathcal{F}'$ such that $\mathcal{F}'.N = \mathcal{F}.N$. Note that $\mathcal{FS}$ constructed above contains every formed session with a smaller number than $\mathcal{F}'.N$. Therefore, from Lemma A.3.8, $\mathcal{F}.N < \mathcal{F}'.N$, a contradiction.

  The existence of $\mathcal{FS}$ implies, by Lemma A.3.8, that $\mathcal{F} \in \mathit{Sessions\_List}(\mathcal{G})$.

□

Theorem A.3.1 *The transitive closure of $\prec$ on formed sessions is a total order.*

Proof: By Lemma A.3.9, each formed session has a unique session number, hence we can define a total order on formed sessions as follows: $\mathcal{F} < \mathcal{F}'$ if $\mathcal{F}.N < \mathcal{F}'.N$. We now show that $\mathcal{F} < \mathcal{F}'$ iff $\mathcal{F} \prec \mathcal{F}'$, implying that $\prec$ is a total order on formed sessions.

$\Leftarrow$ **:** If $\mathcal{F} \prec \mathcal{F}'$, then as in the proof of Theorem A.2.1, it can be shown that $\mathcal{F}.N < \mathcal{F}'.N$.

$\Rightarrow$ **:** If $\mathcal{F}.N < \mathcal{F}'.N$, then by Lemma A.3.9 it is possible to build a formed sequence that contains both $\mathcal{F}$ and $\mathcal{F}'$. For every two succeeding sessions $\mathcal{F}_i, \mathcal{F}_{i+1} \in \mathcal{FS}$, $\mathcal{F}_i \in \mathit{Sessions\_List}(\mathcal{F}_{i+1})$. Therefore, by Lemma A.3.4, $\mathcal{F}_i$ and $\mathcal{F}_{i+1}$ intersect. Consequently, $\mathcal{F} \prec \mathcal{F}'$. $\square$

A.4 Correctness Proof of E3PC

This section proves the correctness of E3PC; it is shown that E3PC and its recovery procedure fulfill the requirements of atomic commitment (as defined in Chapter 7 of [BHG87]) described in Section 8.3.1. The proof follows:

AC1: **Uniform Agreement:** Theorem A.4.1 below proves that all the sites that reach a decision reach the same one.

AC2: In our protocol, a site cannot reverse its decision after it has reached one. When a site in a final state (COMMIT or ABORT) participates in some invocation of the recovery procedure, the decision in this invocation of the recovery procedure will correspond with its state.

AC3: **Validity:** The COMMIT decision can be reached only if all sites voted **Yes**: In the basic E3PC, a committable decision can be made only if all the sites vote **Yes**. If the recovery procedure is invoked with no site in a committable state, then according to the decision rule, a committable decision cannot be reached.

AC4: **Non-triviality:** If there are no suspicions during the execution of basic E3PC, then the basic E3PC succeeds in reaching a decision. If all sites voted **Yes**, then the decision is COMMIT. Since a *perfect* failure detector is assumed, if there are no failures, there are no suspicions.

Without a *perfect* failure detector, the weak non-triviality requirement (defined in [Gue95] and Section 8.7) is fulfilled.

AC5: **Termination:** At any point in the execution of the protocol, if all existing failures are repaired and no new failures occur for sufficiently long, then all sites will eventually reach a decision. Our protocol guarantees a much stronger property:

At any point in the execution of the protocol, if a group G of sites becomes connected and this group contains a quorum of the sites, and no subsequent failures occur for sufficiently long, then all the members of G eventually reach a decision. Furthermore, every site that can communicate with a site that already reached a decision will also, eventually, reach a decision.

This property is immediate from the decision rule and from our assumption that the failure detector is *perfect*. This property is also fulfilled with an *eventual perfect* failure detector, since with such a failure detector, there is a time after which correct sites are no longer suspected.

We now prove that the decision made is *unanimous*, i.e., that if one site decides to COMMIT, then no site can decide to ABORT and vice versa.

