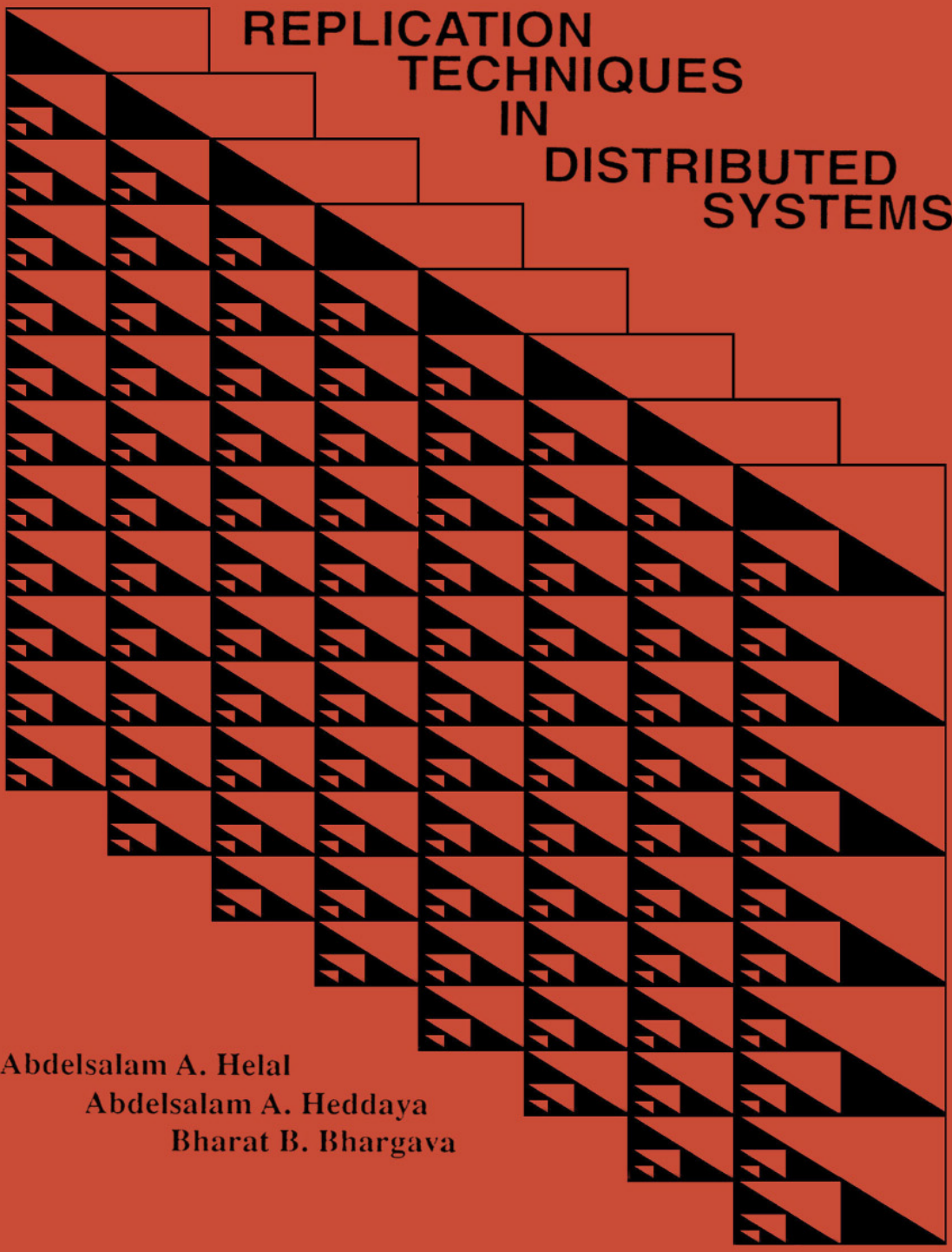
REPLICATION TECHNIQUES IN DISTRIBUTED SYSTEMS



Abdelsalam A. Helal
Abdelsalam A. Heddaya
Bharat B. Bhargava

Foreword by Jim Gray, *Microsoft, Inc.*

KLUWER ACADEMIC PUBLISHERS

Replication Techniques in Distributed Systems

The Kluwer International Series on ADVANCES IN DATABASE SYSTEMS

Series Editor
Ahmed K. Elmagarmid

Purdue University
West Lafayette, IN 47907

Other books in the Series:

DATABASE CONCURRENCY CONTROL: Methods, Performance, and Analysis
by Alexander Thomasian, IBM T. J. Watson Research Center

TIME-CONSTRAINED TRANSACTION MANAGEMENT
Real-Time Constraints in Database Transaction Systems
by Nandit R. Soparkar, Henry F. Korth, Abraham Silberschatz

SEARCHING MULTIMEDIA DATABASES BY CONTENT
by Christos Faloutsos

The Kluwer International Series on Advances in Database Systems addresses the following goals:

- To publish thorough and cohesive overviews of advanced topics in database systems.
- To publish works which are larger in scope than survey articles, and which will contain more detailed background information.
- To provide a single point coverage of advanced and timely topics.
- To provide a forum for a topic of study by many researchers that may not yet have reached a stage of maturity to warrant a comprehensive textbook.

Replication Techniques in Distributed Systems

Abdelsalam A. Helal
Purdue University
West Lafayette, Indiana, USA

Abdelsalam A. Heddaya
Boston University
Boston, Massachusetts, USA

Bharat B. Bhargava
Purdue University
West Lafayette, Indiana, USA

KLUWER ACADEMIC PUBLISHERS
NEW YORK, BOSTON, DORDRECHT, LONDON, MOSCOW

eBook ISBN: 0-306-47796-3
Print ISBN: 0-7923-9800-9

Created in the United States of America

Visit Kluwer Online at: http://kluweronline.com
and Kluwer's eBookstore at: http://ebooks.kluweronline.com

To Mindolina and the wonderful Anna Aysha.

—A. Helal

To my teachers, colleagues, and students ... for ideas.
To Mohga ... for perseverance.
To Mostafa and Shehab ... for the future.

—A. Heddaya

To my students.

—B. Bhargava

This page intentionally left blank

Contents

Foreword

Creating and maintaining multiple data copies has become a key computing system requirement. Replication is key to mobility, availability, and performance. Most of us use replication every day when we take our portable computers with us. These portables have large data stores that must be synchronized with the rest of the network when the portable is re-connected to the network. Synchronizing and reconciling these changes appears simple at first, but is actually very subtle – especially if an object has been updated in both the mobile computer and in the network. How can conflicting updates be reconciled?

Replicating data and applications is our most powerful tool to achieve high availability. We have long replicated databases on backup tapes that are restored in case the on–line data is lost. With changes in technology, data is now "immediately" replicated at other computer sites that can immediately offer service should one of the sites fail. Site replication gives very high availability. It masks environmental failures (power, storms), hardware failures, operator errors, and even some software faults.

Replication exploits locality of reference and read-intensive references to improve performance and scalability. Data is typically read much more often than it is written. Local libraries store and deliver replicas rather than have one global library for all the records, documents, books, and movies. It is a challenge to decide just what to store in a library. The decision is based on usage and storage costs. Automating these decisions is one of the key problems in replication.

There are many different ways to perform replication. Not surprisingly, there is a broad and deep literature exploring these alternatives. Until now, this literature has been scattered among many journals and conference proceedings. Helal, Heddaya, and Bhargava have collected and compiled the best of this material into a coherent taxonomy of replication techniques. The book is very readable and covers fundamental work that allows the reader to understand the roots of many ideas. It is a real contribution to the field.

The book also includes five annotated bibliographies of selected literature that focus on: (1) basic data replication, (2) process, object, and message replication, (3) issues unique to mobile and heterogeneous computing, (4) replication for availability, and (5) example systems and the techniques they use. Each bibliography begins with a brief introduction and overview written by an invited expert in that area. The survey and bibliographies cover the entire spectrum from concepts and theory to techniques. As such, the book will be a valuable reference work for anyone studying or implementing replication techniques.

One cannot look at this book and not be impressed with the progress we have made. The advances in our understanding and skill over the last twenty years are astonishing. I recall the excitement when some of these results were first discovered. Judging from current activity, there are even better ways to do what we are doing today. Data and process replication is an exciting field. I applaud Helal, Heddaya, and Bhargava for surveying and annotating the current literature for students and practitioners.

Jim Gray
Microsoft, Inc.
March, 1996

Preface

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.
—Leslie Lamport, as quoted in [188].

The dream behind interconnecting large numbers of computers has been to have their combined capabilities serve users as one. This *distributed computer* would compose its resources so as to offer functionality, dependability, and performance, that far exceed those offered by a single isolated computer. This dream has started to be realized, insofar as functionality is concerned, with such widely accepted protocols as the Domain Name Server (DNS) and the World-wide Web (WWW). These protocols compose large numbers of interconnected computers into a single system far superior in functionality than a single centralized system, yet whose distribution is *transparent* to the user.

There are no similar examples yet of protocols that are deployed on a large scale, that exploit the inherent failure independence and redundancy of distributed systems, to achieve *dependability* that is higher than that offered by isolated computers. For a service to be highly dependable, it must be both highly *available,* in the sense of the probability that a request for the service will be accepted, and highly *reliable,* as measured by the conditional probability that the service will be carried through to successful completion, given that the original invocation was admitted. Dependability becomes more critical as protocols require the cooperation of more sites, to achieve distributed functionality or distributed performance. As the number of sites involved in a computation increases, the likelihood decreases that the distributed system will deliver its functionality with acceptable availability and reliability.

In this monograph, we organize and survey the spectrum of replication protocols and systems that achieve high availability by replicating entities in failure-prone distributed computing environments. The entities we consider vary from passive untyped data objects, to which we devote about half the book, to typed

and complex objects, to processes and messages. Within the limits imposed by scope, size and available published literature, we strive to present enough detail and comparison to guide students, practitioners, and beginning researchers through the thickets of the field. The book, therefore, serves as an efficient introduction and roadmap.

Many applications are naturally distributed, in the sense that the responsibility for maintaining the currency of data, controlling access to it for security, and safeguarding its reliable storage, is decentralized and spread over a significant span of the communication network. Examples of such inherently distributed applications include international stock trading, financial transactions involving multiple banks, air traffic control, distributed hypertext, military command and control, and mobile computing. All of these applications suffer from potentially severe loss of availability, as the distributed sites become more interdependent, more numerous, or more erratically interconnected. This is because a single function provided by an application may require many sites to be operational simultaneously, an event whose probability decays quickly as a function of the number of resources involved. Centralizing the resources that underly such systems is neither acceptable nor desirable, for it introduces a potential performance bottleneck, as well as a single point of failure, the loss of which jeopardizes the entire range of functionality of the whole system. Replication enables such systems to remain distributed, while enhancing their availability and performance in the face of growing scale. Furthermore, applications that are not necessarily distributed, may be replicated and distributed in order to raise their availability and performance beyond those afforded by a remotely accessed centralized system.

However, replication of data, processes, or messages, risks a number of ill consequences that can arise even in the absence of failures. Unless carefully managed, replication jeopardizes consistency, reduces total system throughput, renders deadlocks more likely, and weakens security. Site and communication failures exacerbate these risks, especially given that—in an asynchronous environment—failures are generally indistinguishable from mere slowness. It is these undesirable effects that have slowed down the transfer of the voluminous of research on replication, into large scale applications. Our overriding concern in this book is to present how the research literature has attempted to understand and solve each of these difficulties.

Deploying replication as a design feature can be done at the application level, or at the level of common system services such as distributed databases, distributed file, hypermedia, or object-oriented systems, as well as inter-process communication systems and process managament systems. In all of these

application-independent services, which form the locus of our discussion throughout the book, data items, messages and processes can be replicated for higher availability, shorter response times, and enhanced recoverability and hence reliability. An advantage of focusing on replication for common services lies in the resulting generality of treatment, which helps the reader grasp the essential ideas without the clutter of application-specific issues. The other side of this coin is that the reader who is interested only in a particular application, will have to spend some effort in tailoring the methods we cover to suit the given application.

The book contains definitions and introductory material suitable for a beginner, theoretical foundations and algorithms, an annotated bibliography of both commercial and experimental prototype systems, as well as short guides to recommended further readings in specialized subtopics. We have attempted to keep each chapter self-contained by providing the basic definitions and terminology relevant to the replicated entity, and subsequently describing and contrasting the protocols. Appropriate uses of this book include as a recommended or required reading in graduate courses in academia (depending on level of specialization), or as a handbook for designers and implementors of systems that must deal with replication issues.

Chapter 1 defines the goals and constraints of replication, summarizes the main approaches, and discusses the baseline failure model that underpins the remainder of the book. The bulk of the book covers several dozen major methods of replicating data (Chapter 2), processes (Chapter 3), objects (Chapter 4), and messages (Chapter 5). Data replication, being the most heavily studied subject in the research literature, takes up about half the main body of the book. Special issues that arise when replication occurs in the context of heterogeneous, mobile, and large-scale systems, are treated in Chapter 6. We conclude with our outlook on the future of replication in Chapter 7. A rich set of appendices support further or deeper study of the topic by the interested reader. Appendix A briefly cites and sketches two dozen experimental and commercial systems that employ replication. A number of invited experts contribute to Appendix B, where they provide detailed reviews and comparisons of selected research articles in a number of subfields of replication. For the mathematically oriented reader, Appendix C summarizes the formal model of serializability theory and its application to replicated data via the notion of one-copy serializability. Finally, the bibliography contains nearly 200 references cited in the text.

We would appreciate feedback about the book, and we will keep an errata sheet on-line at URL:
`(http://cs-www.bu.edu/faculty/heddaya/replication-book-errata.html)`.

Acknowledgements

We are indebted to Ramkumar Krishnan for his valuable contributions to Chapters 2 and 5. Mike Bright, Konstantinos Kalpakis, Walid Muhanna, Jehan François Pâris, John Riedl, and Yelena Yesha contributed the reviews of selected further readings found in Appendix B; we thank them for their cooperation. We are also grateful to Xiangning Sean Liu for his helpful comments on Chapter 2, and to the anonymous reviewers for their useful criticism and suggestions. Substantial revisions were done while the second author was on sabbatical leave at Harvard University.

Abdelsalam Helal helal@mcc.com
Abdelsalam Heddaya heddaya@cs.bu.edu
Bharat Bhargava bb@cs.purdue.edu

June 1996

1

Introduction

A distributed computing system, being composed of a large number of computers and communication links, must almost always function with some part of it broken. Over time, only the identity and number of the failed components change. Failures arise from software bugs, human operator errors, performance overload, congestion, magnetic media failures, electronic component failures, or malicious subversion [94]. Additionally, scheduled maintenance and environmental disasters such as fires, floods and earthquakes, shut down portions of distributed systems. We can expect distributed computing services to be maintained in the presence of partialfailures at the level of *fault-isolation,* or at the higher, more difficult and more expensive level of *fault-tolerance.* At the lower level of fault-isolation, we require only that the system contain any failures so that they do not spread. In this case, the functionality of the failed part of the system is lost until the failure is repaired. If that is not acceptable, we can stipulate that the operational component of the system take over the functionality of the failed pieces, in which case the system as a whole is said to be fault-tolerant.

We can achieve fault-tolerance by *reconfiguring* the service to take advantage of new components that replace the failed ones, or by designing the service so as to *mask* failures on-the-fly. But no matter whether we adopt the reconfiguration approach or the masking approach, we need to put together redundant resources, that contain enough functionality and state information to enable full-fledged operation under partial failures. Distributed redundancy represents a necessary underpinning for the design of distributed protocols (or algorithms), to help mask the effects of component failures, and to reconfigure the system so as to stop relying on the failed component until it is repaired. Individual sites are often also required to employ enough local redundancy to ensure their

ability to recover their local states, after a failure, to that just before it. This helps simplify the operation of distributed protocols, as we will see later in the book.

From the point of view of applications, it matters not what the sources of failures are, nor the design schemes employed to combat them; what matters is the end result in terms of the availability and reliability properties of the distributed system services it needs. The widespread use of mission-critical applications in areas such as banking, manufacturing, video conferencing, air traffic control, and space exploration has demonstrated a great need for highly available and reliable computing systems. These systems typically have their resources geographically distributed, and are required to remain *available* for use with very high probability at all times. Long-lived computations and long-term data storage place the additional burden of *reliability,* which differs from availability. A system is highly available if the fraction of its down-time is very small, either because failures are rare, or because it can restart very quickly after a failure. By contrast, reliability requires that, either the system does not fail at all for a given length of time, or that it can recover enough state information after a failure for it to resume its operation as if it was not interrupted. In other words, for a system to be reliable, either its failures must be rare, or the system must be capable of fully recovering from them.

Of course, a system that is *dependable* in a complete sense, would need to be both highly available, and highly reliable. In this book we concentrate primarily on methods for achieving high availability by replication, to mask component failures, and to reconfigure the system so as to rely on a different set of resources. Many of these methods have an impact also on reliability to the extent that they can increase the likelihood of effective recovery and reconfiguration after component failures, for example by enabling a restarting (or newly recruited) site to recover its state from another site that contains a current replica of that state. To study replication techniques, we need first to understand the types and sources of failures, that these methods are designed to combat.

1.1 How systems fail

A *distributed system* is a collection of sites connected together by communication links. A *site* consists of a processing unit and a storage device. A *link* is a bidirectional communication medium between two sites. Sites communicate

with each other using *messages* which conform to standard communication protocols, but which are not guaranteed to be delivered within a maximum delay, if at all. Further, sites cooperate with each other to achieve overall system transparency and to support high-level operating system functions for the management of the distributed resources. We make no assumptions concerning the connectivity of the network, or the details of the physical interconnection between the sites.

Sites and links are prone to *failures,* which are violations of their behavioral specification. In standard terminology, a failure is enabled by the exercise of an error that leads to a faulty state. A failure does not occur, however, until the faulty state causes an externally visible action that departs from the set of acceptable behaviors. Errors arise from human mistakes in design, manufacturing or operation, or from physical damage to storage or processing devices. An error can lay dormant for a very long time before it ever causes a fault, *i.e.,* a state transition into a faulty state. When that happens, the subsystem may be able to correct its state either by luck or by fault-tolerant design, before any external ill-effects are released, in which case failure is averted. In general, software and hardware tend to be designed so as to fail by shutting down—crashing—when a fault is detected, before they produce erroneous externally visible actions, that can trigger further failures in other non-faulty subsystems.

Sites can fail by stopping, as a result of the crash of a critical subsystem, or they can fail by performing arbitrary, or possibly malicious, actions. The latter are called *Byzantine failures* [167], and we do not deal with them in this book. We generally assume sites to be *fail-stop* [186] which means that the processing and communication is terminated at the failed site before any external components are affected by the failure. A link failure is said to have occurred either if messages are no longer transmitted, or if they are dropped or excessively delayed. Partial link failures such as unidirectional transmission of messages are not considered in our discussion. Nor are we concerned with Byzantine communication failures that can arise from undetectably garbled messages, from improperly authenticated messages, or from messages that violate some high level protocol specification. An especially pernicious consequence of the site and link failures that we do consider, is the partitioning of the distributed system [63]. Sites in a *network partition* can communicate with each other but not with sites in other partitions. The difficulty in tolerating network partitions stems from the impossibility of accurately detecting relevant failures and repairs on the other side of a partition.

In light of the dangers posed by failures in thwarting the operation of a distributed system, it becomes essential to define the goals of failure management

strategies, so that they can be compared, and their benefits and costs analyzed. The next section defines availability and reliability of a distributed system, and relates them through the overarching property of dependability.

1.2 Reliability × Availability = Dependability

In this section, we attempt to give clear and distinguishing definitions of inter-related terminology that has long been used inconsistently by researchers and practitioners. We define reliability, availability, and dependability of a distributed system, whose constituent components are prone to failures. We include descriptive to somewhat formal definitions. The inter-relationships among the three definitions are pinpointed. We also display the factors that impact each definition, including protocol aspects, system policies, and mechanisms. Finally, we address the existing and needed metrics that can quantify the three definitions.

Reliability

System reliability refers to the property of tolerating constituent component failures, for the longest time. A system is perfectly reliable if it never fails. This can be due to the unlikely event that the constituent components are themselves perfectly reliable and the system's design suffers from no latent errors. Or it can arise from the more likely event that component failures and design errors, are either masked or recovered from, so that they never prevent the system as a whole from completing an active task. In a world less than perfect, a system is reliable if it fails rarely and if it almost always recovers from component failures and design faults in such a way as to resume its activity without a perceptible interruption. In short, a system is reliable to the extent that it is able successfully to complete a service request, once it accepts it. To the end user therefore, a system is perceived reliable if interruptions of on-going service is rare.