Lemma A.4.1 *If a coordinator r sets its local value of* Last_Attempt *to i and sends a* PRE-COMMIT (PRE-ABORT) *message to the participants in Step 3 of the recovery procedure, then a quorum of sites have set their value of* Last_Elected *to i during the same invocation of the recovery procedure.*

Proof. It is immediate from the protocol and from the fact that sites cannot "join" the recovery procedure in the middle, but rather the protocol must be reinvoked to let them take part. □

Lemma A.4.2 *At each site, the value of* Last_Elected *never decreases.*

Proof. The value of *Last_Elected* is modified only in Step 2 of the recovery procedure, when it is changed to *Max_Elected*+1. A site may execute Step 2 only if it took part in the election of the coordinator in that invocation of the recovery procedure and its value of *Last_Elected* was used to compute *Max_Elected*, and therefore *Max_Elected*≥*Last_Elected*, and *Last_Elected* increases. □

Lemma A.4.3 *If two sites, p and q, both set their* Last_Attempt *to the number i without changing to a final state, then either both of them set their* Last_Attempt *to i as a response to a* PRE-COMMIT *decision or both of them set their* Last_Attempt *to i as a response to a* PRE-ABORT *decision.*

Proof. A coordinator changes the value of *Last_Attempt* when it reaches a decision (in Step 3 of the recovery procedure or in the basic E3PC), and it remains in a non-final state if the decision is PRE-COMMIT or PRE-ABORT. Other sites change the value of *Last_Attempt* only in response to a PRE-COMMIT or a PRE-ABORT decision, in Step 4 of the recovery procedure, or in response to PRE-COMMIT in the basic E3PC.

Assume the contrary; then w.l.o.g., p set its *Last_Attempt* to i in response to a PRE-COMMIT decision in the course of some invocation, I_0, of the recovery procedure (or of the basic E3PC), and

q, in response to a PRE-ABORT decision, in an invocation I_1. From Lemma A.4.1, a quorum of sites set their *Last_Elected* to i in invocation I_0 and another quorum of sites set their *Last_Elected* to i in invocation I_1. Since the coordinator in invocation I_0 decided to PRE-COMMIT and the coordinator in I_1 decided to PRE-ABORT, I_0 and I_1 were *different* invocations of recovery procedure or of the basic E3PC.

Since every two quorums intersect, there exists a site, s, that set its *Last_Elected* to i in both invocations. W.l.o.g., s set its *Last_Elected* to i in I_0 before setting it to i in I_1. From the protocol, a site cannot concurrently take part in two invocations of the recovery procedure, furthermore, if a site responds to messages from the coordinator in some invocation, it necessarily took part in the election of that coordinator. Therefore, s took part in the election of the coordinator in I_1, *after* it set its *Last_Elected* to i, and from Lemma A.4.2, in the course of the election, the coordinator heard from s that its value of *Last_Elected*$\geq i$ and determined that *Max_Elected*$\geq i$. The new value of *Last_Elected* for this invocation was *Max_Elected*$+1$, which is greater than i, which contradicts our assumption. □

Lemma A.4.4 *At each site, at any given time,* Last_Elected$\geq$Last_Attempt.

Proof. From Lemma A.4.2, the value of *Last_Elected* never decreases, so it is sufficient to show that *Last_Attempt* is never increased to exceed it. This is proven by induction on the steps of the protocol in which *Last_Attempt* changes. *Base:* When E3PC is initiated, *Last_Elected* is set to one, and *Last_Attempt*, to zero. *Step:* Whenever *Last_Attempt* is changed in the course of the protocol, it takes the value of *Last_Elected.* □

Lemma A.4.5 *The value of* Last_Attempt *at each site increases every time the site changes state from a committable state to a non-final, non-committable state and vice versa. The value of* Last_Attempt *never decreases.*

Proof. The only non-final committable state is PRE-COMMIT, and the only way to switch to a PRE-COMMIT state is in response to a PRE-COMMIT decision, when setting *Last_Attempt* to *Last_Elected.* Likewise, the only way to switch from a committable state to a non-final non-committable state is in Step 3 or in Step 4 of the recovery procedure, in response to a PRE-ABORT decision, when setting *Last_Attempt* to *Last_Elected.*

It is sufficient to prove that *Last_Attempt* increases when it is set to *Last_Elected* in Step 3 or 4 of the recovery procedure, i.e., that *Last_Attempt*$<$*Last_Elected* before Step 3. And indeed, in

Step 2, *Last_Elected* is set to *Max_Elected*+1, which is greater than the value of *Last_Elected* was when the recovery procedure was initialized. From Lemma A.4.4, *Last_Elected*$\geq$*Last_Attempt* at all times, therefore, before Step 3, *Last_Elected* is greater than *Last_Attempt*. □

Lemma A.4.6 *If the coordinator reaches a* COMMIT *decision upon receiving a quorum of ACKs for* PRE-COMMIT *when setting its* Last_Attempt *to i, then for every $j \geq i$ no coordinator will decide* PRE-ABORT *when setting its* Last_Attempt *to j.*

Proof. The proof is by induction on j. *Base ($j = i$):* This is immediate from Lemma A.4.3. *Step:* Now, assume that no coordinator decides PRE-ABORT with *Last_Attempt*$= k$ for every $j > k \geq i$, and prove for j. From the assumption, no site can be in a non-final non-committable state with its $j >$*Last_Attempt*$\geq i$. Now, assume some coordinator r sets its *Last_Attempt* to j in Step 3 of the recovery procedure, it is left to show that r did not decide PRE-ABORT during this invocation of the recovery procedure. Assume the contrary, then r collected states, from a quorum of sites with *Last_Attempt*$< j$, and therefore, in this invocation *Max_Attempt*$< j$. Since every two quorums intersect, at least one member of G, p took part in this invocation of the recovery procedure and sent its state to r. Since $j > i$, from Lemma A.4.5, p set its *Last_Attempt* to i (and switched to a committable state) *before* this invocation. But, no site can be in a non-final non-committable state with its $j >$*Last_Attempt*$\geq i$, and therefore *Is_Max_Attempt_Committable* is TRUE in this invocation, which contradicts the assumption that r decides PRE-ABORT. □

Lemma A.4.7 *If the coordinator reaches a* COMMIT *decision when setting its* Last_Attempt *to i, then for every $j \geq i$ no coordinator will decide* PRE-ABORT *when setting its* Last_Attempt *to j.*

Proof. There are two cases to consider:

- If the coordinator reaches a COMMIT decision upon receiving a quorum of ACKs for PRE-COMMIT when setting its *Last_Attempt* to i, then from Lemma A.4.6 for every $j \geq i$ no coordinator will decide PRE-ABORT when setting its *Last_Attempt* to j.

- If the coordinator reaches a COMMIT decision during the recovery procedure upon receiving a COMMIT state, then some coordinator has reached a COMMIT decision before, when its *Last_Attempt* was $< i$. Go back, by induction, to the first coordinator that reached a COMMIT decision. This coordinator must have reached a commit decision according to the previous case. Thus, it can be concluded that for every $j \geq i$ no coordinator will decide PRE-ABORT when setting its *Last_Attempt* to j. □

Lemma A.4.8 *If the coordinator reaches an* ABORT *decision upon receiving a quorum of ACKs for* PRE-ABORT *when setting its* Last_Attempt *to i, then for every $j \geq i$ no coordinator will decide* PRE-COMMIT *when setting its* Last_Attempt *to j.*

Proof. This lemma is dual to Lemma A.4.6 and can be proven the same way. □

Lemma A.4.9 *If the coordinator reaches an* ABORT *decision when setting its* Last_Attempt *to i, then for every $j \geq i$ no coordinator will decide* PRE-COMMIT *when setting its* Last_Attempt *to j.*

Proof. There are three cases to consider:

- If the coordinator reaches an ABORT decision during the basic E3PC, this decision is reached because some site voted **No** on the transaction. In this case, the coordinator does not PRE-COMMIT, and no site reaches a committable state in the course of the protocol. Note: If the recovery procedure is invoked with no site in a committable state, then according to the decision rule, a committable decision cannot be reached.

- If the coordinator reaches an ABORT decision during the recovery procedure upon receiving a quorum of ACKs for PRE-ABORT when setting its *Last_Attempt* to i, then from Lemma A.4.8 for every $j \geq i$ no coordinator will decide PRE-COMMIT when setting its *Last_Attempt* to j.

- If the coordinator reaches an ABORT decision during the recovery procedure upon receiving an ABORT state, then some coordinator has reached an ABORT decision before, when its *Last_Attempt* was $< i$. Go back, by induction, to the first coordinator that reached an ABORT decision, according to one of the previous two cases, and conclude that for every $j \geq i$ no coordinator will decide PRE-COMMIT when setting its *Last_Attempt* to j. □

Theorem A.4.1 *If some site running the protocol* COMMITS *the transaction, then no other site* ABORTS *the transaction and vice versa.*

Proof. A site may COMMIT (ABORT) only upon hearing a COMMIT (ABORT) decision from its coordinator. Assume that a COMMIT or ABORT decision was reached for some transaction T. Note:

It is possible for more than one coordinator to reach a decision for the same transaction. Let i be the lowest value of *Last_Attempt* that a coordinator had when reaching a COMMIT or ABORT decision. There are two cases to consider:

1. Some coordinator reached an ABORT decision when setting its *Last_Attempt* to i:

   Assume for the sake of contradiction that some coordinator also reached a COMMIT decision, and let j be the lowest value of *Last_Attempt* of a coordinator reaching a COMMIT decision. From the assumption, $j \geq i$. Furthermore, since j is the lowest value of *Last_Attempt* of a coordinator reaching a COMMIT decision, no site could have started this invocation of the recovery procedure in the COMMITTED state, and the COMMIT decision must have been preceded by a PRE-COMMIT. But from Lemma A.4.9 no coordinator can decide PRE-COMMIT when setting its *Last_Attempt* to j, a contradiction.

2. Some coordinator reached a COMMIT decision when setting its *Last_Attempt* to i:

   The proof is similar to the proof of Case 1 above, but there is one more case to consider: An ABORT decision reached in the course of the basic E3PC (not in the recovery procedure) is *not* preceded by a PRE-ABORT decision. In this case, *Last_Attempt* is set to 1, and the COMMIT decision could not have been reached with a lower value of *Last_Attempt*; therefore $i = 1$. This case reduces to Case 1 proved above. □

Bibliography

[AAD93] O. Amir, Y. Amir, and D. Dolev. A highly available application in the Transis environment. In *Proceedings of the Hardware and Software Architectures for Fault Tolerance Workshop, at Le Mont Saint-Michel, France*, June 1993. LNCS 774.

[ABCD96] Y. Amir, D. Breitgand, G. Chockler, and D. Dolev. Group communication as an infrastructure for distributed system management. In *3rd International Workshop on Services in Distributed and Networked Environment (SDNE)*, pages 84–91, June 1996.

[ABDL96] T. Anker, D. Breitgand, D. Dolev, and Z. Levy. CONGRESS: CONnection-oriented Group-address RESolution Service. Tech. Report CS96-23, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, December 1996. Available from: `http://www.cs.huji.ac.il/~transis/`.

[ABDL97] T. Anker, D. Breitgand, D. Dolev, and Z. Levy. CONGRESS: Connection-oriented group-address resolution service. In *Proceedings of SPIE on Broadband Networking Technologies*, November 2-3 1997.

[ACBMT95] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the formal specification of group membership services. TR 95-1534, dept. of Computer Science, Cornell University, August 1995.

[ACDK97] T. Anker, G. V. Chockler, D. Dolev, and I. Keidar. The Caelum toolkit for CSCW: The sky is the limit. In *The Third International Workshop on Next Generation Information Technologies and Systems(NGITS 97)*, pages 69–76, June 1997.

[ACDK98] T. Anker, G. Chockler, D. Dolev, and I. Keidar. Scalable group membership services for novel applications. In Marios Mavronicolas, Michael Merritt, and Nir Shavit, editors, *Networks in Distributed Computing (DIMACS workshop)*, volume 45 of *DIMACS*, pages 23–42. American Mathematical Society, 1998.

[ACDV97] Y. Amir, G. V. Chokler, D. Dolev, and R. Vitenberg. Efficient state transfer in partitionable environments. In *2nd European Research Seminar on Advances in Distributed Systems (ERSADS'97)*, pages 183–192. BROADCAST (ESPRIT WG 22455), Operating Systems Laboratory, Swiss Federal Institute of Technology, Lausanne, March 1997. Full version available as Technical Report CS98-12, Institute of Computer Science, The Hebrew University, Jerusalem, Israel.

[ACM96] ACM. *Communications of the ACM 39(4), special issue on Group Communications Systems*, April 1996.

[ADKM92a] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership algorithms for multicast communication groups. In *6th International Workshop on Distributed Algorithms (WDAG)*, pages 292–312. Springer Verlag, November 1992.

[ADKM92b] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *22nd IEEE Fault-Tolerant Computing Symposium (FTCS)*, July 1992.

[ADMSM94] Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. E. Moser. Robust and Efficient Replication using Group Communication. Technical Report CS94-20, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[Ami95] Y. Amir. *Replication Using Group Communication Over a Partitioned Network.* PhD thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1995.

[AMMS$^+$93] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. In *13th International Conference on Distributed Computing Systems (ICDCS)*, pages 551–560, May 1993.

[AMMS$^+$95] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4), November 1995.

[Ank97] T. Anker. CONGRESS: CONnection-oriented Group-address RESolution Service. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1997.