To implement reliability in a distributed system, a fault-tolerance software component is added. It includes failure detection protocols (also known as surveillance protocols) [122, 38, 107, 171, 49, 103, 19, 212], recovery protocols [23, 61, 64, 99, 125, 214], and failure adaptation and reconfiguration protocols [37, 101, 33, 30]. The additional fault-tolerance component, unless itself introspectively fault-tolerant, can render the system unreliable as a result of the added risk of failure of a critical new component.

A simple and classical measure that accounts for system reliability is the mean time to failure, or MTTF. It is a measure of failure frequency that naturally lends itself to the exponential group of failure distributions. By tradition, reliability, $R(t)$, is defined as the probability that the system functions properly in the interval $[0, t]$. In other words, $R(t)$ is the probability that the system's lifetime exceeds t, given that the system was functioning properly at time 0. Thus, $R(t)$ can be determined from the system failure probability density function, $f(x)$, and can be written as $R(t) = 1 - \int_0^t f(x)dx = 1 - F(t)$, where $F(t)$ is the cumulative probability distribution function.

With the emergence of critical business applications (like global commerce), and with the advent of mission critical systems (like the space shuttle), the MTTF-based definition of reliability needs to be extended to include the commitment to successfully completing a system operation whose lifetime may be much longer than the MTTF, once the operation is accepted by the system. In pursuit of such a commitment, a reliable system will take all needed recovery actions after a failure occurs, to detect it, restore operation and system states, and resume the processing of the temporarily interrupted tasks. We name this extended definition *recovery-enhanced reliability.* It should be noted that the extension is not a refinement of the classical definition, but rather an adaptation to a new application requirement. Unfortunately, the field has not yet produced unified metrics to quantify recovery-enhanced reliability.

Availability

System availability refers to the accessibility of system services to the users. A system is available if it is operational for an overwhelming fraction of the time. Unlike reliability, availability is instantaneous. The former focuses on the duration of time a system is expected to remain in continuous operation—or substantially so in the case of recovery-enhanced reliability—starting in a normal state of operation, and ending with failure. The latter concentrates on the fraction of time instants where the system is operational in the sense of accepting requests for new operations. To the end user therefore, a system is highly available if denial of service request is rare.

A reliable system is not necessarily highly available. For example, a reliable system that sustains frequent failures by always recovering, and always completing all operations in progress, will spend a significant amount of time performing the recovery procedure, during which the system is inaccessible. Another example, is a system that is periodically brought down to perform backup procedures. During the backup, which is done to enhance the reliability of the data storage

operation, the system is not available to initiate the storage of new data. On the other hand, a system can be highly available, but not quite reliable. For instance, a system that restarts itself quickly upon failures without performing recovery actions is more available than a system that performs recovery. Yet, the absence of recovery will render the system less reliable, especially if the failure has interrupted ongoing operations. Another available but unreliable system is one without backup down time.

For frequently submitted, short-lived operations, availability is more significant than reliability, given the very low probability of a failure interrupting the small duration of activity (since $R(0) = 1$ for every system, regardless of how reliable it is for values of $t > 0$). Such operations are therefore better served by a high probability of being admitted into the system, than by a high value of $R(t)$ for values of t much greater than the operation execution time. For long-lived, relatively infrequently requested, services and long-running transactions, reliability represents a property more critical than availability. A highly available system can be useless for a very long-lived service that is always admitted into the system, but never completes successfully because it is unable to run long enough before a failure aborts it in mid-stream.

The simplest measure of availability is the probability, $A(t)$, that the system is operational at an arbitrary point in time t. This is the probability that either:

- The system has been functioning properly in the interval $[0, t]$, or
- The last failed component has been repaired or redundantly replaced at time x, $0 < x < t$, and the repaired (redundant) component has been functioning properly since then (in the interval $[x, t]$).

This measure is known as the *instantaneous availability* [210]. It is easier to compute $A(t)$ in systems that—in response to failures—use repairs, but not redundancy, than in systems that use both. This is because repair has known distribution functions and is easy to characterize by the mean time to repair (MTTR). Redundancy, on the other hand, can be envisioned as a repair process with a non-linear distribution function, that is very difficult to characterize. At first the failure is almost instantaneously masked by switching the operation to a redundant component (which is equivalent to a very short and constant mean time to repair). But in the event that the redundant components fail one after the other, subsequent failures cannot be masked, and will require a much longer repair time.

The measure, $\lim_{t\to\infty} A(t)$ is known as the *limiting availability,* and can be shown to depend only on the mean time to fail and the mean time to repair,

$$\lim_{t\to\infty} A(t) = \frac{\text{MTTF}}{\text{MTTF+MTTR}}$$

but not on the nature of the distributions of failure times and repair times [210].

Availability can also be measured experimentally by observing the system states over a long period of time $[0, t]$, and recording the periods of time, u_i, where the system was available for operation. Availability is then stated as

$$A(t) = \frac{\sum_i u_i}{t}$$

The interval $[0, t]$ over which the system is observed is chosen equal to the utilization interval of the system, usually called the mission time.

Availability as defined above is only a broad definition. In specialized systems, additional details are used to narrow down the definition into a practical and useful one. In distributed database systems, for example, characteristics of the database (number of replicas, placement of replicas, *etc*) and the transactions (degree of concurrency, transaction length, data access distribution, operation mix, etc) must be added to the failure characteristics of the distributed system. Several specialized *database availability metrics* have been proposed in the literature [56, 35, 105, 87, 144, 150], each requiring a set of assumptions, and in some instances, defining their own notion of availability. Unfortunately, the diverse assumptions and the computational complexity of the majority of these measures limit their applicability. Some of the researchers who contributed availability enhancing protocols, like replication protocols, resorted to combinatorial techniques to describe the availability features of their protocols. For example, some used counting techniques to count all possible ways in which an operation can be performed. Intuitively, the higher the count, the higher the likelihood of finding a way to accept the operation in case of failures. Counting techniques are attractive, but unfortunately are not very useful. This is because some failures have an extent that can simultaneously render unavailable several of the many ways an operation can be performed. Definitive research in availability metrics in distributed systems is much needed, for it lies on the critical path to further progress in the field.

Dependability

Consider a service s whose duration of execution is τ. We have defined the instantaneous *availability,* $A_s(t)$, of the service to be the probability of $I_s(t)$,

which is the potential event that the system can successfully initiate s at time t. Similarly, we have defined the *reliability,* $R_s(t,\tau)$, of the service s be the conditional probability of $T_s(\tau)$ given $I_s(t)$, where $T_s(\tau)$ is the potential event that the system can terminate s successfully when its duration is τ. In our definition of *recovery-enhanced* reliability, we do not insist that s terminate at time $(t+\tau)$ exactly; that would constitute an additional *timeliness* requirement, typically imposed in real-time systems. By permitting the service to terminate at time $\geq (t+\tau)$, we allow the system to fail during its delivery of the service, so long as it can recover enough of its state to resume s and eventually terminate it successfully. The above definitions for reliability and availability agree to a large extent with Chapter 12 in Özsu and Valduriez's book [157], which contains an excellent review and classification of the sources of failures in distributed systems, and with Y.C. Tay's simple but insightful analysis of the impact of replication on reliability in [204].

Clearly, the overall success of service s depends both on its correct initiation on demand *and* on its proper termination. Therefore, we define the *dependability*[1] of s, $D_s(t,\tau)$, as the probability of the conjunction of the two events, $I_s(t)$ and $T_s(\tau)$. Bayes' law of conditional probability dictates that dependability be the product of availability and reliability. Formally, we write

$$\begin{aligned} A_s(t) &= \Pr[I_s(t)] \\ R_s(t,\tau) &= \Pr[T_s(\tau)|I_s(t)] \\ D_s(t,\tau) &= \Pr[I_s(t) \wedge T_s(\tau)] = A_s(t) \cdot R_s(t,\tau) \text{ , by Bayes' Law.} \end{aligned}$$

So we have in this definition of dependability, the happy coincidence of a single scalar metric that measures the full initiation-to-termination probability of successful service.

1.3 Replication for failure management

Now that we have defined the goals for failure management, it is time to review the available techniques for getting there, and to situate replication within the context of other approaches. We must immediately discard the obvious option of composing distributed systems from ultra-available and ultra-reliable components, whose properties are so good that they raise the corresponding properties for the whole system to the desired level. Aside from questions of

[1] Other researchers include into the notion of dependability additional dimensions such as safety, and security.

expense and feasibility, this approach collapses under the weight of increasing scale. As the size of the distributed system grows, its components' availability and reliability would also have to increase with it, violating a basic prerequisite for scalability in distributed systems: that its local characteristics be independent of its global size and structure. Therefore, the distributed system designer needs to instill mechanisms that combat the effects of failures, into the system architecture itself.

The range of known strategies for dealing with component failures so as to achieve system dependability, includes: failure detection, containment, repair (or restart), recovery, masking, and, reconfiguration after a failure. Only the last two approaches, failure masking and reconfiguration, are usually labelled as *fault-tolerance*[2] methods, since they are designed to enable the system to continue functioning while a failure is still unrepaired, or maybe even undetected. Furthermore, the other four methods—detection, containment, restart, and recovery—all directly enhance the dependability of the individual component, and only by this route do they indirectly boost system dependability. *Replication* techniques mainly address the two fault-tolerance activities of masking failures, and of reconfiguring the system in response. As such, replication lies at the heart of any fault-tolerant computer architecture, and therefore deserves to be the topic of the rest of this book.

This book covers the details of a plethora of replication protocols. In addition to masking failures, replication enables the system to be reconfigured so that it has more (or fewer) replicas, a flexibility that can be a main contributing factor to preserving dependability. Indeed, some replication protocols function primarily by reconfiguring the set of replicas after every failure, and so cannot mask undetected failures. The details of reconfiguration are also well covered in this book.

Fault-tolerance is not the only benefit of replication-based strategies, replication can also enhance failure detectability and recoverability. One of the most effective methods to detect failures is by *mirroring,* which is the simplest form of replication. When the original copy differs from the mirror copy, a failure is deemed to have occurred. The presence of replication can contribute to recoverability from certain failures, that would be otherwise unrecoverable, such as catastrophic failures that destroy the entire state of a site, including any stable storage, recovery logs, shadow values, *etc.*

[2] A fault-tolerant architecture can be viewed as one that achieves a level of system dependability, that exceeds what is attainable via straightforward aggregation of components and subsystems.

Replication in the sense of making straightforward copies of meaningful units of data, processing, or communication, represents a particularly simple instance of the more general fault-tolerance technique of introducing *redundancy,* such as error correcting codes used in communication and memory devices. Indeed, there has been a small number of attempts to bring a coding theoretic approach to bear on distributed system fault-tolerance, and we report in Section 2.8 on the most salient of these. However, we do not count error-correcting codes used in disk, main memory, and communication as falling under the rubrik of distributed replication. Nor do we include some forms of duplication, such as logging and shadowing, that are employed in enhancing individual site dependability.

1.4 Replication for performance

Replication has the capacity to improve performance by bringing the aggregate computing power of all the replica sites, to bear on a single load category. For example, replicating data items that are either read-only or read-mostly, enables as many read operations as there are replicas to be performed in *parallel.* Furthermore, each read operation can select the replica site from which to read, so as to minimize relevant costs, such as communication distance. While the constraint of read-mostly data appears to be restrictive, a large number of applications fall under this category, and hence can benefit from the performance enhancements inherent in replication of this kind of data. For example, distributed name service for long-lived objects, and file service for immutable objects, fall under this category. Furthermore, some replication protocols (such as general quorum consensus) permit a trade-off of read performance against write performance, which means that write-mostly replicated data can benefit from a similar performance enhancement. See Section B.4 for a lengthier discussion of the interplay between performance and availability considerations.

1.5 Costs and limitations of replication

Replication, be it of data, processes, or messages, provides fault-tolerance at the cost of duplicating effort by having the replicas carry out repetitive work, such as performing every data update multiple times, once at ever copy. This is by far the largest cost of replication, and it can reduce total update throughput by a factor equal to the number of copies that must be updated, as compared to

a non-replicated system that uses the same amount of resources. The storage space overhead of replication can be as large as a factor of n, the total number of copies. Some coding-theoretic replication protocols such as IDA [177] (see Section 2.8), can reduce the storage cost of replication, in terms of both size and aggregate bandwidth, at the expense of not being able to update small portions of data files efficiently. Other costs include the overhead associated with consistency control across the replicas, and, in certain situations, replication can increase the hazard of deadlock, therefore any additional system effort required to find and break deadlocks should be charged as an overhead cost of replication.

With respect to limits on availability enhancement, Raab argued in [176] that the availability of update operations—as measured by a certain metric called *site availability*—in a replicated system can be at most $\sqrt{A}$, where A is the update availability of a single, optimally placed, copy. This is a very tight bound that challenges many results that use alternative, more forgiving, metrics for availability. Unfortunately, there has not been much other work in this important area, and therefore this result remains to be interpreted by the community.

This page intentionally left blank

2

Replication of Data

Data can be replicated for performance, as in caches and in multi-version concurrency controllers, in ways that do not affect the overall availability or reliability of the data. Such systems fall beyond the scope of this book. In this chapter, we concern ourselves mostly with data replication schemes that aim primarily at improving data availability. Short discussions of performance issues are scattered throughout the book, and in Appendix B.4.

We begin by introducing the model of a distributed database in Section 2.1, which includes a discussion of concurrency control and atomicity control in non-replicated and replicated distributed databases. Data replication protocols are described starting from Section 2.2 onwards. Protocols that offer relatively limited resiliency to failures are discussed in section 2.2, followed by quorum consensus protocols based on a static quorum specification in Section 2.3. Section 2.4 covers static protocols that additionally impose certain logical structuring, such as a grid or a tree, on the distributed system. The subsequent two sections, 2.5 and 2.6, handle protocols that dynamically reconfigure in response to site failures, and in reaction to network partitions, respectively. Even though most protocols we present guarantee strong consistency, we include, in Section 2.7, methods that have relaxed or weak consistency requirements, including a few optimistic techniques. This chapter concludes with a short section, 2.8, on coding-theoretic redundancy that has had a limited but increasingly important impact on replication.

2.1 Model of Distributed Database System

A **distributed database system** (DDBS) is a collection of data items scattered over multiple, networked, failure-independent *sites* which are potentially used by many concurrent users. The database system manages this collection so as to render distribution, failures, and concurrency *transparent* to the users by ensuring that their reads (queries) and writes (updates) execute in a manner indistinguishable from reads and writes that execute on a single-user, reliable, centralized database [207]. Transparency with respect to all these factors can be expensive and, hence, often guaranteed in practice only to an extent consistent with acceptable performance. The database system we consider in this chapter is homogeneous in that it supports a single data model, schema, and query language.

Many on-line applications such as banking and airline reservation systems require that the unit of execution, from the user s perspective, include multiple individual operations on several different data items. Such databases are typically accessed by means of *transactions* [64, 81] that group these operations. Each transaction consists of a set of operation *executions* performed during a run of a program. The DDBS concerns itself with the synchronization (ordering) of operations[1] that belong to different transactions. The task of ordering those that belong to the same transaction is assigned to the application program.

Other systems and applications—most notably file systems—are satisfied with providing their guarantees on a *per operation* basis. This can be modeled as a special case in which a transaction consists of a single operation, so we do not need to discuss this case separately, except to note that it enables important optimizations in practice.

Transactions must possess the following **ACID** properties [95, 99]:

- Atomicity: a transaction is performed in its entirety or not at all,
- Consistency: a transaction must take the database from one consistent state to another,
- Isolation: a transaction is isolated from ongoing update activities. Updates of only committed transactions are visible,

[1] Henceforth, we will use *operation* to denote an operation execution.

- Durability: a transaction update applied to the database has to be permanently installed.

Of the four properties, only the second (consistency) falls on the shoulder of the application programmer. The remaining three must be ensured by the DDBS.

The property of consistency ensures that any serial execution can be considered correct. It paves the way for defining the correctness of a concurrent failure-prone execution as one that is equivalent to some failure-free serial execution. An execution is *serial* if, for every pair of transactions, all of the operations of one transaction execute before any of the operations of the other. If the initial state of the database is consistent *and* if each transaction program is designed to preserve the database consistency if executed in isolation, then an execution that is equivalent to a serial one, contains no transaction that observes an inconsistent database.

2.1.1 Concurrency Control

An execution is *serializable* if it produces the same output and has the same effect on the database as some serial execution of the same transactions. The theory of *serializability* [26, 158] formally defines the requirements to achieve serializable executions. We present a statement and proof of the core of this theory in Appendix C. *Concurrency control protocols* are used to restrict concurrent transaction executions in a centralized and distributed database system only to executions that are serializable.

Concurrency control protocols fall into one of two categories, *pessimistic* or *optimistic*. Pessimistic protocols prevent inconsistencies by disallowing *potentially* non-serializable executions, and by ensuring that the effects of a committed transaction need not be reversed or annulled. An example of a pessimistic protocol is the *two-phase locking* protocol [81] which is widely implemented in commercial systems. Optimistic protocols, on the other hand, permit non-serializable executions to occur, with anomalies detected—and the relevant transaction aborted—during a *validation* phase [132] before the effects of the transactions are made visible. An example of the optimistic approach is *certification* [132, 205] which performs the validation at commit time.

2.1.2 Atomicity Control

An atomicity control protocol ensures that each transaction is atomic, *i.e., all* the operations in a transaction execute, or none do. The distributed transaction is typically initiated at a *coordinator* site. Its read or write operations are executed by forwarding the operations to the sites containing the relevant data items, called the *participants.* An atomicity control protocol ensures that all participating database sites *agree* on whether to reveal, and to install permanently, the effects of a distributed transaction. *Two-phase commit* [96] is the most popular such protocol.

Each transaction normally issues a request to commit as its final operation. A transaction is said to be committed if and only if the coordinator and all participants agree to do so. Before deciding to commit a transaction, two-phase commit stipulates that its coordinator poll the participants to inquire if they are all *prepared* to commit. Once the coordinator receives acknowledgments from all the participants confirming their prepared state, the coordinator commits the transaction locally and informs the participants of its decision to commit. At any point in time before the coordinator records the commit of a transaction, it is free to *abort* it simply by recording the fact in its local permanent storage. In all cases, however, the coordinator must inform the participants of the fate of the transaction either actively by notifying them, or passively by responding to their queries. If the coordinator fails before all participants have learned of its decision, a *cooperative termination protocol* [135] attempts to deduce the missing information based on the states of the participants.

Nevertheless, certain failures of the coordinator or communication failures between it and the participants can cause the two-phase commit protocol, even with the aid of the cooperative termination protocol, to *block* the transaction until the failure is repaired. The *three-phase commit* protocol [194] has been devised to handle the first case, namely the failure of the coordinator. But in a partitioned network, no atomic commitment protocol can guarantee the execution of a transaction to its termination as long as a network partition exists. Failures also necessitate the use of reliable *logs* to record each action. These logs are used by *recovery protocols* [99] to restore the database to a consistent state.

2.1.3 Mutual Consistency in Replicated Databases

In a replicated database with consistency requirements similar to the one discussed in Section 2.1.1, it is important that the replication be transparent to the user. In other words, the concurrent execution of a set of transactions must be equivalent to the serial execution of the same transactions in a database that contains only one copy of each data item. This correctness criterion for replicated data is an extension of serializability and is termed *one-copy serializability* [23]. We provide a formal description of one-copy serializability in Appendix C. To achieve one-copy serializability, a *replication control* mechanism is required to ensure that operations performed on a logical data item are reflected on the physical copies. It must also ensure that the replicated system always presents the most current state of the distributed database even under site and communication failures.

2.2 Read One Write All (ROWA)

The most obvious protocols are those that keep multiple copies that must all be updated, and of which any one can be read. This section describes several such protocols in increasing sophistication, beginning with the literal *read one write all* (ROWA), and the more flexible read-one/write-all-available (ROWA-A), which tries to ensure that all sites involved in a transaction agree over the identity of the failed sites. The *primary copy* and the *true-copy token* schemes designate one copy as required for writes to succeed, but allow that designation to change dynamically. The protocols in this section can tolerate site failures, but not communication failures, unless augmented with reconfiguration as discussed in Section 2.5 below.

2.2.1 Simple ROWA Protocol

This algorithm translates a *read* operation on data item d, into one read operation on any a single copy, and a *write* operation into n writes[2], one at each copy. The underlying concurrency controller at each site synchronizes access to copies. Hence this execution is equivalent to a serial execution. In serial execution, each transaction that updates d will update all copies or none at all.

[2]We henceforth use n to denote the number of copies or sites.

So, a transaction that reads any copy of d reads the most recent value, which is the one written by the last transaction that updated all copies.

The obvious advantages of this approach is its simplicity and its ability to process reads despite site or communication failures, so long as at least one site remains up and reachable. But, in the event of even one site being down or unreachable, the protocol would have to block all write operations until the failure is repaired.