[AW96] Y. Amir and A. Wool. Evaluating quorum systems over the internet. In *IEEE Fault-Tolerant Computing Symposium (FTCS)*, pages 26–35, June 1996.

[BDGB94] Ö. Babaoğlu, R. Davoli, L. Giachini, and M. Baker. RELACS: A Communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems. TR UBLCS94-15, Laboratory of Computer Science, University of Bologna, 1994.

[BDM95] Ö. Babaoğlu, R. Davoli, and A. Montresor. Failure Detectors, Group Membership and View-Synchronous Communication in Partitionable Asynchronous Systems. TR UBLCS-95-18, Department of Conmputer Science, University of Bologna, November 1995.

[BDM97] Ö. Babaoğlu, R. Davoli, and A. Montresor. Partitionalbe Group Membership: Specification and Algorithms. TR UBLCS97-1, Department of Conmputer Science, University of Bologna, January 1997.

[BFH97] K. Birman, R. Friedman, and M. Hayden. The Maestro Group Manager: A Structuring Tool For Applications With Multiple Quality of Service Requirements. Technical report, Dept. of Computer Science, Cornell University, Ithaca, NY 14850, USA, February 1997.

[BFHR98] K. Birman, R. Friedman, M. Hayden, and I. Rhee. Middleware support for distributed multimedia and collaborative computing. In *Multimedia Computing and Networking (MMCN98)*, 1998. To appear.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[Bir] K. Birman. Personal Communication.

[Bir96] K. Birman. *Building Secure and Reliable Network Applications*, chapter 18. Manning, 1996.

[BJ87] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *11th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 123–138. ACM, Nov 1987.

[BMR96] R. Baldoni, A. Mostefaoui, and M. Raynal. Causal delivery of messages with real-time data in unreliable networks. *Real-Time Systems*, 10, 1996.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[BvR94] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[Car94] Georg Carle. Reliable group communication in ATM networks. In *Proceedings of the Twelve Annual Conference on European Fibre Optic Communications and Networks EFOC&N'94*, June 21–24 1994.

[CHD98] G. Chockler, N. Huleihel, and D. Dolev. An adaptive totally ordered multicast protocol that tolerates partitions. In *17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 237–246, June 1998.

[CHKD96] G. Chockler, N. Huleihel, I. Keidar, and D. Dolev. Multimedia multicast transport service for groupware. In *TINA Conference on the Convergence of Telecommunications and Distributed Computing Technologies*, September 1996. Full version available as Technical Report CS96-3, The Hebrew University, Jerusalem, Israel.

[Cho97] G. V. Chockler. An Adaptive Totally Ordered Multicast Protocol that Tolerates Partitions. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1997.

[CHT92] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 147–158, 1992.

[CHTCB96] T.D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 322–330, May 1996.

[CK85] D. Cheung and T. Kameda. Site optimal termination protocols for a distributed database under network partitioning. In *4th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 111–121, August 1985.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CR83] F. Chin and K. V. S. Ramarao. Optimal termination protocols for network partitioning. In *ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*, pages 25–35, March 1983.

[CS95] F. Cristian and F. Schmuck. Agreeing on Process Group Membership in Asynchronous Distributed Systems. Technical Report CSE95-428, Department of Conmputer Science and Engineering, University of California, San Diego, 1995.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[DB85] D. Davcev and W. Burkhard. Consistency and recovery control for replicated files. In *10th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 87–96, 1985.

[DFKM96] D. Dolev, R. Friedman, I. Keidar, and D. Malki. Failure Detectors in Omission Failure Environments. TR 96-13, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, September 1996. Also Technical Report 96-1608, Department of Computer Science, Cornell University.

[DFKM97] D. Dolev, R. Friedman, I. Keidar, and D. Malki. Failure detectors in omission failure environments. In *16th ACM Symposium on Principles of Distributed Computing (PODC)*, page 286, August 1997. Brief announcement.

[DKM93] D. Dolev, S. Kramer, and D. Malki. Early delivery totally ordered broadcast in asynchronous environments. In *23rd IEEE Fault-Tolerant Computing Symposium (FTCS)*, pages 544–553, June 1993.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[DM95] D. Dolev and D. Malki. The design of the transis system. In K. P. Birman, F. Mattern, and A. Schipper, editors, *Theory and Practice in Distributed Systems: International Workshop*, pages 83–98. Springer Verlag, 1995. LNCS 938.

[DM96] D. Dolev and D. Malkhi. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4), April 1996.

[DMS94] D. Dolev, D. Malki, and H. R. Strong. An Asynchronous Membership Protocol that Tolerates Partitions. Technical Report CS94-6, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[DMS95] D. Dolev, D. Malki, and H. R. Strong. A Framework for Partitionable Membership Service. TR 95-4, Institute of Computer Science, The Hebrew University of Jerusalem, March 1995.

[DPFLS98] R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A dynamic view-oriented group communication service. In *17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 227–236, June 1998.

[DPLL97] R. De Prisco, B. Lampson, and Lynch. Revisiting the Paxos algorithm. In Marios Mavronicolas and Philippas Tsigas, editors, *11th International Workshop on Distributed Algorithms (WDAG)*, pages 111–125, Saarbrucken, Germany, September 1997. Springer Verlag. LNCS 1320.

[EAD91] A. El Abbadi and S.N. Dani. A dynamic accessibility protocol for replicated databases. *Data and Knowledge Engineering*, 6:319–332, 1991.

[EASC85] A. El Abbadi, D. Skeen, and F. Christian. An efficient fault-tolerant algorithm for replicated data management. In *ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*, pages 215–229, March 1985.

[EAT89] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems*, 14(2):264–290, June 1989.

[EMS95] P. D. Ezhilchelvan, A. Macedo, and S. K. Shrivastava. Newtop: a fault tolerant group communication protocol. In *15th International Conference on Distributed Computing Systems (ICDCS)*, June 1995.

[FGL$^+$96] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 300–309, May 1996.

[FJM$^+$95] Sally Floyd, Van Jacobson, Steven McCanne, Ching-Gung Liu, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. In *Proceedings of the IEEE/ACM Transactions on Networking.*, November 1995. An earlier version of this paper appeared in ACM SIGCOMM 95, August 1995, pp. 342–356.

[FKM$^+$95] R. Friedman, I. Keidar, D. Malki, K. Birman, and D. Dolev. Deciding in Partitionable Networks. TR 95-16, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, November 1995. Also Cornell TR95-1554. Available via anonymous ftp at cs.huji.ac.il (132.65.16.10) in users/transis/TR95-16.ps.gz.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, April 1985.

[FLS97] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partionable group communication service. In *16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 53–62, August 1997.

[FvR95] Roy Friedman and Robbert van Renesse. Strong and Weak Virtual Synchrony in Horus. TR 95-1537, dept. of Computer Science, Cornell University, August 1995.

[GBCvR93] B. Glade, K. Birman, R. Cooper, and R. van Renesse. Lightweight process groups in the Isis system. *Distributed Systems Engineering*, 1:29–36, 1993.

[Gif79] D.K Gifford. Weighted voting for replicated data. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, December 1979.

[GM82] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31, NO.1:48–59, Jan. 1982.

[Gra78] J.N. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*, volume 60, pages 393–481. Springer Verlag, Berlin, 1978.

[GS95] R. Guerraoui and A. Schiper. The decentralized non-blocking atomic commitment protocol. In *IEEE International Symposium on Parallel and Distributed Processing (SPDP)*, October 1995.

[Gue95] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In Jean-Michel Hélary and Michel Raynal, editors, *9th International Workshop on Distributed Algorithms (WDAG)*, pages 87–100. Springer Verlag, September 1995. LNCS 972.

[Her86] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[Her87] M. Herlihy. Concurrency versus availability: Atomicity mechanisms for replicated data. *ACM Transactions on Computer Systems*, 5(3):249–274, August 1987.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *chapter in: Distributed Systems.* ACM Press, 1993.

[HvR96] M. Hayden and R. van Renesse. Optimizing Layered Communication Protocols. Technical Report TR96-1613, Dept. of Computer Science, Cornell University, Ithaca, NY 14850, USA, November 1996.

[Jaj87] S. Jajodia. Managing replicated files in partitioned distributed database systems. In *3rd IEEE International Conference on Data Engineering*, pages 412–418, 1987.

[JM89] S. Jajodia and D. Mutchler. A hybrid replica control algorithm combining static and dynamic voting. *IEEE Trans. Knowledge and Data Eng.*, 1(4), Dec. 1989.

[JM90] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Transactions on Database Systems*, 15(2):230–280, 1990.

[KD94] I. Keidar and D. Dolev. Increasing the Resilience of Atomic Commit, at No Additional Cost. Technical Report CS94-18, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[KD96] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 68–76, May 1996.