2.2.2 Read One Write All Available (ROWA-A)

In this modified ROWA approach, a transaction is no longer required to ensure updates on all copies of a data item, but only on all the *available* copies. This avoids the delay incurred by update transactions when some sites in the system fail. For this scheme to work correctly, failed sites are not allowed to become available again until they recover by copying the current value of the data item from an available copy. Otherwise, a recovering site can permit stale reads and hence, non-l-SR executions. ROWA-A can tolerate $n - 1$ site failures, but it does not tolerate network partitioning, nor communication failures in general.

The *available copies algorithm* [24, 26], which is the earliest ROWA-A method, synchronizes failures and recoveries by controlling when a copy is deemed *available* for use. An alternative idea, proposed in [36] has recovering sites use *faillocks* to detect stale copies, in which case, they initiate *copier transactions* to bring the stale copies up-to-date. In the remainder of this section, we describe the available copies algorithm, which ensures that both the transaction coordinator and an unresponsive site agree on whether the latter is down or up, before committing the transaction.

In the basic ROWA-Available algorithm, $w[d]$ operations issued by a transaction T are sent to all the sites holding copies. If a particular site s is down, there will be no response from it and T's coordinator will timeout. If s is operational then it responds indicating whether $w[d_s]$ was rejected or processed. Copy updates for which no responses are received by the coordinator are called *missing writes* [73] (the Missing Writes protocol is separately discussed in Section 2.3.5). The protocol as described so far would have been sufficient, if there were no risk that the coordinator may have made a mistake regarding the status of a participant. The following validation protocol is dedicated to discovering such a mistake before committing the transaction.

Before committing the transaction, its coordinator initiates a two-step validation protocol:

1. *Missing writes validation:* determines if all the copies that caused missing writes are still unavailable. The coordinator sends a message UNAVAIL to every site s holding a copy d_s whose write is missing. If s has come up in the meantime and has initialized d_s, it would acknowledge the request, causing T to abort since the occurrence of a concurrent $w[d_s]$ operation before T commits indicates inconsistency. If no response is received, then the coordinator proceeds to the next step.
2. *Access validation:* determines if all copies that it read from, or wrote into are still available. To achieve this, the coordinator sends a message AVAIL to every site s hosting a copy d_s that T read or wrote, s acknowledges if d_s is still available at the time it receives the message. If *all* AVAIL messages are acknowledged, then access validation has succeeded and T is allowed to commit.

The static assignment of copies to sites has a number of disadvantages. It requires that transactions attempt to update copies at down sites. Repeated update attempts at a site which has been down for a long time is a waste of resources. Also, dynamic creation and removal of copies at new sites is not possible. The validation protocol ensures correctness but at the expense of increased communication costs. The problem could be mitigated by buffering messages to the same site into a single message or by combining the communication involved in access validation with that of the *vote-req* phase of the atomic commitment protocol. An even better approach is to keep track explicitly of the set of available sites (see Section 2.5.2).

2.2.3 Primary Copy ROWA

In this method, a specific copy of a data item is designated as the *primary copy* [9, 198]; the remaining copies are called *backups*. A write operation is carried out at the primary copy and all operational backups while a read operation is executed only at the primary copy. A transaction that writes the replicated item is allowed to commit only after the primary and all operational backups have successfully recorded the write operation.

When the primary fails, a backup, chosen via a fixed line of succession or by an election protocol, takes its place as the new primary. This requires that failure

of the primary be detectable and *distinguishable* from failure to communicate with it. Otherwise, it is possible to have two primary copies if a network partition leaves the first successor of a primary in a partition other than the one in which the primary copy resides. After a backup fails, it is not allowed to participate in future elections of a new primary until it has *recovered* by querying the existing primary copy for the current value of the data item. For writes to continue to be available despite the failure of a backup, the primary has to ascertain that the backup has indeed failed and is not just slow in responding or unreachable over the network. Since distinguishing a site failure from a communication failure is difficult to achieve in practice using commonly available networks, we characterize the primary copy algorithm as unable to withstand network partitions.

Oki and Liskov [155] augment the primary copy protocol to tolerate network partitioning by combining it with a voting-based protocol that detects the existence of a majority partition. Their protocol ensures consistency by seeking agreement among all sites in this partition as to its contents. This frees the system to choose a new primary copy, knowing that there can be at most one majority partition, and hence at most one operational primary copy at any point in time.

In practice, a naming mechanism that maps logical item names to physical sites also routes read and write operations to the current primary copy. The primary copy returns the value of the item in the case of a read or acknowledges the successful receipt of a write before propagating it to the backups. This achieves replication transparency by shielding the database client program from the need to execute the replication control protocol. During two-phase commit, the primary votes its preparedness based on whether or not all of its backups agree.

Though this method appears overly simple when compared with other protocols, it is one of most widely implemented replication techniques. A popular example is the Sun NIS, or Yellow Pages [215], which is offered as a simple non-transactional network management solution for Unix-based systems. The primary copy, designated as the master NIS server, periodically updates other slave servers distributed across the network. We will also see this method used in a case of process replication in Chapter 3.

2.2.4 True Copy Token ROWA

In the *true copy token* scheme [148], each data item d has, at any point in time, either one *exclusive* token associated with it or a set of *shared* tokens. A write operation must acquire an exclusive token, while a read operation can proceed with a shared or an exclusive token. Whenever a copy needs to perform a write operation, it locates and obtains the exclusive token. If no exclusive token exists, then it must locate and invalidate all the shared tokens and create a new exclusive one in their place. To carry out a read, a copy locates another one with a shared token, copies its value, and creates and holds a new shared token. If no shared tokens exist, then the exclusive token must first be found and converted into a shared one.

The tokens act as a mechanism to ensure mutual consistency by creating conflicts among writes, and between reads and writes. In the event of a failure, only the partition containing a token is allowed to access a copy of d. If the token happens to reside in a partition containing rarely used sites, this effectively makes d unavailable.

The failure of sites that do not possess tokens will not block subsequent reads or writes. In the best case, up to $n-1$ copies can fail, so long as the remaining copy holds an exclusive token. However, the failure of a site holding a shared token prevents any future writes from proceeding. Furthermore, the failure of the site hosting an exclusive token, or of all the sites harboring shared tokens, renders the replicated item completely unavailable until these sites recover, unless a token regeneration protocol is employed. To regenerate a new exclusive token without jeopardizing its uniqueness, a majority of copies have to agree, and the site(s) that contained the previous token(s) in circulation must: (1) be *known* to have failed and (2) discard their tokens after they recover.

Another token-based algorithm is found in [199] where mutual exclusion in a system of n sites is achieved with at most n messages. In [179], a spanning tree of the network is used to locate the token, resulting in an average of $O(\log n)$ messages.

2.3 Quorum Consensus (QC) or Voting

The ROWA family of protocols implicitly favors read operations by allowing them to proceed with only one copy, while requiring write operations to be

carried out at all the up sites. This latter condition also means that ROWA algorithms cannot permit write operations to succeed when it is not possible to communicate with an up site because of a network failure. These two drawbacks: inflexible favoring of read availability, and inability to tolerate communication failures, give rise to the *quorum consensus* (QC) approach.

QC methods—often termed *voting* methods—in general allow writes to be recorded only at a subset (a *write quorum*) of the up sites, so long as reads are made to query a subset (a *read quorum*) that is guaranteed to overlap the write quorum. This *quorum intersection* requirement ensures that every read operation will be able to return the most recently written value. A site that participates successfully in an operation is said to have *voted* for it, hence the alternative name term for quorum consensus: *voting*. A great advantage of QC techniques is that they mask failures, with no need for intervention in order to resume operation after network partitions are repaired and merged.

Each QC method in this section employs a different style for specifying quorum membership, ranging from quorums that number a simple majority, to explicit enumeration of the membership of each possible quorum. The next Section, 2.4, discusses additional methods for characterizing quorum sets, which illustrate the wide range of design choices enabled by the quorum consensus approach. Quorums can be *static,* as when they are specified by votes that are assigned once and for all at system startup time. But, they can also be *dynamic,* if the sites are capable of reconfiguring the quorum specification, for example, in response to failures, load changes, or other system events. This section, together with Section 2.4, which looks at structural methods of quorum assignment, covers static methods, while Section 2.6 is devoted to dynamic methods.

2.3.1 Uniform Majority QC

The application of voting to the problem of replicated databases was first demonstrated in the uniform majority quorum consensus (or voting) method [205]. The DDBS is modeled as a group of sites that vote on the acceptability of query/update requests. An operation, be it a read or a write, succeeds if and only if a majority of the sites approve its execution. Not all the sites that vote for an operation need to carry it out on their local copies. In particular, a read operation needs to be executed at only one current copy while a write operation must be executed at a majority of the copies.

A majority requirement ensures that there be an intersection between the read and write operations of two transactions and that the underlying concurrency control protocol detects the conflicts and permit no conflicting updates. Resiliency to both site and network failures is achieved, but at high read and update costs; at least half of the n sites must participate, via voting, in every access of data items. This method works on the presumption that a network failure partitions the sites into two groups–a majority partition and non-majority partition. But repeated failures may splinter the system into many groups of sites, with none of the groups forming a majority.

2.3.2 Weighted Majority QC

The weighted majority QC algorithm [88], generalizes the notion of uniform voting. Instead of assigning a single vote per site, each copy d_s of a data item d is assigned a non-negative *weight* (a certain number votes) whose sum over all copies is u. The data item d itself is assigned a *read threshold,* denoted by r, and a *write threshold,* denoted by w, such that:

- $r + w > u$, and
- $w > u/2$. This constraint is required only if version numbers are used to determine the most current copy. An alternative method that uses timestamps [109] can have $w \leq u/2$.

A *read* (or *write) quorum* of d is any set of copies with a weight equal to at least r (or w). This additional constraint of read and write thresholds provides greater flexibility in vote assignment, besides ensuring mutual consistency of the copies just as in majority consensus.

In this protocol, a versioning mechanism determines the currency of the copies. Each copy is tagged with a *version number* which is initially set to zero. Each $w[d]$ by a transaction T is translated into a set of $w[d_s]$ operations on each copy belonging to some write quorum of d. This process involves reading all of the copies in the write quorum first, obtaining their version numbers, incrementing the maximum version number by one, and then performing the write with the new version number attached to each copy.

Each $r[d]$ is translated into a set of $r[d_s]$ operations on each copy of some read quorum of d. An $r[d_s]$ operation on a copy at site s returns its version number along with its value. The copy with the maximum version number, called the

current copy, is returned to the requesting transaction *T*. There is always a non-empty subset of the copies in a read quorum that are up-to-date. Thus, any read quorum is guaranteed to have a current copy.

This method works with any concurrency control algorithm which produces serializable executions. If a transaction T' reads a copy of d, then it can be safely concluded that it reads it from a transaction T that last wrote this copy of d. This is because T wrote into a write quorum of d and T' read from a read quorum of d, and every read and write quorum have a nonempty intersection. Also, the version number written by T will be larger than that written by all previous transactions. This ensures that T' will read the value written by T and not by some earlier transaction.

The quorum consensus algorithm, to its great advantage, requires no complicated recovery protocol. A copy of the data item d that was down and therefore missed some writes will not have the largest version number in any read or write quorum of which it is a member. Upon recovery, it will not be read until it has been updated at least once. Transactions will continue to ignore stale copies until they are brought current.

The algorithm is flexible. By altering r, w, and the weight assignment to the copies of each data item in the replicated database, the performance characteristics and the availability of the protocol can be altered. Assuming that the copies are equally distributed among the sites, a completely centralized scheme is achieved by assigning all of the votes to copies at a single site, and a completely decentralized scheme results from assigning equal weights to every copy. An optimal vote assignment scheme, which is theoretically valid for a limited number of copies, is proposed in [131].

Because of the need to poll multiple sites before every read operation, workloads with a high proportion of reads will not perform well under weighted majority QC. This can be remedied only by setting the r, w thresholds, and the individual site weights so that the algorithm degenerates into simple ROWA. Further, a large number of copies is required to tolerate a given number of site failures. For example, if we assume that a data item has one copy per site, each allocated a unary weight, then participation from at least three copies (and hence sites) is required to tolerate the failure of one copy, five sites for two failures, and so on. Goldman [93] exhibits a reconfiguration protocol for weighted majority QC that helps address the problem of low fault-tolerance to site failures.

Random Weights

Choosing weights and thresholds can be a difficult task for a system administrator. Kumar [129] proposes a randomized method , adapted from the process of annealing, that iteratively adjusts these parameters until they reach near optimal values. *Annealing* is a crystallization process where a molten substance is cooled in successive stages, each stage corresponding to a minimal energy state of the system at a certain temperature. The minimal energy level is a detectable steady state. The main characteristics of annealing are summarized as follows:

- A transition from a higher energy state to one of a lower energy is always most likely.
- A transition from a lower energy state to a higher energy state is possible, in a probabilistic sense, with the probability of such transition decreasing as the temperature is reduced.
- The probability of the transition from a current state, to a worse state (positive gain of energy) is given by: $\Pr[\Delta E] = exp(-\Delta E \frac{K}{T})$, where ΔE is a positive change (change to the worse) in the objective function; T is the temperature associated with a cooling stage; and K is a constant factor.

In this method, the equivalent of energy is availability (even though the latter is to be maximized while the former is to be minimized). Read and write quorums assume equal importance and each quorum is a set, the sum of whose votes equals to the majority of the sum of all votes. Each state, represented as a vector of n votes $(v_1, v_2, \ldots, v_n)$, is a possible solution to the problem (that is a vote assignment). The votes v_i, held by site i, are ordered by decreasing site reliability. A state change simulation involves selecting a pair of votes, v_i and v_j, from this vector and adding to (v_i, v_j), one of the following four ordered pairs $(1,1),(-1,-1),(1,-1),(-1,1)$.

Randomized voting assumes a single cooling stage and, therefore, temperature is treated as a constant. The probability of a transition from a current state to a state with worse availability, is given by: $\Pr[\Delta A] = exp(-K\Delta A)$, where $\Delta A =$ (new availability – best availability so far) and K is a constant. The steady state condition (maximum availability) is defined in terms of the iterations of the algorithm and is assumed to be achieved once there is no change in the best solution over a certain number of iterations.

2.3.3 Weighted Majority QC for Directories

Weighted majority quorum consensus for conventional data items is extended in [44] to handle directories. A *directory* is an abstract data type that maps *keys* drawn from a totally ordered set of constants called *key space* to *values.* The following operations modify and provide access to directories:

- *Insert(k: Key, v: Value).* Associates a value v with k, only if k is not already present in the directory. Once inserted, the key is said to be in the directory.
- *Update(k: Key, v: Value).* Associates a new value v with k, only if k is already present.
- *Delete(k: Key).* Removes k from the directory, only if k was already present.
- *Lookup(k: Key) : (Boolean, Value)* Returns TRUE and the value associated with k, if k is present, FALSE otherwise.

A direct application of weighted voting to directories has several concurrency limitations. If each directory was implemented as a single large read/write item, each copy of the directory would have a single version number. This would cause the serialization of all operations that modified the values associated with different keys in the directory. Any modification to the directory would require sending the entire updated directory to each representative in a write quorum, resulting in high communication costs. The solution to this problem is to associate a version number with each key in the directory. This idea is best illustrated by an example, taken from [44].

Consider the *3-2-2* directory suite shown in Figure 2.1, containing 3 sites, each with weight = 1, and having a read quorum of size 2 and a write quorum of size 2. Let another key 'b' be inserted into directory representatives *A* and *C* with version number 1. A request to look up 'b' in *B* and *C* at this point will yield the responses "not present" and "present with version number 1" respectively. Now if 'b' is deleted from *A* and *B,* the same request will elicit the same response. This ambiguity arises from failure to associate a version number with keys that are *not present* in the directory. Hence, version numbers need to be associated with every possible key in the key space at each representative by partitioning the key space into disjoint sets and associating a version number with each set at every representative.

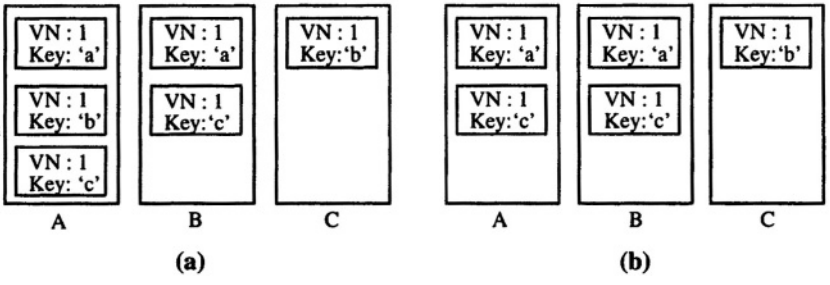


Figure 2.1 Directory suite: (a) after inserting 'b'; (b) after deleting 'b'.

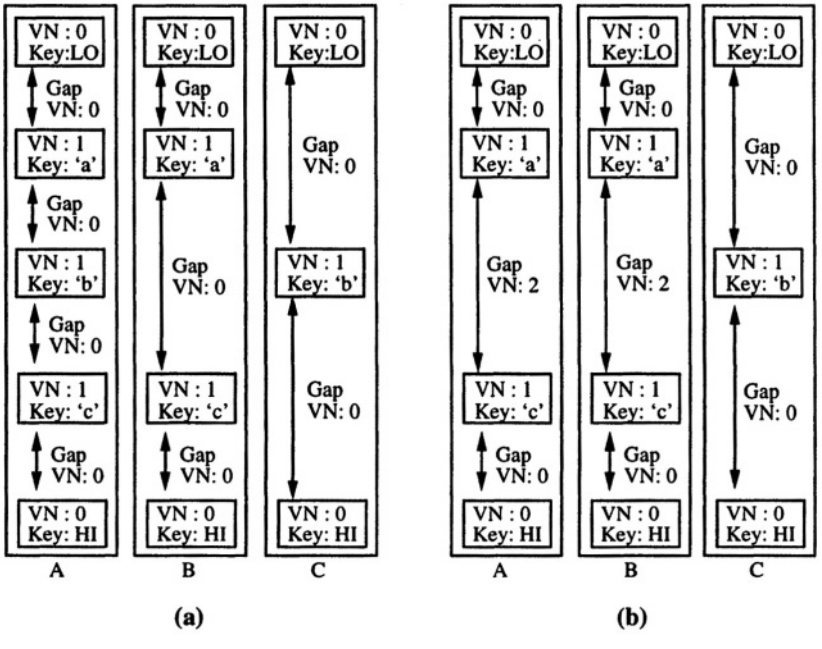


Figure 2.2 Directory suite with dynamic partitioning of key space (a) after inserting 'b'; (b) after deleting 'b'.

Static partitioning techniques divide the key space into a number of fixed ranges, which can act as a constraint on concurrency. Hence *dynamic partitioning* is used, wherein a partition is created for each key that has an entry in the representative, as well as one for each range of keys between successive entries. A range of such keys is called a *gap* (shown in Figure 2.2). Both keys and gaps are assigned version numbers, and two special keys *LO* and *HI* are associated with the first and last gaps in each representative.

Now, if 'b' were inserted in *A* and *C,* and subsequently deleted from *A* and *B*, the deletion would cause the gaps preceding and succeeding the partition of 'b' to coalesce into a new gap with an updated version number 2. A request to look up 'b' in *B* and *C* will now yield the responses "not present with version number 2" and "present with version number 1" respectively, thereby clearly indicating that the key has indeed been deleted.

Bloch, Daniels, and Spector in [44] provide a comprehensive discussion on the maintenance of replicated directories using remote procedure calls and type-specific locking mechanisms. Sarin, Floyd, and Phadnis in [183] add two extensions to the basic algorithm: (1) allowing control over individual keys in a directory to migrate independently; and (2) providing means to define relationships between the keys, such that if control over a given key migrates, then the keys that are dependent on it are automatically migrated as well. Bloch [43] proposes a distributed data structure called *replicated sparse memories* (*RSM*), which are more generalized and simpler than directories. *RSMs* map *addresses* to values, and have only three access operations–*Read*, *Write, and Erase.* The generality of *RSMs* facilitates their adaptation to several useful data types such as single/multiple indexed record sets, queues, and counters.

2.3.4 General QC for Abstract Data Types

With the exception of weighted majority QC for directories, the methods studied thus far perform replication based on the *values* of objects being stored at distributed sites, thus requiring that read and write operations constitute the fundamental operations on the replicated objects. Alternatively, if necessary and sufficient quorum constraints are derived from an analysis of the data type's algebraic structure, better availability and more flexible reconfiguration are observed to be possible. This approach differs from the value-based one in that it requires knowledge of the semantics of the data being replicated. Data is assumed to be encapsulated in *Abstract Data Type (ADT) objects.* Access to the ADT is restricted to the invocation of a fixed set of operations, whose

algebraic properties are explicitly specified to the system. Since different copies of an object do not necessarily contain the same set of entries, they are called *representatives,* not copies.

General quorum consensus (GQC) [109] exploits type-specific properties of the data to achieve better availability and more flexible configuration than is possible with value-based replication methods. In this method, rather than replicate the value of a data item at multiple sites, the *events* that affect the item are replicated and logged. The events stored at certain subsets of an object's set of representatives, when taken together, yield a *complete history,* or log, of all the events that affected the object. This provides a complete representation of the object's state.

The algorithm uses the model of a distributed system proposed in [97], which is that of a network of servers, each managing typed objects. Clients communicate with each server using a standard protocol consisting of request/response message pairs. Each *object* is an instance of some—possibly user-defined—abstract data type (ADT) [146]. An object can also be envisioned as a basic data container having a *type,* which defines a set of possible *states* and a set of primitive *operations* [109]. The request/response pair of messages resulting from an operation invocation and response can be considered an *event* experienced by the object. An object's state is modeled by a sequence of events, called the *history,* or log. A *specification* for an object is the set of possible histories for that object. A history is termed *legal* by virtue of its inclusion in an object's specification.

The replication method is assumed to operate on top of an atomic transaction system but is independent of the atomicity mechanism. A replicated object is stored in a set of sites, called *object representatives,* each of which provides long-term storage for a *partial history* of events that affect the object's state. A quorum for an operation is a set of *representatives*[3] whose cooperation suffices to successfully complete the operation. An event is an operation invocation, together with the corresponding response composed of the operation, its arguments and results.

Client transactions issue operations on objects to be carried out by *front end* servers capable of operating on objects of a given specific type. The front-end queries an *initial quorum* of repositories for their local histories and merges them together to produce a history, called the *view,* that contains all the past

[3]In this case, 'replica' is not the correct term, since sites do not store complete information about the state of the replicated object.

events relevant to the requested operation. The view is applied to a suitable initial value of the object to obtain a current value on which the operation can be carried out. Once results are produced, the new event—the operation, its arguments, and result—are appended to the view.

The updated view is then recorded at a *final quorum* of the object's representatives. The sites in the final quorum are now termed the *participants* in the event, and the event is said to have *occurred* at these sites. Each object has a minimal *serial dependency* relation, which embodies constraints sufficient to guarantee that information flows between any two events that must observe each other. The initial and final quorums are chosen using quorum intersection relations that are equivalent to serial dependency relations. Since the final quorum is typically smaller than the entire set of representatives, the local history generally does not contain all the events experienced by the object.

The above ideas are elucidated in [100] with the help of an example *queue* abstract data type, which supports two operations *enqueue* (add an element to the queue) and *dequeue* (remove an element from the queue and return its value). The state of the queue is represented as a log of (*timestamp, event*) entries. For the *queue* to work in a FIFO manner, a *dequeue* invocation must be able to observe at least one copy from every previous enqueue event, *i.e.,* the dequeue operation *depends* on the enqueue operation. In addition, and to avoid dequeuing the same element twice, every *dequeue* invocation needs to observe at least one copy of every previous *dequeue, i.e.,* the *dequeue* operation depends on itself. The resulting quorum intersection relations are:

1. $FQ(enqueue) \cap IQ(dequeue) \neq \emptyset$, $\forall$ FQ(*enqueue*), IQ(*dequeue*).
2. $FQ(dequeue) \cap IQ(dequeue) \neq \emptyset$, $\forall$ FQ(*dequeue*), IQ(*dequeue*).

where FQ and IQ denote the *final quorum* and the *initial quorum* respectively.

Suppose that the queue is replicated at 9 sites, labeled $s_1 \ldots s_9$, which are grouped into three clusters of three sites each $(s_1 s_2 s_3, s_4 s_5 s_6, s_7 s_8 s_9)$, such that failures of sites in the same cluster are statistically correlated. Assume that we would like the *enqueue* operation to be available despite six site failures or less, and the *dequeue* to be tolerant of up to two site failures. If the initial and final quorums are specified using the number of votes, the quorum specification must be:

- $|IQ(enqueue)| = 0$

- |FQ(*enqueue*)| = 3
- |IQ(*degueue*)| = 7
- |FQ(*dequeue*)| = 3

which does not take account of failure correlations. However, if the quorums are specified as *sets,* a more refined configuration is possible:

- $IQ(enqueue) = \emptyset$
- $FQ(enqueue) \in \{ \{s_1, s_2, s_3\}, \{s_4, s_5, s_6\}, \{s_7, s_8, s_9\} \}$
- $IQ(dequeue) \in \{\{x,y,z\} : x \in \{s_1, s_2, s_3\}, y \in \{s_4, s_5, s_6\}, z \in \{s_7, s_8, s_9\}\}$
- $FQ(dequeue) \in \{ \{s_1, s_2, s_3\}, \{s_4, s_5, s_6\}, \{s_7, s_8, s_9\} \}$

The availability remains the same under the refined configuration, even though the size of *dequeue* initial quorums is smaller. The reason this is the case is that, if six failures occur, then with high probability, the six failures are confined to two clusters, and the remaining cluster can still form an *enqueue* quorum. The availability of *dequeue* also remains the same, since clearly no two failures can prevent the formation of either an initial or a final *dequeue* quorum. But the significant advantage here is the generality and increased performance: *dequeue* needs the cooperation of only five sites against the seven required in the voting method.

A client accesses the queue by issuing *enqueue* and *dequeue* operations to a front end. The front end determines the quorums sufficient to run a received operation, say *dequeue,* from the quorum specification. If its subsequent effort to communicate with an IQ(*dequeue*) fails, it either blocks or aborts the transaction. If it succeeds, owing to a history *closure* property, it is guaranteed that merged histories from a valid initial quorum will contain all the events that may be needed for *dequeue* to return the correct result. Upon obtaining the correct result, the front end attempts to communicate with a valid FQ(*dequeue*) at which the result with the merged history can be stored. Upon success, it returns the result to the client that requested *dequeue*; otherwise it aborts.

This method is general enough to encompass most of the voting methods, since it is applicable to objects of arbitrary type. The generality also arises from specifying the operation quorums as sets of sites, rather than numerical votes.

The data type specification determines the criteria for a correct implementation, leading to fewer constraints on availability. It also provides the flexibility required for dynamic reconfiguration (see Section 2.6.4).

2.3.5 Hybrid ROWA/QC

We have mentioned that the *QC* algorithm is expensive if communication failures are infrequent. This hybrid ROWA/QC algorithm, dubbed *missing writes* algorithm by its originators [73] proposes a method to reduce this cost which adopts a *ROWA* strategy during reliable periods of operation and switches to the *QC* method during periods when there are site or communication failures. A transaction works under the *ROWA* method in its *normal mode* and switches to a *failure mode* when it becomes *aware of missing writes* at which point it obeys the *QC* protocol.

The presence of missing writes indicates that some copy d_s does not reflect the updates applied to other copies of d. A transaction T that has been submitted to the replicated database becomes aware of a missing write for copy d_s in some execution if: (1) either T performs $w[d]$ but is unable to perform $w[d_s]$, or (2) some transaction T' is aware of a missing write for d_s, and there is a path from T' to T in the *serialization graph* [26] of that execution.[4]

If a transaction T becomes aware of a missing write for some copy d_s that it has *already* read, then T must abort. However, if T learns of the missing write on d_s before reading this copy, it must switch to the *QC* mode and read from copies that are not missing any writes. That is, T reads a quorum which should include at least one copy that did not miss any writes.

The first case discussed above is simple. Transaction T detects the missing write when it times out on the acknowledgment from site s. However, for transaction T to learn about the missing write on d_s before it accesses d_s, additional mechanisms are required. Such early detection of missing writes is important, if performance degradation is to be controlled during periods of failure [107]. To detect missing writes before access, a versioned missing write list (*MWL*) is associated with each copy of the data item. A transaction T that accesses the copy tags the missing writes that it is aware of in this list. When another transaction T' accesses the same copy but in a conflicting mode, it becomes aware of the missing writes. T' then propagates this *MWL* to all other sites that it visits. Upon recovery from site failure, a recovering site s

[4]The definition of a serialization graph is provided in Section C.

has to bring the local copy up-to-date using a *copier transaction* and also has to eliminate the entry corresponding to d_s from the *MWL*s of all other copies.

The proof of correctness and the algorithm for the above operations are discussed in detail in [73]. The algorithm maintains availability during failures but at the expense of increased complexity and communication costs. Other schemes that compete with ROWA/QC include QC with reconfiguration.

2.4 Quorum Consensus on Structured Networks

Many of the more recent data replication algorithms emphasize *cost optimization,* in addition to fault-tolerance. One optimization is the savings in communication costs obtained by decreasing the number of sites that need to be contacted in order to ensure mutual consistency. The new protocols try to reduce the number of sites from $n/2$ in the case of majority voting to a lower number by imposing a *logical structure* the set of sites. Sections 2.4.1, 2.4.2, and 2.4.6 discuss matrix based algorithms, while Sections 2.4.4 and 2.4.5 cover tree-based algorithms.

The basis for most of these protocols is the generalization of quorums found in [86,109,133]. We define some of the important concepts in this section before moving on to discuss the algorithms. A read (or write) *quorum* is the minimal set of copies whose permissions are required for a read (or write) operation to proceed [16]. A collection of quorums used by an operation forms a *quorum set.* When each quorum intersects with every other in the quorum set (intersection property), and no quorum is a superset of another (minimality property), the quorum set is called a *coterie* [86, 133]. In other words, a coterie is a set of sets with the property that any two of its members have a nonempty intersection.

2.4.1 $\sqrt{n}$ Algorithm

The $\sqrt{n}$ algorithm [142] requires the participation of all the n sites that constitute a distributed system with an error-free communication network. It associates a quorum with each site in the network so that this quorum has a non-empty intersection with all quorums corresponding to other sites. That is,

the set of quorums forms a coterie[5]. A process must obtain the consensus of all sites in the quorum associated with its home site in order to achieve mutual exclusion. Since this set intersects with every other quorum, mutual exclusion is guaranteed.

The protocol achieves mutual exclusion with $c\sqrt{n}$ messages, where $3 \leq c \leq 5$. Whenever a site s_i issues a mutual exclusion request, the protocol requires it to obtain consensus from a quorum Q_i of sites, such that all the following conditions hold:

- $Q_i \cap Q_j \neq \emptyset$. There must exist at least one site in common among every pair of quorums, Q_i and Q_j, that serves as an arbitrator when members of Q_i disagree with those of Q_j as to which process is due to enter its critical section.
- $s_i \in Q_i$. The site issuing a request obtains permission from itself without any message transmission.
- $|Q_i| = K$. Each site needs to send and receive the same number of messages to obtain mutual exclusion.
- $|\{Q : s_i \in Q\}| = M$, where M is constant. Each site serves as an arbitrator for the same number of sites bearing equal responsibility for mutual exclusion control.

The above requirements translate into a relation $n = K(K - 1) + 1$. The problem of finding a set of Q_i's that satisfy these conditions is determined to be equivalent to finding a finite projective plane of n points. It is shown that such a set exists if $(K - 1)$ is a power of a prime, and the optimal value of K for a given n is determined to be $K = \sqrt{n}$. Thus each member Q_i of the quorum set has $|Q_i| = \sqrt{n}$. In other words, this algorithm represents a special case of coteries where each member is of equal size given by $\sqrt{n}$. For cases where n cannot be expressed in the above form, degenerate quorum sets are created. The algorithm for forming these sets is given in [142].

The significance of these criteria for specifying quorums is in the equal distribution of load among all the sites at $O(\sqrt{n})$ cost, whereas other quorum consensus methods permit the imposition of greater responsibility on sites with higher weights.

[5] A coterie is a set of sets such that every two members have a non-empty intersection.

2.4.2 The Grid Protocol

The Grid Protocol [52] achieves mutual exclusion with $O(\sqrt{n})$ messages and emphasizes load sharing. Sets of sites that store data replicas are logically arranged into a grid network with sites in a row and N sites in a column of the grid, numbered 1,2,..., and 1,2,..., N respectively, such that $M \times N \leq n$. As in Maekawa's algorithm, a reliable communication network is assumed over which a multicast mechanism is used for message transmission.

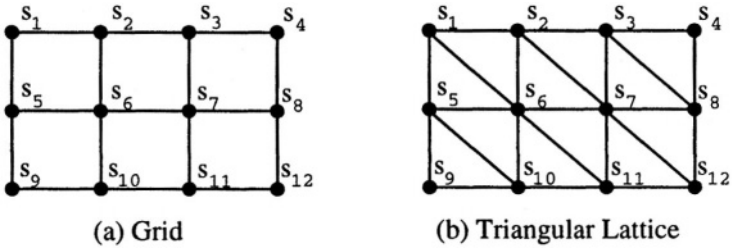


Figure 2.3 An example of (a) grid, and, (b) triangular lattice arrangement of sites

A read coterie in this method contains groups of sites, each containing *exactly one* of its member sites in each column. A write coterie consists of groups that contain sites of a read group and all sites of a column. A set of sites G is defined as a *C-cover* if each column intersects with G. For example, the set $\{s_1, s_2, s_7, s_8\}$ in Figure 2.3 (a) is a C-cover.

A read operation selects a random permutation π_r of the M sites in a random row r. π_r determines the order in which the sites are interrogated to read-lock a C-cover. It also distributes the load evenly over the sites. An *R-cast* (for read multicast) operation sends a request to all the sites in π_r to obtain a C-cover. If a C-cover is not obtained, then more permutations, are formed for successive rows until a C-cover is read-locked. When this occurs, the read operation is guaranteed to find at least one copy of the latest value written to the data item, among the sites in the C-cover. If no C-cover is obtained, the operation is aborted, and all read-locks released.

A write operation first locks a C-cover using the read protocol and then proceeds to lock all the sites in a random column c in the order determined by π_c, a random permutation of the N sites in a column. For any two writes to execute

concurrently, each must lock a C-cover and all sites in a column. But if two operations have each locked a C-cover, neither can obtain write-locks from all the sites in any column since one of the operations will get a negative response from the sites already locked by the other. The same can be shown for a read-write conflict. Thus mutual consistency is ensured.

The primary advantage of the grid protocol lies in its ability to distribute load and hence improve response times, but other quorum consensus schemes can similarly benefit from randomization in the selection of quorums to poll. The idea of using random permutations to sequence the requests sent to individual sites is unlikely to be useful in practice. First, it precludes parallel polling of quorum members, which can be faster, and, second, it eliminates the opportunity of using a fixed sequence, which is a common technique employed to reduce the likelihood of unnecessary deadlocks.

2.4.3 Asymptotically High Availability

We say that availability is asymptotically high if it approaches 1 as the number of data copies approaches ∞. Ranagarajan *et al.* [178] propose a protocol based on Maekawa's algorithm that achieves asymptotically high availability if $p > 0.5$, p being the availability of a single data copy. But this is achieved at the cost of an increase in quorum size to $O(\sqrt{n \log n})$.

The **hierarchical grid protocol** [130] organizes data copies into a multilevel grid structure. Each grid at level 0 represents the real data copy, and each grid at level k is defined a grid of $m_k \times n_k$ level-$(k-1)$ grids. The quorum size here is $O(\sqrt{n})$, and the algorithm exhibits asymptotically high availability. But if the grids at each level are square, it requires $p > 0.63$ to achieve this. Also, no less than $\sqrt{n}$ data copies have to be collected upon failure of a single copy.

The **triangular lattice protocol** [218] improves upon the above methods by providing asymptotically high availability with $p > 0.5$. Data copies are arranged as a M N triangular lattice, whose interconnection can be conveniently modeled by a graph $G = (V, E)$, where V represents the set of site coordinates on a unit grid, and E the communication topology that interconnects them, as shown in Figure 2.3b.

A *horizontal crossing* is defined as a path $s_0, s_1, \ldots, s_i$ in G, such as s_0 is in column 1 and s_i in column N. A *vertical crossing* has the dual definition, with s_0 being in *row* and s_i in row M. For example, in Figure 2.3b, $\{s_1, s_2, s_7, s_8\}$ and

$\{s_1, s_6, s_{10}, s_{11}, s_7, s_{12}\}$ are horizontal crossings; $\{s_1, s_6, s_{10}\}$ and $\{s_1, s_6, s_7, s_{11}\}$ are vertical crossings. A read quorum is a set of sites that contains either a vertical crossing or a horizontal crossing. For a write quorum, both a vertical and a horizontal crossing are required. The proofs that show that these crossings provide asymptotically high availability are based on the site percolation problem on a plane triangular lattice [121] and described in [218].

2.4.4 Tree Quorums

A mutual exclusion protocol which is tolerant to both site and communication failures in a network that is logically structured in the form of a tree is discussed in [6].

Given the simple case of a binary tree which has a well-known root, the process at some site requests a *tree quorum* by calling a recursive function with the root as its parameter. The algorithm tries to construct a quorum by obtaining permissions from sites located on any path beginning with the root and ending with any of the leaves. If it is unsuccessful in finding a path because of the failure of some site s_i, it must substitute for that site with two paths, both of which start with the children of s_i and terminate with leaves.

To summarize, when all sites are accessible, the quorum is any set $\{s_1, s_2, \ldots, s_n\}$ where s_1 is the root and s_n a leaf, and s_{i+1} is the child of s_i for all $i <$ n. In case of failures, for each inaccessible site s_i, a path of sites starting from s_j and s_k (the children of s_i) and ending with leaves is recursively included into the tree quorum.

The following group formations are possible for the tree shown in Figure 2.4.

Failed Sites	**Tree Quorums**
None	$\{\{s_1, s_2, s_4\}, \{s_1, s_2, s_5\}, \{s_1, s_3, s_6\}, \{s_1, s_3, s_7\}\}$
s_1(root)	$\{\{s_2, s_4, s_3, s_6\}, \{s_2, s_5, s_3, s_6\}, \{s_2, s_4, s_3, s_7\}, \{s_2, s_5, s_3, s_7\}\}$
s_2	$\{\{s_1, s_4, s_5\}, \{s_1, s_3, s_6\}, \{s_1, s_3, s_7\}\}$
s_3	$\{\{s_1, s_2, s_4\}, \{s_1, s_2, s_5\}, \{s_1, s_6, s_7\}\}$
s_1, s_2	$\{\{s_4, s_5, s_3, s_6\}, \{s_4, s_5, s_3, s_7\}\}$
s_1, s_3	$\{\{s_2, s_4, s_6, s_7\}, \{s_2, s_5, s_6, s_7\}\}$
s_1, s_2, s_3	$\{\{s_4, s_5, s_6, s_7\}\}$

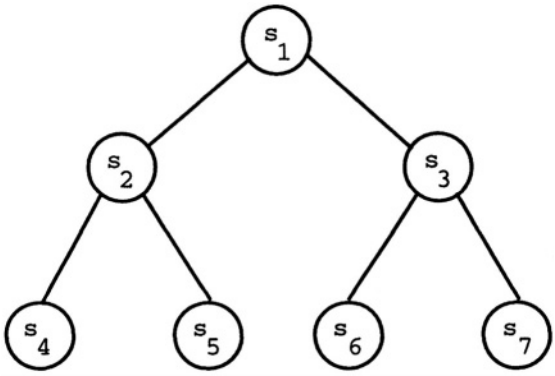


Figure 2.4 Tree quorum formation for a sample 7 site network.

The correctness of the algorithm is established by the fact that tree quorums satisfy the intersection and minimality properties of coteries. Quorums can be formed with only $\lceil \log n \rceil$ sites in the best case and degenerating to $\lceil \frac{n+1}{2} \rceil$ as the number of failures increase thus demonstrating graceful degradation [143].

For the algorithm to fail to form a quorum, it is necessary that more than $\lceil \log n \rceil$ sites fail. At the same time, it is not necessary that the number of concurrent failures exceed $\lceil (n+1)/2 \rceil$, for the replicated item to become unavailable. For example, if the sites s_1, s_2, and s_4 in Figure 2.4 are inaccessible, the remaining set of sites $\{s_3, s_5, s_6, s_7\}$ do not include a tree quorum, though they constitute a majority quorum. The extension of the binary tree algorithm to a k-degree tree yields $\lceil \log_k n \rceil$ and $\lceil \frac{(k-1)n+1}{2} \rceil$ as the number of sites in the best and worst cases, respectively. Given a value t, which represents the maximum number of possible site failures out of a total of n sites, the degree k of a tree that can guarantee formation of quorums is given by the relation $\lfloor \frac{k^{t+1}-1}{k-1} \rfloor = n$. A discussion of vote allocation for a tree topology that provides optimal solutions in polynomial time is presented in [147].