[KD98] I. Keidar and D. Dolev. Increasing the resilience of distributed and replicated database systems. *Journal of Computer and System Sciences special issue with selected papers from PODS 1995*, Dec. 1998. To Appear. Previous version in ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS), May 1995, pp. 245–254.

[Kei94] I. Keidar. A Highly Available Paradigm for Consistent Object Replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994. Also available as Technical Report CS95-5, and via anonymous ftp at cs.huji.ac.il (132.65.16.10) in users/transis/thesis/keidar-msc.ps.gz.

[KSDM] I. Keidar, J. Sussman, D. Dolev, and K. Marzullo. A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership. In preparation.

[KT96] M. F. Kaashoek and A. S. Tanenbaum. An evaluation of the amoeba group communication system. In *16th International Conference on Distributed Computing Systems (ICDCS)*, pages 436–447, May 1996.

[Lam89] L. Lamport. The part-time parliament. TR 49, Systems Research Center, DEC, Palo Alto, September 1989.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 78.

[Mal94] D. Malki. *Multicast Communication for High Avalaibility.* PhD thesis, Institute of Computer Science, The Hebrew University of Jerusalem, 1994.

[MAMSA94] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *14th International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, June 1994. Full version: technical report ECE93-22, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA.

[MFSW95] C. P. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A toolkit for building fault-tolerant, distributed applications in large scale. In *Workshop on Parallel and Distributed Platforms in Industrial Products*, October 1995.

[MHS89] Tim Mann, Andy Hisgen, and Garret Swart. An Algorithm for Data Replication. Technical Report 46, DEC Systems Research Center, June 1989.

[MJ95] S. McCanne and V. Jacobson. Vic: A flexible framework for packet video. In *Proceedings of ACM Multimedia*, pages 511–522, November 1995.

[MLO86] C. Mohan, B. Lindsay, and R. Obermark. Transaction management in the r* distributed database management system. *ACM Transactions on Database Systems*, 11(4), February 1986.

[MMSA93] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Asynchronous fault-tolerant total ordering algorithms. *SIAM Journal on Computing*, 22(4):727–750, August 1993.

[MMSA$^+$96] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4), April 1996.

[MPS91a] S. Mishra, L. L. Peterson, and R. D. Schlichting. A Membership Protocol based on Partial Order. In *Proc. of the intl. working conf. on Dependable Computing for Critical Applications*, Feb 1991.

[MPS91b] S. Mishra, L. L. Peterson, and R. L. Schlichting. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. TR 91-32, dept. of Computer Science, University of Arizona, 1991.

[MPS93] S. Mishra, L. L. Peterson, and R. L. Schlichting. Experience with modularity in consul. *Software Practice and Experience*, 23(10):1059–1076, October 1993.

[MS94] C. Malloth and A. Schiper. View synchronous communication in large scale networks. In *2nd Open Workshop of the ESPRIT project BROADCAST (Number 6360)*, July 1995 (also available as a Technical Report Nr. 94/84 at Ecole Polytechnique Fédérale de Lausanne (Switzerland), October 1994).

[MSMA91] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Membership algorithms for asynchronous distributed systems. In *International Conference on Distributed Computing Systems (ICDCS)*, May 1991.

[Now98] A. Nowersztern. MOSHE: Membership Object-oriented Service for Heterogeneous Environments. Lab project, High Availability lab, The Hebrew University of Jerusalem, Jerusalem, Israel, January 1998. Available from: `http://www.cs.huji.ac.il/labs/transis/`.

[OMG98] OMG (Object Management Group). *CORBA/IIOP 2.2 Specification*, 1998. `http://www.omg.org`.

[PL88] J.F. Paris and D.D.E. Long. Efficient dynamic voting algorithms. In *13th International Conference on Very Large Data Bases (VLDB)*, pages 268–275, 1988.

[PL91] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *ACM SIGMOD International Symposium on Management of Data*, May 1991.

[Pos81] J. Postel. Internet Protocol. RFC 0791, USC/Information Science Institute, September 1981.

[Pow91] D. Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing.* Springer Verlag, 1991.

[PSK94] Sanjoy Paul, Krishan K. Sabnani, and David M. Kristol. Multicast transport protocols for high speed networks. In *Proceedings of the International Conference on Network Protocols*, pages 4–14, 1994.

[PSLB97] Sanjoy Paul, K. Sabnani, J.C. Lin, and S. Bhattacharyya. Reliable multicast transport protocol (RMTP). *IEEE Journal on Selected Areas in Communications*, April 1997.