2.4.5 Hierarchical Weighted Majority QC

The hierarchical quorum consensus (HQC) algorithm [127] proposes a method to minimize the number of objects required to participate in a quorum and hence the synchronizing communication between them. A set of physical ob-

jects are logically organized into a multi-level tree of depth k with the root as level 0 and l_i *logical* (sub)objects at level i. The physical objects form the leaves of the tree at level k. Consequently, there are l_k level k (physical) objects for each logical object in level $k-1$. The total number of physical objects in the system is therefore $l_1 \cdot l_2 \cdots l_m$.

A read (or write) quorum for an object at level i, denoted by r_i (or w_i), is formed by assembling r_{i+1} (or w_{i+1}) subobjects at level $i+1$. The process is initiated at the root of the hierarchy and proceeds recursively down to the leaves, resulting in a quorum of physical objects. A concurrency control scheme that forms such a quorum is shown to be correct, provided $r_i + w_i > l_i$, and $2w_i > l_i$, for all levels $i = 1, \ldots, k$.

The number of physical data items in the read quorum is $r_1 \cdot r_2 \cdots r_m$, and that for the write quorum is $w_1 \cdot w_2 \cdots w_m$, and the write quorum is at least as large as the read quorum. The above results can be extended to form a family of algorithms for replicated data $\{l_i, r_i, w_i, i = 1, \ldots, k\}$ such that:

- $\prod_i l_i \geq n$.
- $r_i + w_i \geq l_i$.
- $2w_i \geq l_i$.

A write quorum of minimum size can be obtained by constructing a hierarchy whose number of levels is obtained as follows. Given n objects, n is repeatedly rewritten in the form $n_{new} = 3^a \times 5^b$. The value a denotes the number of levels of the tree, b is a constant that assumes a value 0 or 1, and n_{new} is the least upper bound of n. The number of levels is a if b is 0, and $a+1$ if b is 1. It is shown that the HQC algorithm requires a minimum of $n^{0.63}$ messages as compared to $n/2$ for majority voting, and $\lceil \frac{n+1}{2} \rceil$ in case of *QC*.

A performance study [128] with traffic and availability as the main criteria shows that HQC outperforms majority and dynamic voting in message cost and offers better availability for a low connectivity network with highly reliable links. The average number of direct links per site is used as the basis to define connectivity. HQC is also impervious to the frequency of transactions on the replicated objects, unlike dynamic voting (see Section 2.6.1 below), which is dependent on changes in the network state.

2.4.6 Multidimensional Weighted Majority QC

Multidimensional (MD) voting [55] is a scheme for specifying quorum sets, that generalizes weighted voting to achieve more flexible specification of quorum sets. In a distributed system of n sites, the vote value assigned to a site is a k-dimensional vector of non-negative integers. Formally, the MD vote assignment is a matrix $V_{n \times k}$ in which the element v_{ij} denotes the vote assignment to site s_i in the jth dimension. Similarly, the quorum assignment is also a k-dimensional vector $\vec{q} = (q_1, q_2, \ldots, q_k)$, of positive quorum thresholds, together with a number $l, 1 \leq l \leq k$, denotes the number of dimensions of vote assignments for which the quorum must be satisfied.

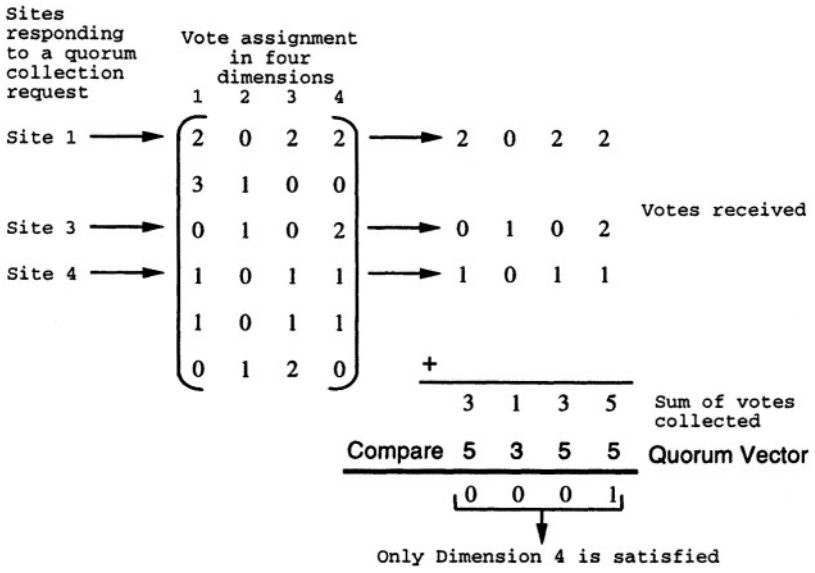


Figure 2.5 Voting procedure in multidimensional voting.

To form a quorum, there are now two levels of requirements: (1) at the vote level, the number of votes received per dimension must be greater than or equal to the quorum requirement in that dimension; and (2) at the dimension level, quorums must be collected for a number of dimensions greater than or equal to l. MD-voting with the quorum requirement in l of k dimensions is termed MD(l, k)-voting.

In a sample MD-vote assignment for a system with 6 sites shown in Figure 2.5, let us assume that sites s_1, s_3 and s_4 respond to a vote request. The vector sum of these voting vectors is compared dimension by dimension with the vector quorum assignment. We see that the quorum requirement is satisfied only in the fourth dimension. The operation can now execute only if MD(1,4) voting is used and will fail for $l > 1$.

The extra dimensions give this method the flexibility to represent various voting schemes such as QC, grid, tree, and hierarchical voting. For example, QC simply becomes MD(1,1)-voting. Moreover, when used as a replication control protocol, an MD$(1, k)$-voting and quorum assignment has the power to represent mutual exclusion coteries in cases where a single dimensional scheme would fail. The vote assignment and quorum vector in Figure 2.5 is one such example. A coterie (see Section 2.4.1 for a definition),

$$Q = \{\{s_1, s_2\}, \{s_1, s_3, s_4\}, \{s_1, s_3, s_5\}, \{s_1, s_4, s_6\},$$

$$\{s_1, s_5, s_6\}, \{s_2, s_3, s_6\}, \{s_2, s_4, s_5\}\}$$

cannot be formed by a *SD-* assignment. The proof of the method's correctness and the MD-voting assignment algorithm is provided in [55].

2.5 Reconfiguration after Site Failures

The degree of availability afforded by a replication protocol decreases as a result of failures. When failures are long-lived, the probability of multiple concurrent failures heightens, threatening to render the replicated item completely inaccessible. The protocols in this section handle this situation by reconfiguring so as to shrink the required quorums, or to regenerate new copies to replace those stored on the failed sites.

In order for the system to reconfigure itself after a site failure, it must first be able to *detect* this failure reliably. The difficulty in doing so stems from the similarity between the failure of a site, and simple slowness of communication or computation. The technical condition whose satisfaction prevents such confusion from causing ill-effects, is that all sites must essentially agree on the new configuration. More precisely, all up sites need to agree on the new configuration, and all down sites must either know that they are down, or at least be prevented from acting to jeopardize consistency. When a failed or isolated site needs to rejoin the protocol, it typically must run a recovery procedure that is

designed to inform the recovering site of the new configuration, and to update its local copy if necessary.

2.5.1 Primary Copy ROWA

This scheme has already been discussed in Section 2.2.3, but must be mentioned here because its basic method of operation can be construed as employing reconfiguration. A site failure triggers selection and installation of a new primary, or the reduction in the number of backups that must be updated, as a result of failure of the primary or a backup, respectively. Both of these actions amount to reconfiguring the replicated data item, hence the inclusion of primary copy ROWA in this section.

2.5.2 Directory-based ROWA-Available

The basic available copies method of Section 2.2.2 can be extended to a *directory-oriented available copies* algorithm [24, 26]. *Directories* containing entries for each data item d are replicated and treated like ordinary data items. Each entry in the directory for d, denoted as $D(d)$, lists all of d's copies. An *include transaction* (denoted by *IN)* and an *exclude transaction* (denoted by *EX)* are defined for creating and destroying new copies respectively. When a site s recovers from failure or needs to create a copy of d, it runs a new transaction $IN(d_s)$ which adds d_s to each available copy of $D(d)$, thereby declaring d_s available. *IN* brings d_s current by reading from an available copy of $D(d)$ to find an available copy, then copying it to d_s.

When a site s fails, another site s' that notices this failure runs a new transaction $EX(d_s)$ to remove d_s from every available copy of $D(d)$, thereby declaring d_s unavailable. To do this, *EX* reads some copy of $D(d)$, removes d_s from the list of available copies, and writes the updated list into each available directory copy.

Subsequently, transactions no longer attempt to update copies at down sites. Dynamic creation and deletion of copies is possible, and recovery and integration are facilitated by the directories. Other uses of directories can be found in [24, 44, 107]. The downside of this method is the overhead associated with maintaining directories and additional processing, due to possible failures of sites containing the directory copies.

2.5.3 Regenerative ROWA

Regeneration of failed replicas may be a suitable approach when failures are accurately detectable, and storage resources are abundant. New copies can invariably be created faster than sites can be repaired. Hence, availability can be enhanced by creating new replicas on additional sites in response to changes in the operational subset of the system.

The regeneration algorithm [174] works in a ROWA manner, providing increased availability in a *partition-free* distributed system. Among the n sites, $n-k$ hold copies of a data item d, with the other k sites being *spare* sites. New copies can be installed in these spare sites when the inaccessibility of one or more of the current replicas is detected by the algorithm.

A read operation requires that there be at least one operational site at which $r[d_s]$ is performed. A $w[d]$ requires that there be $n-k$ valid copies of d to record the write. If more than k of the n sites fail, $w[d]$ does not go through.

A *recovery* operation allows recovering sites to rejoin the replication if the replica at that site is current. If not, then the recovering replica is discarded. It is not possible that a copy d_s at a recovered site s find both of the following conditions to be simultaneously true: (a) that it is current, and (b) that there are already $n-k$ replicas. Condition (a) implies that d_s is the copy responsible for restoring the total number of copies to the original value of $n-k$ and no writes took place while s was down. Condition (b) implies that some write actually took place; *i.e.,* that some other copy has been regenerated and d_s is no longer current.

The regeneration of copies is done only during writes at which time all failed replicas are rendered obsolete, and hence invalid by the above recovery procedure. Manual intervention is required in the event of total system failure. This factor, along with high communication costs and lack of resiliency to network failures, makes this algorithm inefficient for certain types of networks [140].

2.5.4 Regenerative ROWA-Available

Regeneration significantly improves the write availability of the *available copies* method [154]. New replicas may be regenerated to replace missing replicas whenever it is feasible, but a write $w[d]$ is translated to writes on all *available* copies, even if some or all of the missing replicas could not be regenerated. Data

can then be read from any available copy. When a site s restarts following a failure, if there is another site s' that holds the most recent version of the data, then s can recover immediately by obtaining the most current value from s'.

Regeneration also helps to speed up the recovery process from a total system failure. In the basic ROWA available copy (discussed in Section 2.2.2), the recovering sites do not know which one of them contains the most up-to-date copy until the last site to fail can be found. A regenerative protocol, on the other hand, has to maintain the replica states to enable regeneration. This feature benefits the available copy scheme during recovery from total failure, in terms of the knowledge of the most current copies. Recovery from total failure only requires the recovery of sites belonging to the last set of available replicas.

If no spare sites are available, the algorithm degenerates to basic or directory-based available copy, Spare sites have the effect of accelerating partial recoveries when replicas can be generated faster than sites can be repaired.

2.5.5 Regenerative Quorum Consensus

Regeneration can be incorporated in the basic QC algorithm (see Section 2.3) by introducing the concept of *generation* to determine the currency of replicas. A generation is defined as the set of replicas that have participated in a particular regeneration [140] with their participation indicated by a *generation number* tagged onto them.

Read and write quorum definitions remain the same as in basic QC. Spare sites replace failed ones and are assigned the roles (*e.g.,* votes) of the failed ones, ensuring that in the system is retained across all generations. When the protocol determines that there are fewer than the desired number of replicas in the system but still enough to collect a quorum, it initiates a regeneration. The replicas involved in the regeneration must be members of the current generation.

If a quorum exists and there are spare sites available, new replicas are placed on some of them by copying the data with its version and generation numbers from the most recent copy. The spares are then transformed into full copies and all participating sites increment their generation numbers. This disallows non-participating sites from taking part in future quorums, thus preserving mutual consistency. Excess replicas will have obsolete generation numbers at the next quorum request and are relegated to spare site status.

Site recovery protocol is not required. A recovering replica will realize its obsolescence at the next quorum and can be transformed into a spare. One-copy serializability is preserved by this method, as proved in [140] and [163].

2.5.6 QC with Witnesses

This method is an extension of the basic QC algorithm, in which some copies of the data items are replaced by mere recordings of their version numbers possessing their vote assignments, but with no data attached [162]. These lighter representatives of the state of the data item are called *witnesses.*

Witnesses participate just as conventional copies in the formation of read and write quorums but with the additional restriction that every quorum must include at least one current copy. This implies that one cannot read from a witness or use it to bring a copy current. The version number always reflects the most recent write recorded by the witness. The minimum number of copies to maintain consistency is thus reduced from three in the case of basic QC to two in this method. Moreover, witnesses have negligible storage costs with a trivial update mechanism of just increasing the version number. However, the above advantages can be claimed vis-a-vis QC only if some of the would-be QC replicas are converted to witnesses, but this reduces availability under certain circumstances. An extreme example would be when none of the replicas that contain the current version of the data item is available, yet some witnesses that know the current version number are—a total system failure.

An enhancement to the basic scheme is to let witnesses be upgraded to replicas and replicas transformed into witnesses, depending on the number of actual replicas available in the system and constraints on communication overhead. Another utility is conversion of obsolete copies into *temporary* witnesses, thus enabling a faster update of these copies than with conventional *QC.* In *QC,* a background process copies the most current value of the data item into obsolete copies read by a read quorum [88] which can be slow.

Volatile witnesses [161] are witnesses stored in volatile memory and hence in diskless sites which results in greater savings in storage space. But volatile witnesses will not be able to recover on their own from a site failure. They will need to read the current version number of the replicated data object from a valid quorum of witnesses and replicas before making any contribution to a quorum formation. Regenerable volatile witnesses [164, 196] is a mechanism proposed to remove the above limitation.

2.5.7 QC with Ghosts

Ghosts are entities similar to *witnesses* in functionality, but different in structure. A ghost [200] is a *process* without storage which takes the place of an unavailable copy. Failures are assumed to occur only at gateways which partition the network into *segments.* In the event of a site failure, a boot service invokes a ghost within the same segment with the votes of the crashed object assigned to it. The ghost participates only in write quorums updating the object's version number until the crashed site recovers. Upon recovery, the available copy remains *comatose* until the next write operation when the ghost directs the latest version of the data value to it.

Like witnesses, this mechanism requires that the write quorum contain at least one stable non-ghost data copy. The obvious advantage of this method is the increased write availability. Enhancements to this method are the use of weak representatives to allow caching of data and the use of witnesses to cut down on storage requirements. Ghosts are similar in concept to temporary witnesses. But the difference is that the former substitutes for a *crashed* copy without retaining even its version number, whereas a temporary witness dynamically replaces a replica with an *obsolete* copy. Ghosts are implemented in the distributed mail servers of the Amoeba system [201] which is discussed in Appendix A.

2.6 Reconfiguration after Network Partitions

The major strength of the quorum consensus class of protocols resides in their ability to mask severe communication failures that lead to network partitions, but at a high price in terms of the number of replicas needed to tolerate a given number of site failure. In order to render these protocols competitive with ROWA ones, it is therefore essential to show that they are able to reconfigure in such a way as to degenerate gracefully under multiple failures. As failures accumulate, we would like for the system to reconfigure so as to retain as much fault-tolerance as is feasible using the remaining resources, and to render operations unavailable only when truly necessary to preserve consistency. This means that quorums must be allowed to shrink as well as expand.

Two major concerns drive reconfiguration under network partitions: first, it should not be possible for more than one partition to update the same data item concurrently. Second, updates performed at a shrunken quorum of sites,

under a particular network partition, must be visible to any potential future partition that can access the item. These two constraints mean that it is generally not possible to shrink both read and write quorum simultaneously.

A common thread that runs among the protocols of this section, is that they attempt to achieve agreement among all the transactions that use a particular configuration.

The first two protocols, dynamic voting and virtual partitions, reconfigure the uniform majority voting protocol. The idea of a *view,* introduced as part of the virtual partitions protocols, has found independent application in other protocols. A view represents an approximation—or hint—of a network partition, which, if correct, permits transactions to run within the view with smaller read quorums. Sections 2.6.3 and 2.6.4 discuss methods to reconfigure weighted majority quorum consensus, and general quorum consensus.

2.6.1 Dynamic Uniform Majority Voting

The dynamic voting algorithm [116, 160] is equivalent to an earlier protocol found in [61] and retains the static vote assignment property of weighted voting. The main difference between this method and plain uniform majority QC is in the definition of the *majority partition.* In the former, a majority partition is the group of sites which possess the majority of the votes, whereas in this method, the majority partition is one which has a majority of the *most current* copies of a data item. This definition helps decrease the size of the minimal set of sites required for a quorum protocol to function.

Each copy d_s of d at some site s has a version *number,* denoted by r_s. A partition is said to be a *majority partition* if it contains a majority of the *current copies* of d. Also associated with each copy d_s at site s is another integer called the *update sites cardinality* denoted by c_s, initialized to the value n. For each $w[d_s]$ operation, c_s is set to a value equal to the number of copies updated during that particular $w[d]$ operation. Each site s must maintain r_s and c_s values for each copy d_s.

As in QC, each $w[d]$ by a transaction T is translated into a set of writes on each copy belonging to some majority partition. When a site s' receives a request to write on copy $d_{s'}$, it performs an operation $w[d_{s'}]$ if it can determine its membership in a majority partition; otherwise, it rejects the update. Site s' determines if it belongs to a majority partition as follows. It obtains the r and

c values from all other sites possessing copies of d in its partition P. It then calculates:

$$r_{max} = \max_{s \in P} r_s,$$

$$S = \{s : r_s = r_{max}, \ s \in P\},$$

$$c_{max} = \max_{s \in S} c_s.$$

If $|S| > N/2$, then s lies in the majority partition. All sites in the set S perform writes on copies of d, tagging them with new r and c values. These new values are given by:

$$r_s = r_{max} + 1,$$

$$c_s = |S|.$$

In order to perform a read, it is necessary that $|S| \geq N/2$. Sites can make their copies current by running a new transaction to determine their membership in a majority partition and by updating their r and c numbers and their local copy from any site in S.

Dynamic voting works correctly with as few as three copies. But there are certain vote assignments, whose occurrence is presumed to be rare, that could lead to inconsistent operations. In addition, the limitations of QC in terms of computing the majority partition and the associated communication overhead persist in this method.

Lower Bound on the Size of the Majority Partition. Dynamic voting schemes attempt to enhance availability, but at the expense of a possible decrease in the size of the majority groups. This is an inherent problem with the method since the most recent majority group is defined relative to the size of the earlier majority group, and not the absolute size of the entire system. This leads to the *tiny majority problem*; a situation where the rest of the sites in the system have recovered and merged, but still are not able to participate in the quorum. To avoid this problem, an additional attribute in the form of a lower bound can be associated with each data item to be used in the computation of majority groups [203]. The use of this attribute by a modified dynamic voting algorithm prevents the size of the majority group from becoming too small.

2.6.2 Virtual Partitions

The *virtual partitions* algorithm [75] operates on two levels, in a manner reminiscent of the missing writes algorithm (see Section 2.3.5). The protocol represents network partitions with data structures called *views*. A view is a set

of sites, uniquely identified by a view identifier, that *agree* that they are all connected and operational. A transaction is allowed to commit only if all of its participants agree that they are members of the same view. The success of an individual operation is determined at two levels. At one level, the protocol decides whether to run read (or write) operations in a new view by checking whether the view contains votes that exceed a global static read (or write) *accessibility threshold.* At the second level, within a particular view, a read (or write) operation completes only if it can collect enough votes to meet a dynamic local read (or write) *execution threshold,* chosen at the time a new view is created. Thus, both levels employ a form of QC.

Accessibility thresholds and execution thresholds both obey the same constraints established in Gifford's weighted voting protocol, and an additional constraint linking the two pairs of thresholds: the write execution threshold must exceed the write accessibility threshold. This extra constraint ensures that at most one partition can write at any one time.

Each site s maintains a *view,* v_s, of potential sites with which it "believes" it can communicate. A transaction T is initiated at site s with view $v(T) = v_s$. T executes as if the network consisted of just the members of $v(T)$. It is imperative that $v(T)$ contain enough votes to meet the accessibility threshold for every operation in T. If not, or if v_s changes before T commits, then T has to be aborted.

Each $w[d]$ is translated to a set of writes to be carried out at a quorum of sites in $v(T)$. Each $r[d]$ is translated to one read on a current copy of d in $v(T)$, determined by polling an appropriate read quorum. Thus the algorithm creates and maintains consistency of data items in abstract network partitions (the views), hence the *virtual partition* name.

The task of maintaining the consistency of views is the responsibility of a special *view update transaction* (an idea similar to the *IN* and *EX* transactions seen in conjunction with directory-oriented available copies in Section 2.2.2). When a site s detects a difference between its view and the set of sites it can actually communicate with, it may choose to change its view to a new one at which time a fresh view-specific pair of read and write execution thresholds must be chosen. This change of view is not a mandatory requirement for correctness; rather, it is a measure to improve the performance of the protocol.

A site s changes its view either by participating in a view change transaction, or by inheriting a newer view from some other site. To generate a new view, the site executes the *view formation protocol* [76] shown below.

1. Site s forms a new view, denoted by v_s' which is the new set of sites with which it believes it can communicate.
2. Site s assigns to the new view a *view id* greater than the current one. The latter would be known only if the members of v_s' can muster votes that meet the read accessibility threshold.
3. For each data item d, site s polls enough sites in v_s to meet the read accessibility threshold criterion, for the most current version number and replica.
4. A copy of the most current version of d, together with the new v_s itself, is then written to as many sites in v_s as is necessary to satisfy the write accessibility threshold.

   This step guarantees that sites that attempt to form a subsequent new view will always contain at least one copy of the most recent view ID, and a copy of the most recent version of every item that is writable in v_s.
5. Site s chooses an execution threshold for operations that are accessible in the new view, so that the write execution threshold is greater than or equal the write accessibility threshold.
6. The initiator s then attempts to commit the view formation transaction. If it succeeds, ht it has installed the new view v_s, and user transactions may execute in it.

All sites successfully accessed by the view formation transaction agree to be members of the same view v_s; the remaining sites in v_s must inherit it in order to make use of the new accessibility thresholds in effect within it.

The timing of the new view formation by a site and the sites to be included in the new view are determined by a set of *tracking policies* [76]. These could include aggressive tracking, where the network topology is monitored with each access, or change of view only when some high-priority data items are read accessible in the new view. The various policies and a discussion of performance is provided in [76].

2.6.3 Dynamic Weighted Majority Voting

In the static vote assignment schemes presented in Sections 2.3 and 2.4, the mutual exclusion protocol will halt if the appropriate site groupings cannot be

formed. It would be better if, in addition to being initialized with an optimal vote assignment, the sites can dynamically reassign votes in the presence of failures. This scheme, which makes the current majority group more resilient to *future* failures, is proposed in [17, 18].

In this protocol, each site maintains an estimate of the votes possessed by every other site. Periodic vote reassignment is performed either autonomously or in a coordinated manner so that the vote assignments of reliable sites are more than those of unreliable sites. Just as in the virtual partition protocol, only a site already belonging to an existing view may initiate a *vote reassignment* protocol if it believes that some failure has occurred. The view, in this case, is rather the set of sites that agree on a vote assignment that ensures mutual consistency, without regard to network topology.

These algorithms achieve mutual exclusion by virtue of performing conflicting operations on intersecting quorums or in the original terminology, *majority groups.* The paradigm proposed in dynamic vote reassignment is for the majority groups to *reassign* their votes intelligently, such that an inconsistent or halted state can be avoided. Sites perform this reassignment and install the changes by initiating a two-phase commit using the current vote assignment. There are two general methods:

- *Group consensus.* Members of a majority group agree upon a new vote assignment using either a distributed algorithm or coordinator election. Sites outside the majority group do not receive any votes.
- *Autonomous reassignment.* Each site autonomously decides to change its votes and picks new vote values. Before the change is made final, though, the site must collect a majority of votes.

In this section, we discuss only autonomous reassignment. The problem of dynamic vote reassignment is split into: (a) *policy,* a criterion for selecting the new vote value, and (b) *protocol,* its subsequent installation in a majority group. We discuss one protocol and one policy designed to achieve vote reassignment below. The protocol works as follows.

1. At each site s_i, define a vote vector V_i, such that $V_i[j]$ denotes site s_j's votes, according to s_i.
2. *Vote collection.* Assume site s_i is collecting votes to decide upon an event. All members of *G,* the group of sites that can communicate with s_i, vote by sending s_i their respective vote vectors.

3. Let $v_i[k]$ denote the number of votes (the weight) cast by site s_k for reconfiguration, as determined by s_i from s_k's response to the vote collection message, using the following rules. (Note that $V_i[k]$ does not necessarily equal $v_i[k]$.)

   (a) If s_i receives V_k from s_k, then $v_i[k] = V_k[k]$.
   Also, set $V_i[k] = \max(V_i[k], V_k[k])$.

   (b) If s_i does *not* receive V_k from s_k within a timeout period, s_i sets:

$$V_i[k] = v_i[k] = \max_{j \in G} V_j[k].$$

4. Now s_i has to determine if it has the majority of votes. For this, it computes the total number of possible votes as $t = \sum_k v_i[k]$, and the number of votes it has collected $v = \sum_{j \in G} v_i[j]$. Site s_i has a majority if $2v > t$.

5. *Vote increase.* Site s_i sends its new voting weight $V_i[i]$ to all the sites in G, waits for a majority of the acknowledgments to arrive, then installs the change in $V_i[i]$. Each *participant* s_j updates $V_j[i]$, and acknowledges receipt of the new weight for s_i.

6. *Vote decrease.* Votes can also decrease giving more flexibility for vote selection. A vote decrease can be used to fall back onto a better vote assignment after a failure is repaired. Details of the vote decrease protocol can be found in [18].

As defined earlier, a policy is a systematic way to pick a new vote value. Vote increasing techniques can be classified under two basic strategies: *alliance,* in which the majority group members uniformly increase their weights, and *overthrow,* where only one site in the active group takes on more votes. The fundamental idea behind these techniques is to enable surviving sites to assume the voting power of failed or disconnected ones.

Dynamic voting can reassign the total votes to tolerate both site and communication failures. This method also enables autonomous reconfiguration of votes in sites, unlike the virtual partition method where a consistent view needs to be coordinated between the sites.

2.6.4 Dynamic Quorum Adjustment

Herlihy's method for dynamic quorum adjustment [111] applies to his general quorum consensus (GQC) replication control protocol for abstract data types

(see Section 2.3.4. GQC distinguishes between *initial* and *final* quorums, which control two separate vote collection phases during the execution of a single operation. This method does not require a global reconfiguration to adjust quorums; that is done dynamically by switching to another quorum set if the current set does not permit the transaction to progress.

Under dynamic quorum adjustment, a sequence of configurations is computed when a majority of sites are available, to be used when failures occur later. A transaction that fails to collect a quorum using a particular configuration level, aborts and restarts at a higher level. Transactions serialize by the level number of the configurations they use. The availability of blind-write operations (and operations with relatively small initial quorums) improve at higher levels because final quorums can shrink. On the other hand, initial quorums cannot shrink with level, since they are required to intersect the final quorums at all lower levels. In addition to favoring blind-write operations over the more numerous read-only ones, a main disadvantage of Herlihy's scheme is that the sequence of configurations must be computed before failures occur, which constrains the choice of configurations to those that perform well on average.

2.7 Weak Consistency

The traditional voting-based algorithms may prove to be overly restrictive for certain applications whose data consistency requirements are relaxed enough to permit data accesses on two different partitions in the event of failures. For example, consider the case of two branches (sites) of a bank, one managing the checking account and another the savings account of a particular customer, partitioned due to failure. The bank's policy is assumed to permit an overdraw from the checking account as long as the funds in the savings account cover the overdraft. But if a partition occurs, no voting scheme would allow the occurrence of a checking withdrawal (which requires reading the balance of both the accounts) in one partition and a savings deposit in the other, even though these operations violate neither the bank policy nor the notion of correctness for the replicated data.

2.7.1 Class Conflict Analysis

Class conflict analysis [195] consists of three distinct steps–transaction classification, conflict analysis, and conflict resolution. The first step involves clas-

sification of transactions, each *class* being a well-defined transaction type such as "checking withdrawal" or a syntactically defined entity, such as "a class of all transactions reading and writing a subset of items a, b, and c." Classes are characterized by their readsets and writesets. A readset (writeset) of a class is the union of the readsets (writesets) of all its member transactions.

Two classes C_i and C_j are said to conflict if one's readset intersects with the other's writeset. A conflict between a pair of classes indicates that a pair of transactions belonging to these classes *may* issue conflicting data operations and, therefore, may have an order dependency existing between them. Each site assigns a *preference* value to each class, that reflects its need to execute that class when a partition occurs. Each site must make its preferences known to all other sites.

Once partitioning is detected, each site runs a protocol to determine the sites within its communication partition. Partition-wide preferences are aggregated from individual preferences of the partition members and used to determine a set of classes from which the sites can execute transactions. Conflict analysis uses a *class conflict graph* (CCG), which provide the *potential* order dependencies between conflicting classes. It is similar in concept but differs from a serializability graph (SG) in that the latter gives all the *actual* order dependencies between conflicting transactions.

A site in a CCG represents the occurrence of a given class in a given partition. Edges are drawn between any two occurrences of conflicting classes C_i and C_j, readset(C_i) $\cap$ writeset(C_j) $\neq \emptyset$, based on two rules: if C_i and C_j are in the same partition, then a pair of edges pointing in opposite directions connect them; if C_i and C_j are in different partitions, then a directed edge extends from C_i to C_j. Cycles existing among class occurrences in the same partition are common and harmless, since the concurrency control algorithm operating in the partition will prevent non-serializable executions.

For example, consider transactions from C_i, C_r, C_w, C_r, C_i, executing in that order in partition 1, in Figure 2.6. There are three logical data objects i (interest rate), s (savings balance), c (checking balance) and four classes C_i (modify current interest rate), C_s (add interest payment to a savings account), C_w (checking withdrawal), and C_r (read current interest rate and current checking balance). However, multipartition cycles, those spanning two or more partitions, indicate the potential for non-serializable executions, since there is no global mechanism to prevent their occurrence. For example, if transactions from C_i, C_r, C_w, C_r, C_i, execute in that order in partition and a transaction from C_s executes in partition 2, then the resulting execution is unserializable.

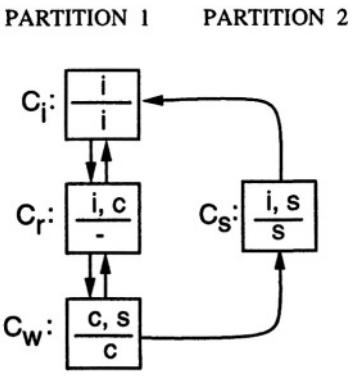


Figure 2.6 Class conflict graph for the banking example. Notation: $\frac{\text{readset}}{\text{writeset}}$.

The conflict resolution step involves finding an appropriate set of constraints to impose on transaction processing so that multipartition cycles can be removed. Typically, classes are deleted in the conflict graph so as to render it multipartition acyclic. But the problem of finding the set of classes with minimal aggregate constraints that render a graph multipartition acyclic is -complete. A less restrictive, but less efficient, alternative is edge deletion.

2.7.2 Read-only Transactions

Similar to the applications amenable to class conflict analysis, read-only (RO) transactions in different partitions can benefit from relaxed serializability constraints or weak consistency requirements [84]. For example, consider a set S of five sites $\{s_1, s_2, s_3, s_4, s_5\}$ in a replicated database that uses the QC protocol, and suppose that S is partitioned into two groups $\{s_1, s_2\}$ and $\{s_3, s_4, s_5\}$. Under the conventional scheme, no transaction—either read-only or read-write—can access the sites in the first partition. Read-only protocols are designed to handle such cases. Typical examples of RO transactions are queries that extract information from databases for auditing and checkpointing. In this section, we briefly enumerate some of the protocols that have been proposed in this area of replication study.

A *weakly consistent* or *insular* transaction is one that must see a consistent database state resulting from a serializable execution of update transactions,

but its execution need not be serializable with respect to *other* RO transactions. The choice of consistency level is determined by the application's usage of the information returned by the transaction.

Weihl, in [213], defines three levels of consistency for RO transactions and provides algorithms for executing RO transactions in multiversion databases. Two protocols use virtual partitioning (discussed in Section 2.6.2) as their basis: (1) a method, proposed in [47], increases the availability of cluster-based replica control protocols, ensuring serializable executions for RO transactions, but forcing read-write transactions to write to all copies in the cluster; and (2) the algorithm in [208] identifies the partitions in which read-only transactions may execute without violating correctness. These methods are necessarily tied to the replication control protocol and identify the conditions when an RO transaction would execute correctly.

Commit Propagation Mechanism. A method that dynamically produces sufficient conditions for RO transactions is discussed in [74]. The correctness requirements for this protocol are external consistency [213]—all read-only and read-write transactions that access the same quorum need to observe 1-SR executions—and each of the read-only transactions must be one-copy serializable with read-write transactions. However, two read-only transactions need not be one-copy serializable with respect to each other. Thus this method guarantees the effects of one-copy serializability when required but also provides flexibility for stale reads.

The central theme of the protocol is a *commit propagation mechanism* (CPM) that piggybacks version lists, on the messages of a two-phase commit protocol. When some transaction T_i accesses a copy at site s_1, all copies residing at s_1 are updated to incorporate all updates executed by previously committed transactions that conflict with T_i. In other words, copies at site s_1 reflect all the changes made to the database by transactions that *may have* affected T_i. A read-only transaction can opt to read the most up-to-date version of the data copy by calling a quorum and executing the CPM, or it may proceed to read the local copy, which may diverge from the most up-to-date copy.

Other methods that use data propagation to increase availability in a replicated system are found in [83, 102, 138, 219, 219] where gossip messages are used to determine partial causal relationships between operations.

2.7.3 Optimism and Conflict Resolution

An optimistic replication method is one that does not limit availability under network partitions in order to maintain consistency. Within each partition, a *partition log* is kept of the transactions that are optimistically allowed to execute, risking possible inconsistency with transactions running in other partitions. When the partitions are merged, these inconsistencies are detected and then resolved. In the worst case, transactions that have been committed under network partition, must be later undone, then redone using a possibly different database state, and yielding possibly different results. Several approaches have been proposed in the literature to detect these worst case situations as accurately as possible, *i* , without over estimating. Of these approaches we mention three: version vectors, partition logs, and precedence graph.

Version Vectors

In the technique given in [165], a *version vector* (s_i, v_i) is associated with each of the n copies of a file, where v_i is the number of updates originating from site s_i. A version vector v is said to *dominate* v', a relationship denoted by $v \geq v'$, if every element of v is greater than the corresponding element in v'; v and v' being two vectors of the same file. A conflict occurs if neither v nor v' dominates, in which case it has to be resolved manually. This is a rudimentary technique with no capability to detect read-write conflicts, hence it is not suited for multifile transactions.

Partition Logs

A partition log is the sequence of transactions executed in a given partition. After a partition is repaired, the reconciliation problem consists of finding a way to merge these logs. Blaustein's method [42] performs *log transformations* based on the semantic properties of predefined transaction pairs. Upon reconnection, these semantic properties such as commutativity $(T_iT_j = T_jT_i)$ and overwriting $(T_iT_j = T_j)$, help to reduce the size of the *merge log*—the combined undo/redo log in each partition. In the original method, each partition has a notion of "absolute time , which helps in merging transactions based on the time at which they were executed. This idea is useful only if *external consistency*[6] must always be guaranteed, but this is not necessarily the case in all database systems, especially under conditions of network partition.

[6] global serialization order is externally consistent if it agrees with the real time transaction submission order.

Precedence Graphs

The optimistic protocol suggested in [63] uses a precedence graph constructed from partition logs, defined above, to model the order of transactions. The nodes of the graph represent a set of transactions $T_{i1}, T_{i2}, \ldots, T_{in}$ arranged in serialization order in a given partition *i*. The edges of the graph are of three types: *data dependency* edges, *precedence edges,* and *interference edges.* Data dependency edges are expressed as writeset$(T_{ij}) \cap$ readset$(T_{ik}) \neq \emptyset, j < k$, implying that T_{ik} reads a value produced by T_{ij} and that T_{ik}'s existence depends on that of T_{ij}. *Precedence* edges are expressed as readset$(T_{ij}) \cap$ writeset$(T_{ik}) \neq \emptyset, j < k$, implying that T_{ij} read a value which was later changed by T_{ik}. *Interference* edges are expressed as readset$(T_{ij}) \cap$ writeset$(T_{kl}) \neq \emptyset, i \neq k$, implying a read-write conflict and that T_{ij} logically executed in a different partition before T_{kl}.

The state of the database is consistent if the precedence graph for a set of partitions is acyclic. Inconsistent transactions have to be resolved by rolling back the transactions until the graph is acyclic. An $O(n^{2.81})$ time solution to the -complete problem of minimizing the associated cost function is provided in [62].

2.8 Coding-theoretic Redundancy

Much of the resistance to the transfer of replication technology, from the realm of research to the domain of applications, arises from the exorbitant storage cost of replication, both interms of size, and in terms of I/O bandwidth. Anexample of a direct attempt to address this problem is a method by Agrawal and El Abbadi [5], that reduces this overhead by fragmenting the file and replicating the fragments independently. But when we observe that error-correcting codes have long been profitably and unobtrusively implemented in products ranging from main memory sus-bsystems to communication and storage devices, we must ask ourselves why coding theory has not been exploited to dramatically reduce the overheads of replication. The Information Dispersal Algorithm (IDA) by Rabin [177] has been influential in this regard, so we describe it below.

Given a file that is represented as an array $F_{m \times \frac{N}{m}}$ of *N* bytes, IDA transforms it into another array $S_{n \times \frac{N}{m}}$, $n \geq m$, each row of which is stored on a different site. Given any m of the n rows of $\hat{}$, the algorithm recovers the original file *F* intact. This means that IDA can tolerate the failure or inaccessibility of $(n-m)$

sites without losing the ability to reconstruct the file. Let $S(\sigma)$ represent the available m rows of , where σ is the index set that identifies these rows. Choose a prime p greater than the largest possible value for a single byte of *F*. IDA can be summarized in the following encoding and decoding steps, to be performed over the finite field Z_p, *i.e.,* all arithmetic is performed modulo p:

$$S_{n\times\frac{N}{m}} \leftarrow A_{n\times m} \times F_{m\times\frac{N}{m}} \pmod p$$
$$F_{m\times\frac{N}{m}} \leftarrow A(\sigma)^{-1}_{m\times m} \times S(\sigma)_{m\times\frac{N}{m}} \pmod p$$

where $A_{n\times m}$ is a randomly chosen array of rank m, *i.e.,* every square subbarray $A'_{m\times m}$ is non-singular, or, equivalently, any m rows are linearly independent. Rabin [177] gives a randomized procedure for generating *A* that, with high probability, will have the correct rank.

In its raw form, IDA has good read availability, since an operation can proceed with any m of the n segments, and both parameters are controllable. Furthermore, IDA requires the transmission of only as many bytes of *S* as there are in *F,* in order to perform a read operation. More importantly, IDA has a *security property* that makes it unique among replication schemes: if an adversary can obtain $(m-1)$ segments from , they cannot deduce even one byte of *F*. On the negative side, IDA exhibits poor update availability, because a write operation, even if it involves modifying a small number of bytes from *F,* requires that all n replicas be available. Agrawal and Jolte [7] boost the low availability inherent in IDA by replicating segments and storing more than one segment at a site. Other work on the security of replicated data includes [4, 112].

This page intentionally left blank

3

Replication of Processes

Replication of processing modules as a mechanism to achieve fault tolerance has been mentioned in the literature even earlier than data replication [211]. Since then, work in this area presents two fundamental approaches. The first is *modular redundancy* in which each component performs the same function[10, 133, 141,186]. The replicas here are called *active replicas.* The second approach involves *primary/standby* schemes where a primary component functions normally, periodically checkpointing its state to the secondary replicas, which remain on stand-by and take over in case the primary fails [21, 45]. Such replicas are called *passive replicas.*

3.1 Replication based on Modular Redundancy

The simplest mechanism for remote, and hence distributed, execution of programs is the *remote procedure call* [41]. RPCs are capable of providing distribution transparency only as long as all the systems are running correctly. Failure of some of the sites causes a distributed program to crash at those sites, resulting in *partial* failure of the system, thereby violating transparency. Replication offers a method to mask these partial failures.

As compared to providing fault tolerance in hardware or embedding fault tolerance code in application software, modular-redundancy replication provides a simple and cost-effective alternative. Hardware fault-tolerance is expensive–it advocates the use of ultra-reliable or redundant hardware for all components, and not just for the critical ones. Application-level fault-tolerance, on the other

hand, could be too complicated, rigid, and brittle. It requires fault-tolerance experts to write the applications (thus wasting the experts time, or the company's money!) Modular redundancy replication, by contrast, can be applied systematically to any application. It also can be applied selectively to critical components.