[PW95] D. Peleg and A. Wool. Availability of quorum systems. *Inform. Comput.*, 123(2):210–223, 1995.

[RB91] A. M. Ricciardi and K. P. Birman. Using process groups to implement failure detection in asynchronous environments. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 341–352, August 1991.

[RCHS97] I. Rhee, S. Cheung, P. Hutto, and V. Sunderam. Group communication support for distributed multimedia and cscw systems. In *International Conference on Distributed Computing Systems (ICDCS)*, 1997.

[RD97] A. Rowley and J. Dollimore. Secure group communication for groupware applications. In *Proceedings of European Research Seminar in Advanced Distributed Systems (ERSADS'97)*, pages 222–227. Laboratoire de Systemes d'Exploitation Ecole Polytechnique Federale de Lausanne, March 1997.

[RGS$^+$96] Luis Rodrigues, Katherine Guo, Antonio Sargento, Robbert van Renesse, Brad Glade, Paulo Verissimo, and Ken Birman. A dynamic light-weight group service. In *15th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 23–25, October 1996. also Cornell University Technical Report, TR96-1611, August, 1996.

[RHDB98] O. Rodeh, M. Hayden, D. Dolev, and K. Birman. Secure Ensemble. Technical report, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1998. In preparation.

[RKBvR94] M.K. Reiter, K.P., Birman, and R. van Renesse. A security architecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 4(12), November 1994.

[Rod91] Tom Rodden. A survey of CSCW systems. *Interacting with Computers*, 3(3):319–353, 1991.

[RR96] M. Rautenberg and H. Rzehak. A control system for an interactive video on demand server handling variable data rates. In *Interactive Distributed Multimedia Systems and Services (IDMS)*, pages 265–276, March 1996. LNCS 1045.

[RV92] L. Rodrigues and P. Verissimo. xAMp, a protocol suite for group communication. RT /43-92, INESC, January 1992.

[Sch90] F. B. Schneider. Implementing fault tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[Ske82] D. Skeen. A quorum-based commit protocol. In *6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 69–80, Feb. 1982.

[SM98] J. Sussman and K. Marzullo. The *bancomat* problem: An example of resource allocation in a partitionable asynchronous system. In *12th International Symposium on DIStributed Computing (DISC)*, September 1998. To appear. Full version: Tech Report 98-570 University of California, San Diego Department of Computer Science and Engineering.

[Smi94] B. C. Smith. *Implementation Techniques for Continous Media Systems and Applications.* PhD thesis, University of California at Berkeley, 1994.

[SS83] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, SE-9 NO.3, May 1983.

[Top90] C. Topolcic. *Experimental Internet Stream Protocol: Version 2 (ST-II)*, October 1990. Internet RFC 1190.

[VKCD98] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group Communication System Specifications: A Comprehensive Study. Technical report, Institute of Computer Science, The Hebrew University of Jerusalem, 1998. In preparation.

[Vog96] Werner Vogels. World wide failures. In *Proceedings of the ACM SIGOPS 1996 European Workshop*, September 1996.

[vRBM96] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4), April 1996.

[vRHB94] R. van Renesse, T. M. Hickey, and K. P. Birman. Design and Performance of Horus: A Lightweight Group Communications System. TR 94-1442, dept. of Computer Science, Cornell University, August 1994.

[VvR94] Werner Vogels and Robbert van Renesse. *Support for Complex Multi-Media Applications using the Horus system.* Ithaca, NY 14850, USA, December 1994. On-line html document: `http://www.cs.cornell.edu/Info/People/rvr/papers/rt/novsdav.html`.

[WMK95] B. Whetten, T. Montgomery, and S. Kaplan. A high perfomance totally ordered multicast protocol. In K. P. Birman, F. Mattern, and A. Schipper, editors, *Theory and Practice in Distributed Systems: International Workshop*, pages 33–57. Springer Verlag, 1995. LNCS 938.

[Yav92] R. Yavatkar. MCP: a protocol for coordination and temporal synchronization in multimedia collaborative applications. In *12th International Conference on Distributed Computing Systems (ICDCS)*, pages 606–613, 1992. IEEE press.

[YLKD97] E. Yeger Lotem, I. Keidar, and D. Dolev. Dynamic voting for consistent primary components. In *16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 63–71, August 1997.

[ZDE$^+$93] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new resource reservation protocol. In *IEEE Network*, September 1993. The RSVP Project home page: `http://www.isi.edu/div7/rsvp/rsvp.html`.