Compared to the passive replica approach, the active replica approach offers a few important advantages: (1) it eliminates complicated checkpointing procedures during normal operation; (2) a site failure need not induce any explicit failure recovery protocol; (3) better performance can be achieved by using a local replica, if available; and (4) leads to better utilization and load-balancing of system resources. However, these advantages are realizable only if user transparency to the underlying process replication is achieved, which brings us to the question of maintaining the consistency of the replicated processes.

3.2 Consistency of Processes

The simplest form of consistency between processes is *no consistency* at all. An example is the Sun NFS file server [181], which is a *stateless* server. Replicated instances of the same server are allowed to run concurrently in order to improve performance. None of the servers maintain critical state information about ongoing remote file system interactions. This way, in the case of a failure, a complex crash recovery procedure is avoided and the client just resends the request until a new response is received.

Another example of *no consistency* process replication is dynamic server squads in Yackos [108]. A *squad* is a homogeneous collection of processes, that cooperate to provide a particular service. The principal motivation for Yackos is fast response to clients, not availability or reliability. When a member of a squad receives a request, it either services it directly or forwards the request to another member that will be able to do so. In case of a load increase, the member spawns an apprentice member to which it may assign some of the work. On the other hand, if it does not have enough work to justify its existence, it distributes its responsibilities among the other servers and deletes itself. These phenomena are called the *growth* and *shrinking* of the squad. An ordinary server process can thus be converted to a squad, and the process is called *breeding*. The design of Yackos is of particular relevance to replication since replication via squads is an attractive opportunity to achieve reliability and availability in tightly coupled systems.

When replicated processes are used as a means to achieve reliable program execution, some mechanism is required to maintain mutual consistency of the processes in order to achieve replication transparency. The problem of consistency maintenance in processes and the cause for inconsistencies are clearly stated in [153]. An *inconsistent execution* is a phenomenon where different replicated processes behave differently. Inconsistent executions lead to two main problems: inconsistent process states and inconsistent external behavior, and hence, lack of replication transparency.

Process execution inconsistencies can arise due to non-deterministic program behavior. For example, there is a potential inconsistency if different operands are input to different replicas. The consistency requirements proposed in the literature are based on two main models of replicated processes. In the Circus approach discussed in the next section, processes are mutually consistent if all the processes are *deterministic* and if all of them are in the same state. Consistency is achieved using the transaction concept. In the Clouds approach, the problem of non-determinism is avoided by committing only one of the replicated processes and *copying* its state to other processes in order to maintain consistency.

3.3 Replicated Distributed Programs and the Circus Approach

The concept of replicated processes and replicated distributed programs is introduced in [58] and is implemented in the *Circus* replicated system [57]. Programs are modeled as a collection of distributed modules. Similar to an ADT, the *module* packages the procedures and state information needed to implement a particular abstraction and separates the interface to that abstraction from its implementation. But there can be only one logical instance of a module (with a single state variable), whereas there can be many instances of an ADT (each with its own state variable).

The basic unit of execution in the modules is a *thread.* A program begins as a single thread of control, and additional threads may be created and destroyed using operation primitives. A thread can execute in exactly one module at any given time, but several threads can concurrently run in the same module. A thread moves from any given module to the next by making RPC calls to, and returning from, procedures in the latter module, with their control flow following a LIFO (stack) discipline. A module is said to be *deterministic* if

the state of the program at the time that an event (such as a call to one of the procedures in a module) occurs, and a time interval, say *I*, during which a sequence of events have taken place, uniquely determine the event that follows *I* and the state of the module when that event occurs.

A replicated module is termed a *troupe,* and each module is called a *troupe member.* A *deterministic troupe* is a set of replicas of a deterministic module. A troupe is *consistent* if all its members are in the same state. Similar to data consistency in data replication, *troupe consistency* is a necessary and sufficient condition for replication transparency. Clients of a consistent troupe of servers need not be aware of the fact that the server module is replicated.

The second tool that facilitates process replication in Circus is the *replicated procedure call,* which is a generalization of remote procedure call for many-to-many communication between troupes. Remote procedure calls help in the implementation of distributed *threads of control* between the modules and provide location transparency. In the event of partial failures of the distributed program due to processor and network failures, the replicated procedure call must have the capability to mask these failures. These calls are of the types: *one-to-many, many-to-one,* and *many-to-many* with each type managed by corresponding message algorithms.

Reliable programs can be constructed using troupes. An inter-module procedure call results in a replicated procedure call from a client troupe to a server troupe (shown in Figure 3.1). The troupe members do *not* communicate among themselves and are not even aware of each other's existence. Each server troupe member executes the procedure, just as if the server had no replicas. The execution of a procedure can be viewed as a tree of procedure invocations with invocation trees rooted at each member of the server troupe. The global determinism ensures that the invocation trees are identical, and each member makes the same procedure calls and returns with the same arguments and results, thus maintaining the same execution order.

If there is only a single thread of control in a globally deterministic replicated distributed program, and if all troupes are consistent to begin with, then all troupes remain consistent. In the case of multiple threads of execution, concurrent calls to the same server troupe may leave its members in inconsistent states; hence, the transaction concept is needed. Concurrent calls, even in the absence of replication, have to be serialized. In the case of a troupe, each troupe member serializes the concurrent calls (transactions, in this context) in the same order as other members in the troupe in order to achieve the dual objectives of serializability and troupe consistency. This calls for a protocol

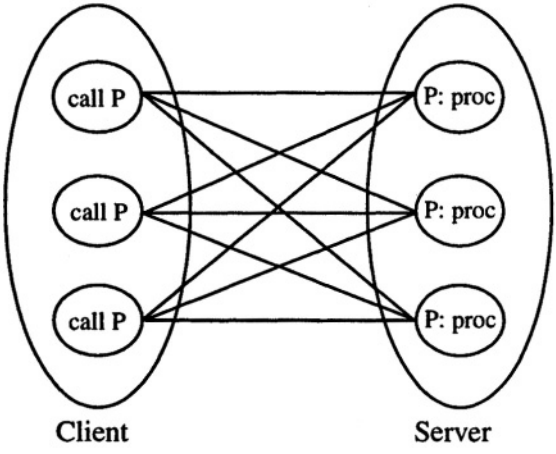


Figure 3.1 Replicated procedure call of type *many-to-many* in Circus.

that helps coordination between the replicated procedure call mechanism and the concurrency control mechanism within each troupe member. Moreover, this protocol should be devoid of any inter-member communication within a troupe.

An optimistic *troupe-commit protocol* is proposed for this purpose. This protocol is *optimistic* since it assumes no conflicts between concurrent transactions. It is also *generic* since it makes no assumptions about the local concurrency controller within each member. If members of a troupe attempt to serialize transactions differently, this protocol detects and transforms such an occurrence into a deadlock. Deadlock detection is then used to abort and retry offending transactions. The semantics of the replicated procedure call are derived from the above troupe consistency requirements. Each member of the server troupe executes the requested procedure *exactly once,* and each member of the client troupe receives all the results.

A replicated distributed program constructed in this manner will continue to function as long as at least one member of each troupe survives. Network connectivity is an essential requirement; the replicated calls cannot differentiate between the site and communication failures and rely on the underlying messaging protocols to detect crashes using surveillance techniques. A network partition can cause troupe members to diverge with each member continuing to execute with the belief that others have crashed. Since there is no inter-member communication, one solution to this problem is to require each troupe member to receive a majority of the expected set of messages before commencing the execution.

3.4 Replicated Transactions and the Clouds Approach

The approach adopted in Circus requires the operation to be deterministic in order to maintain the members in a consistent state. The Clouds [60] approach avoids non-determinism by committing *only one replica* of a replicated computation. The state of a member at the site where a replicated procedure is committed is copied to other members (similar to the *primary copy* mechanism in data replication; *cf.* Section 2.2.3). Each member of a client troupe invokes only one member of a server troupe. Duplicate invocations are avoided.

An improvement is proposed in [153] in the form of a replicated transaction scheme. The system is modeled as a collection of objects; each object replicated at several sites. Each object replica is composed of the following: *shared state*–which refers to the state of the object that captures the modifications made by successfully committed transactions; *code*–read only object code; *buffered updates and locks*–for transaction management within each object, to keep track of the updates made and locks acquired, so that this can be installed in the shared component upon commit time.

A transaction is executed as several concurrent transaction replicas after being invoked by a *root object.* A user-specified *resiliency level* is used to determine the number of site/communication failures that can be tolerated. A resiliency level R implies that at least R transaction managers, and hence, sites must be participants in the protocol. A selection algorithm is used to determine the sites that contain the participants. When a transaction replica invokes a replicated object, the operation is executed on only one of the object replicas. Access to the objects by multiple transactions is coordinated so that no two transactions may access the same object replica. When a transaction manager is notified of its being selected, it can immediately commence processing the transaction. The fastest transaction replica gets committed, given that there are no failures.

Upon the return of the invocation to the root object, the transaction manager attempts to commit the transaction replica. The associated commit protocol ensures that only one replica is committed. This method is designed to handle both site and communication failures. But unlike troupes, inter-replica communication is necessary.

3.5 Replication in Isis

Isis [38, 39, 120] is a good example of a collection of tools and mechanisms that support the active replication approach for providing reliability in distributed systems. The computing paradigm used in Isis is essentially the provision of a communication facility for a distributed computer system supporting *fault-tolerant process groups* in the form of a family of reliable multicast protocols. These protocols are designed for use in both local and wide-area networks [39]. Besides providing a high degree of concurrency, these protocols strictly follow application-selectable delivery ordering constraints.

A distributed computation is represented as a set of events operating on a process state and a partial order (which corresponds to the thread of control) on those events. The protocols ensure that members of a fault-tolerant process group observe consistent orderings of events affecting the group as a whole. Events may be local computations, process failures, recoveries, and broadcasts subject to ordering constraints. This communication facility has many applications, of which data replication is an important one.

Three major broadcast primitives are defined:

- *Atomic broadcast* (abcast). Named somewhat misleadingly, the *abcast* is not only an atomic group broadcast (or multicast), but it is also guaranteed to be delivered at all members of the group in exactly the same order. This ensures that all members of the group observe identical sequences of messages, assuming they are all *abcasts.*
- *Causal broadcast* (cbcast). It is often not sufficient that broadcasts be received in the same order at overlapping destinations for some applications; it is also necessary that this order be the same as some predetermined one. As an example, consider a computation in which different group members repeatedly update a shared variable, then broadcast its new value. Here, it is not enough for operations to be carried out in the same order at all copies; they have to be observed in the order that reflects the dependency between successive values, by which earlier values determine later ones. The *cbcast* primitive ensures this level of synchronization, but only to an extent controlled by the application, as elaborated below.
- *Group broadcast* (gbcast). Broadcasts process group *views*[1] to operational group members whenever some change to a global property of the group occurs. The *gbcast* is designed to inform group members of process failures and recoveries, as well as voluntary joins and withdrawals. Upon receipt of this message, recipients update their local copies of the group membership. This primitive enables group members to act on group view information directly without needing any further agreement protocols.

We shall dwell further on the *cbcast* primitive since it is of particular use in replication control. We have seen that the ordering of updates to a replicated data item needs to be the same at all the sites of a replicated database system. In an asynchronous distributed system where sites communicate with each other using messages, the notion of *potential causality* [134], prevents messages from

[1]See Section 2.6.2 for a discussion of views in the context of network partitions.

crossing each other in transit and maintains the causal ordering among them. To be specific, a broadcast B is said to be *potentially causally related* to a broadcast B' if:

1. They were sent by the same process and B occurred before B' according to that process' program ordering, or,
2. $B$was deliveredat *sender*(B') before B' wassent.

This definition lacks transitivity, which must hold if this relation is to model causality. The straightforward solution of taking the transitive closure results in a relation that is too inclusive, in that it can very quickly grow to relate most of the broadcasts being exchanged in the system. For many applications, only a small subset of broadcasts may be truly causally related. An application-controlled mechanism is therefore supplied in Isis, to reduce the explosion in the size of the potential causality relation.

IntheIsiscausal broadcastprimitive,denoted as *cbcast*(*msg, clabel, dests*), the *clabel* argument is a label drawn from a partially ordered domain, with the goal of restricting potential causality to *cbcasts* whose *clabels* are related. This can be seen from the following definition. Given two broadcasts B and B', the notation $B \xrightarrow{c} B'$ represents the condition that $clabel_B < clabel_{B'}$. We define the relation *precedes,* over the set of *cbcasts,* as the transitive closure of:

B precedes B′ if $B \xrightarrow{c} B'$ and B potentially causes B'.

Thus, *clabels* are implicitly endowed with the ability to specify if a potential causality relationship between two broadcasts is significant. The ordering properties of *cbcasts* can now be summarized: they are atomic, and if *B precedes B′,* then B is delivered before B' at any overlapping destination site.

In an environment where causal broadcast ordering properties are guaranteed, updates of replicated data can be consistently applied simply by the use of *cbcasts* to propagate updates among group members. Because *cbcast* updates are delivered by Isis in the same order, consistent with their position in the causal chain of updates, an update can be considered complete by its issuer when it is carried out locally. The Isis broadcast primitives present a simple interface that can be used in both local and wide area networks.

While process groups facilitate active process replication, the *implementation* of the former in Isis can be classified as a primary/standby because of the

following operational feature. Isis designates a *coordinator* in each process group to handle the resilient objects and designates the rest of the group, called *cohorts,* as its passive backup processes. Upon failure of the coordinator, the cohort takes over and restarts the request. Cohorts can pick a new coordinator consistently if a *gbcast* is used. The ordering properties of *gbcast* prevent any inconsistencies during failure and recovery.

3.6 Primary/Standby Schemes

As discussed earlier in this section, primary/ standby schemes involve passive replicas. Only one primary replica performs the computation while the rest of the replicas monitor the state of the primary replica through some failure detection algorithm. The primary has to periodically checkpoint its status to the secondary replicas so that one of the secondaries can take over if and when the primary fails.

The Tandem Non-Stop™ Kernel

The Tandem Non-Stop System [20] is an early and most popular commercial implementation of the primary/standby mechanism. The Tandem system is a multi-computer system with redundant hardware components. Each system can have anywhere from 2 to 16 processing modules, each containing its own memory, power supplies and input-output channels. Furthermore, up to 14 such systems can be interconnected by fiber-optic links. Most components like processors, disks, *etc.,* work in a mirror mode. The software fault-tolerance feature in Tandem is the provision of *process pairs*—a *primary* process running the application always has a *backup* process mirroring its activities.

3.7 Process Replication for Performance

In addition to fault-tolerance, process replication techniques could be used to improve system performance. This approach of using redundancy to improve performance is particularly beneficial in systems with abundant resources. While abundant resources are usually not to be expected in conventional systems, they are more common in real-time environments, which are engineered to cope with rare high-load conditions, rather than normal average-load conditions. In [29], Bestavros and Braoudakis introduced and applied this con-

cept to Real-Time Database Systems (RTDBS). RTDBS are engineered not to guarantee a particular throughput, but to ensure that in the rare event of a highly-loaded system, transactions (critical ones in particular) complete before their set deadlines. This often leads to a computing environment with far more resources than what would be necessary to sustain average loads.

This page intentionally left blank

4

Replication of Objects

Object-oriented database management systems (OODBMS) are discussed and well documented in the literature [28, 50, 79, 98, 114]. Research in centralized object-oriented databases has recently focussed on representing complex information in a database using a *composite* object model where a group of related objects are accessed and manipulated as a single entity. Such composite objects could very well model multi-media documents. In distributed object-oriented systems, components of the composite object may be distributed across multiple machines. Replicating a composite object improves the availability and provides efficient reads but could be expensive in terms of update and storage. Hence, replication should be *selective.* Selective replication in distributed relational database systems can be achieved by fragmenting the relations and replicating the horizontal and/or vertical fragments [46, 82, 152]. Object-oriented database systems provide selective replication using the reference attributes of the object. These reference attributes act as pointers to object replicas [189] or to procedures [117]. Selective replication of a distributed composite object is addressed in [31]. In the following sections, we present some of the issues encountered in distributed composite object replication and give some details of the replication algorithms in the GUIDE object-oriented distributed system.

4.1 Replication of Composite Objects

Distributed composite object replication requires *selection strategies* to choose the sub–objects to replicate, and *replication schemes* to be used in the management of selected replicas. An example of a composite object is given in [31]. In this example, a document is replicated in two sites (see Figure 4.1). The

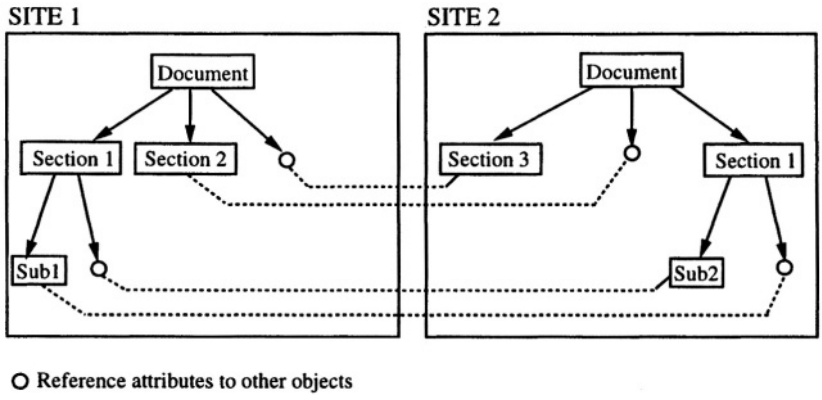


Figure 4.1 Selective replication of a document composite object.

replication of the document object involves replication of the *Section_1* object. Sub–objects of *Section_1* are selected to be partitioned rather than replicated.

Four selective replication strategies are suggested in [31]: replication of objects at particular levels in the object's hierarchical structure, replication of objects and sub–objects based on access frequency, replication of object capabilities, and dynamic reconfiguration. These strategies provide the flexibility of tuning replication granularity to increase the efficiency of managing the composite object. Replication based on access frequency, for example, provides a design that avoids unnecessaryreplication of sub–objects. Furthermore, since the access patterns for sub–objects may change over time, a reconfiguration strategy will be important. One constraint that needs to be incorporated is that the object IDs (OID) should be mutually consistent among the various replicas. A reference field of the composite object replicas should refer to a logical object rather than a physical object.

The same work [31] introduced three distributed composite object replication schemes.

- *Class-specific.* A single replication scheme is specified for all objects of the same class. Objects of a class are stored together and hence get replicated at the same sites. Class-specific replication can be used when a

class is registered with the database system. This scheme provides the least flexibility, but it also has nominal cost in terms of replication management.

- *Instance-specific.* The replication information is specified separately for each object (instance). Instance-specific replication is done when the instances are created. This scheme provides maximum flexibility to the user but has higher cost in terms of replication management.
- *Instance-set-specific.* This scheme allows specifying different replication for different collections of objects belonging to the same class.

4.2 Replicated Objects in Guide

The GUIDE system [15] employs the object model to support cooperative applications distributed on a network of possibly heterogeneous machines. Object sharing is the key feature of the model as persistent objects are the principal way of communication between concurrent activities. Thus it is imperative for the system to maintain object availability and consistency despite node failures or network partitions. GOOFY [53] is a replicated object server intended to provide a fault tolerant storage system for objects used by the GUIDE system. The implementation of GOOFY is based on three main mechanisms: a pessimistic algorithm to ensure coherency of objects, client caching, and lock invalidation to increase object availability.

The GUIDE prototype

The initial GUIDE prototype, developed on top of the Mach 2.5 micro–kernel, consists of one component for distributed execution management and another component for managing GUIDE objects. The latter component, also called the GUIDE Memory Server (GMS), provides two basic services. The first deals with the persistent storage management to create, delete, load, and store objects in persistent storage, the second deals with execution memory management to bind and unbind objects in the virtual memory of a job and to serve paging requests sent by Mach kernels. The GMS is a special task executed on every node. Thus when an activity calls a method on an object which is not yet bound in its job address space, a request is sent to the GMS in order to map the object. Once the object is mapped in the region of the job address space, the Mach kernel is in charge of requesting object pages to the GMS on each page-fault. The limitation of this design is its lack of support for fault tolerance, particularly node failures and network partitions. In order to achieve

reliability in the system, objects are replicated at the level of secondary storage. This approach increases object availability under certain conditions as it allows access to objects even if a node failure or a network partition occurs.

The GOOFY server was developed to manage these replicated objects, and to provide the following two services: (1) storing objects in Reliable Persistent Storage (i.e., in the persistent storage managed by the GOOFY server), (2) loading objects in the GMS space (i.e., the execution memory). The server is replicated at each node and its clients are the GMS of each node. The servers cooperate to make the replication of all objects transparent to the GMS. The following section describes how the goals of providing higher availability while ensuring object consistency are achieved.

The GOOFY Replicated Object Server

The consistency requirement to guarantee that all jobs share the same coherent image of the object is met by allowing only one GMS to obtain a copy of the shared object. GOOFY uses a locking mechanism that refuses to serve another load operation of an object so long as a GMS already manages this object. Further, a GMS managing an object is responsible for providing coherent copies to the jobs it serves. This ensures consistency since there exists only one updatable image of an object at any given time in the memory managed by the GMS. Also, if there is a GMS failure due to node failure or network partition, it is considered permanent by GOOFY, and all the locks owned by the GMS are invalidated to allow other GMS to take over by using other copies of the object still in reliable persistent storage. If no other GMS tries to load objects locked by the crashed GMS, the locks it owns will remain valid until it is repaired. The *quorum consensus* algorithm is applied as a pessimistic method that resists network partitions during the collection of votes. Similarly, replica failures or message loss is detected by using timestamps for each lock or message. During each stage of the locking protocol, the actions of the replicas are guided by the value of the timestamps. When a replica detects that a timestamp is out of date, it tries to update the object.

In order that a job be able to locate the GMS which manages an object (since an object could be loaded in the execution memory by different GMS during its lifetime), every object is assigned an identifier which can be used to locate the object, either in the GMS or in the reliable persistent storage.

The management of the replica group, which constitutes the GOOFY server, must take care of two classes of events: the insertion and removal of the replica

and the failure and restart of the replica. The first class of events occurs only if the composition of the group is allowed to change dynamically which, in turn, affects the quorum consensus algorithm since the size of the group is used during the voting stage. The first version of GOOFY considers only static groups, thus avoiding the problem. Failure detection is performed by checking replicas periodically when there are few communication exchanges between them. When a replica failure is detected and repaired, the out–of–date copies managed by this replica are updated by copying the entire state into the repaired node.

This page intentionally left blank

5

Replication of Messages

The management of data or process replication usually requires similar messages to be exchanged to all or some of the replicas. This gives rise to message replication. The simplest form of message replication is to generate as many message replicas as required by the implementation of the data (process) replication protocols. In this case, the replication of messages is a by-product, and no special message coordination schemes are needed.

Two other forms of message replication have been introduced in the literature. They do more than obey a higher level replication protocol. The first is there liable multicast group of message replication [39, 51,120]. In this form, the low-level message subsystem (like the process group in the V system [51]) or a higher level abstraction (like the group broadcast primitives in Isis [39]) provides for fault-tolerance and message delivery guarantees in the presence of communication or replica failures. Higher level data (process) replication protocols that use this form of message replication are much simpler in design and implementation. This is because they replace complicated failure scenarios that they must cope with, with a clean set of failure semantics. Such semantics include atomic broadcast, ordered broadcast, and group semantics. In essence, this form of message replication contributes to the simplification of the management of replicated data (processes).

The second form of message replication introduced in [92] shifts even more responsibilities of replicated data management from the higher level protocols and into the message subsystem. The quorum multicast suite of protocols introduced in [92] pushes the most part of replication management into an *above-kernel,* high-level communication subsystem. In these protocols, the one-copy semantics of data is maintained by the exchange of a necessary, yet "mini-

mum," number of messages. The idea is very interesting and perhaps should be researched further. An extension of this work could be to re-package the high-level communication subsystem into a kernel-level implementation, similar to the V system approach. Quorum multicast is a nice idea that happened to choose a particular replication technique to use at the message level. Other replication techniques should also be investigated. The success of this form of message replication can lead to a substantial reduction in the complexity of data (process) replication design and implementation.

In the following sections, we give a brief presentation of the reliable multicast and the quorum multicast approaches.

5.1 Reliable Broadcast Protocols

Reliable multicast or broadcast protocols [39, 51] for reliable communication support and simplify replication in fault-tolerant distributed systems. These protocols avoid several rounds of message exchange required by current protocols for consensus agreement, and only a single broadcast message is typically required for distributed operations such as locking, update, and commitment. The implementation of the Isis reliable multicast protocol [39] allows members of a group of processes to be able to monitor one another so that they can take action based on failures, recoveries, or changes in the status of group members.

See Section 3.5 for a detailed description of Isis' most important multicast primitives, and a discussion of their use in supporting data and process replication.

5.2 Quorum Multicast Protocols

Quorum-oriented multicast protocols [92] allow delivery to a *subset* of the destinations, selected according to availability, distance or expected data currency. These protocols provide well-defined failure semantics and can distinguish between *communication failure* and *replica failure* with high probability. Data replication can provide good performance when quorum multicasts are used. Although the ideal protocol would provide the highest availability with the lowest traffic and latency, trade-offs between these measures must be made. Broadcast messages on a LAN allow replication protocols to send requests to all replicas in one message while a separate message must generally be sent to

each replica in an internetwork. The quorum multicast protocols can dynamically select this subset to be the closest available destinations, limiting the portion of the internetwork affected by any particular multicast. These protocols are useful when implementing replication methods based on voting or when communicating with one of a group of replicas in a fault-tolerant manner.

Quorum multicast protocols first use the closest available replicas, fall back on more distant replicas when nearby ones are unavailable, and are parameterized so as to provide the replication protocol with hints to improve the performance of subsequent communication. Replication protocols can be implemented using quorum multicast, limiting the cost of message traffic and using nearby replicas for low latency. Quorum multicast protocols also address the problems associated with transient failures by resending messages to replicas.

Four variants of quorum-oriented multicast protocols are introduced in [92]. The choice of one of the protocols depends on the relative importance of the probability of success, operation latency, and message count. The details of these variant protocols can be found in [92].

This page intentionally left blank

6

Replication in Heterogeneous, Mobile, and Large-Scale Systems

This chapter considers three impediments to the implementation of replication schemes: heterogeneity, mobility, and scale. We review a number of solutions that are available in the literature, but we caution the reader that research on these issues is both recent and sparse, despite the criticality of the problems.

Replication in *heterogeneous environments* is difficult due to site autonomy. First, it is hard to enforce atomic actions in these environments [77], even in the absence of replication. In addition, global consistency [89, 69, 151, 78, 85], which is different from local consistency, needs to be enforced. Several solutions have been proposed including those in [22, 48, 70, 77]. The problem of enforcing global consistency in a heterogeneous system is further complicated when data is replicated. In Section 6.1, we describe some recent solutions to replication management in heterogeneous environments.

Mobility is another challenge that necessarily involves replication, yet complicates it. Ubiquitous access to data and information is becoming a highly desired service. On-demand replication and caching and cache replication are new approaches that support weakly connected operations by mobile users. Section 6.2 discusses the main issues of replication in the mobile environment and presents available solutions.

The third impediment that we cover in this chapter is that of *scale*. Classical replication implicitly or explicitly assumes a "small" number of replicas and a "small" size system. In large-scale systems, classical replication does not work adequately due to performance degradation and to assuming an inaccurate failure model. The degradation of performance is due to the insistence on immediate consistency enforcement over a large number of replicas (perhaps

tens of thousands) that must be accessed through a wide area network. The failure model in these large systems is also different from that of a smaller one in that failures are almost always happening. This mode of failure is called the *continuous partial operation* in [168]. Section 6.3, is devoted to the impact of scale, and to methods of achieving scalable replication.

6.1 Replication in Heterogeneous Databases

Fault-tolerance and reliability in heterogeneous databases (HDBS) is a relatively open area. Methods to achieve reliability through recovery protocols have been discussed in the literature, but they do not address availability. The problems of exploiting redundancy of data and achieving resiliency against site and communication failures have only recently been addressed.

In contrast to explicit replication in a DDBS, with the purpose of achieving fault-tolerance, data redundancy in a HDBS may possibly be a pre-existing fact. Much of the data existing across the various platforms in an enterprise may be redundant—either by replication or by some logical relationship. To make use of this redundant information as a global resource in a cost-effective manner, interdependencies between data objects need to be identified, specified, and then manipulated from a level that is global to all the disparate systems. Further, it is desirable to tap this redundancy as a means to increase the overall availability of the enterprise-wide data management system.

In the following sections, we describe some of the recent work done in this area. Some of the work deals mainly with the management of interdependent data without specific emphasis on fault-tolerance. Other work emphasizes fault-tolerance without giving sufficient explanation as to how and why identical replicas are implanted in a collection of autonomous, heterogeneous systems. Regardless of whether interdependency management or fault-tolerance is the emphasis, site autonomy proves to be a major obstacle that disables the applicability of data replication techniques in the heterogeneous environment. Getting around the restrictions of site autonomy, especially to enforce agreement on transaction commitment, represents the largest emphasis of the work presented in the following sections.

6.1.1 Identity Connection

Wiederhold and Qian in [216] introduced the concept of *identity connection* which relates attributes belonging to different relations or fragments allocated at different sites. An identity connection between different data fragments means that they are identical replicas. The concept applies to data fragments whose interdependency lies within the same database or across different databases. Based on the identity connections, consistency constraints may be specified to determine the permissible lag of currency between two *connected* data items. The notion of primary/secondary copy is used to specify the consistency constraints. An update to the primary component is accompanied by posting transactions at sites of the secondary components. Consistency enforcement follows a temporal constraint that is specified as part of the identity connection.

6.1.2 Update through Current Copy

A contribution towards extending the identity connection idea, with a major emphasis on its practical realization, is found in [180, 191]. The term *interdependent data* (**id**-data) is used to characterize data items that are stored in different databases but related to each other through an integrity constraint that specifies the data dependency and consistency requirements between these data items. Examples of these interdependencies and constraints are shown in Figure 6.1 (taken from [191]). Unlike an identity connection that links identical fragments, this approach captures the interdependency of non-identical **id**-data. For example, an **id**-data item can be derived or fragmented from another data item. An **id**-data can also be an aggregation of other data items. Regardless of the specific interdependency relation, **id**-data are considered *replicas in part,* requiring some form of mutual consistency enforcement.

In [191], consistency of the **id**-data is maintained on the basis of a criterion other than 1-SR called *current-copy serializability.* The mutual consistency protocol based on this correctness criterion ensures that there exists at least one current data component in the system. The rest of the data components are made to converge to the value of this most current copy based on the specification of an eventual consistency requirement. If the requirement is that of immediate consistency (like 1-SR), and if site autonomy would allow, then consistency is enforced by immediate updates at each site. Otherwise, eventual consistency is brought about according to the specified consistency requirements. The method offers limited failure-resiliency since it is based on a primary-copy technique.

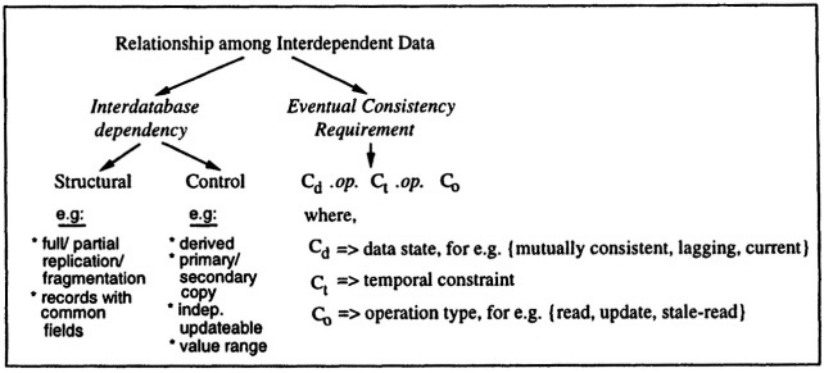


Figure 6.1 Classification of constraints on interdependent data.

The failure of the site maintaining the interdependency information can halt the entire system while a communication failure will only delay updates, which may be acceptable according to the correctness criterion.

Current-copy serializability is enforced *asynchronously* by propagating updates from the current copy to the other replicas. Such asynchrony is suitable to autonomous systems, where updates can be retried until they are accepted. Epsilon-serializability [172, 173] is a generalization of current-copy serializability, with finer control over inconsistency tolerance and restoration.

6.1.3 Interdependent Data Specification and Polytransactions

The above work on *current copy* is generalized further in [180] where a scheme is proposed for the specification and management of interdependent data. The interdatabase dependency criterion is extended to D^3—a *data dependency descriptor,* which is a five-tuple $< S, U, P, C, A >$. S is a set of *source objects* and U, the target object. P is the interdatabase dependency predicate that defines a relationship between source and target data objects. P evaluates to true if this relationship is satisfied. C is the mutual consistency predicate that specifies the consistency requirements and defines when P must be satisfied, and A is a set of consistency restoration procedures—actions that must be taken to restore consistency and to ensure that P evaluates to true.

We use an example to illustrate the D^3 criterion. The example describes a replicated database consisting of three copies. Consider three components $d_{1,1}$, $d_{1,3}$, and $d_{1,5}$. These three related components of a heterogeneous data item d_1 are each managed by disparate database managers at sites s_1, s_3, and s_5. A possible D^3 between these components is the following:

$S : \{d_{1,3}, d_{1,5}\};$	– source components
$U : d_{1,1};$	– target component
$P : d_{1,1} = d_{1,3} = d_{1,5};$	– all three components are pure replicas
$C : \epsilon(18 \text{ hrs}) \vee d_{1,3} > 5000;$	– update within 18 hours or if $d_{1,3}$ exceeds 5000
A : *Propagate_Update;*	– action to eventually satisfy P

The C factor in the D^3 specification provides only the consistency *requirement* and not the consistency *constraint.* The effect of a constraint such as 1-SR is implicitly achieved from the combined specifications of P, C and A.

Management of interdependent data is accomplished using the D^3 specification as the basis. An *interdatabase dependency manager* (IDM) is assumed to co-exist with every local database manager LDBM that participates in the multidatabase environment. The IDMs act as an interface between different databases and can communicate with each other.

Transaction management also relies on the IDM. If a transaction updates a data item in a LDBM which is interdependent with data in other LDBM, a series of related transactions may be generated and scheduled for execution at the local and/or remote systems. Such transactions, which are generated solely to maintain the interdependencies that exist between data, are termed *polytransactions.* A polytransaction (T_i^+) is a transitive closure of a transaction T_i submitted to the interdependent data management system. It can be represented by a tree whose nodes correspond to the component transactions with edges defining the *coupling* or association between the parent and child transactions. It has been proposed that the schedule for the execution of these polytransactions be determined dynamically from the declarative specifications of interdependencies in the IDM. Actions may then be performed using event-driven mechanisms like triggers [67].

6.1.4 Weighted Voting in Heterogeneous Databases

In [202], Tang adapts the weighted voting [88] method for replication control in a partition-free heterogeneous system. Quorum consensus is applied to *global copies* located at each site as representatives of the data item in the local database. These copies are also assumed not to be affected by the local autonomy of the item's host site. Associated with each global copy is a version number *gvn* and a forward address to the site having the most current local copy, denoted by *fwd.*

An update to a data item d_i, initiated at a site s_j, is handled as follows. In the read phase of the update, the replication controller at s_j first attempts to collect a read quorum, denoted by r_{d_i}. If this is unsuccessful, the update is aborted. Otherwise, the *gvn* and *fwd* values from all other sites in the quorum are solicited. Based on these values, *M, K,* and *F* are computed as follows: $M = max(\ gvn_k)$, such that $s_k \in r_{d_i}$; $K = \{s_k : s_k \in r_{d_i} \wedge gvn_k = M\ \}$; and $F \subseteq K$, such that there exists fwd_f at each site $s_f \in F$. If $r_{d_i} \cap F = \emptyset$, either abort the read, or wait until sufficient sites in *F* recover, and this intersection becomes non-empty. If $r_{d_i} \cap F \neq \emptyset$, read the local copy from each site $s_k \in r_{d_i} \cap F$.

In the write phase of the update, the replication controller attempts to collect a write quorum w_{d_i}. If the attempt fails, write is aborted. If not, a subtransaction is submitted to the local transaction manager LTM at site s_j to update the copy $d_{i,j}$. If the subtransaction aborts, the write, and hence the update, is aborted. Otherwise, in a *single, atomic* action: set $gvn_j \leftarrow M + 1$; set $fwd_j \leftarrow j$; and send an $update(d_i)$ request to all other sites $s_k \in (w_{d_i} - s_j)$.

Upon receipt of $update(d_i)$ from s_j, each other site $s_k \in (w_{d_i} - s_j)$, submits a subtransaction to the local LTM to update the local copy $d_{i,k}$. If this subtransaction commits, in a single atomic action, s_k sets $gvn_k \leftarrow M + 1$ and $fwd_k \leftarrow k$. If the subtransaction does not commit, s_k sets $gvn_k \leftarrow M + 1$ and $fwd_k \leftarrow j$.

The correctness of this method hinges on the atomic update of both the global and local copies at the coordinator site. The method is tolerant to site failures, provided that the global transaction manager module at the failed site can inquire about the status of pending subtransactions which had been submitted to the LTM before failure but are currently pending at the global level. Compensating transactions are required to resolve partial and incorrect commits.

Given the difficulties presented by site autonomy, it may come as a pleasant suprise to the reader that replication replication in the heterogeneous environment can in fact moderate the impact of site autonomy. We make this observation based on Tang's adaptation of the quorum consensus method. To cope with autonomy, quorums can be selected in such a way as to avoid highly autonomous sites. Furthermore, without *a priori* knowledge of the "degrees of autonomy" of sites, broadcast can be used to solicit consensus of only a quorum of agreeable sites. In the latter case, committing a transaction can survive refusal of one or possibly several autonomous sites. The commitment process, in this case, can be described as a simple Bernoulli process whose probability of success can be increased by parameter tuning, including quorum parameters and number of replicas. One has to admit though that such *autonomy countermeasure* is specific to the quorum consensus method, and is not a grant of replication alone. While Tang [202] did not exploit such an autonomy countermeasure in his algorithm, we give him some credit for leading us to this observation.

6.1.5 Primary Copy in Heterogeneous Databases

In [71] and [118], Du *et al.* proposed a primary copy approach to ensure mutual consistency of replicated data in heterogeneous database systems. The goal is to increase data availability by allowing users to access otherwise remote data locally without compromising consistency (e.g., 1-SR). This approach assumes that there is a primary copy for each replicated data item that contains the most up-to-date value. All updates to the data are first done at the primary copy and then propagated to other copies. The approach is particularly suited to environments where most applications update only local primary copies but may read any non-primary copies at the site. There are two implementations for the proposed approach. The work in [71] presented a pessimistic algorithm which avoids inconsistencies by imposing restrictions on global applications that update data whose primary copies reside at multiple sites. The implementation presented in [118] is an optimistic one which ensures global consistency using a certification protocol. The protocol is an extension to the two-phase commit (2PC) protocol and detects inconsistent access in the first phase of the 2PC (i.e., before parent transactions enter into PREPARED-TO-COMMIT state.) The main advantage of this approach is that it ensures replicated data consistency without compromising local autonomy. The global transaction managers in both pessimistic and optimistic algorithms work as regular users to the local database systems. The only requirement is that an agent be superimposed on

top of each local database system to propagate updates to replicated data by local applications.

6.2 Replication in Mobile Environments

Recent advances in hardware technologies such as portable computers and wireless communication networks have led to the emergence of mobile computing systems. Examples of mobile systems that exist today include portable computers that use cellular telephony for communications and trains and airplanes that communicate location data with supervising systems. In the near future, tens of millions of users will carry a portable computer and communicator that uses a wireless connection to access a worldwide information network. The mobile environment is constrained, however, by the limited communication bandwidth, frequent mobile unit disconnection, and limited mobile unit battery lifetime, among other factors.

In such a mobile environment, the replication of data and services will be essential. This is because users and their transactions move from one cell to another while accessing the database. Unless the data moves with the user in the form of a cache or a mobile disk storage, data will need to be re-transmitted back and forth. Retransmission wastes the limited communication bandwidth and slows down transaction processing because of retransmission delays. Moreover, re-transmission/reception results in faster power consumption of the limited battery resource.

In general, data can be replicated at three levels in the mobile environment [72]. The first level of replication is at the source system level, where multiple copies are maintained at different stationed hosts. Classical data replication protocols are sufficient to manage replication at this level. Dynamic replication [111] at this level can be used to redistribute the data to become closer to the access activities. At the second level, data can be cached into the data access agents (DAA, also known as data proxy) on base stations, where the mobile unit is connected. Only part of the data stored at the stationed host will be cached. Replication at this level is dynamic and relatively volatile. Initially, only one DAA holds a cache for the data when it is first requested by the mobile user. When the user moves, the DAA replicates the cache to as many other DAAs as will support the efficient execution of the transaction. At the third level, the DAA cache can, in turn, be cached at the mobile unit. For, example, a spreadsheet user is most likely to issue many formula evaluations on the same

data sheet before retrieving other data. Caching the spreadsheet will be worth the transmission overhead. Updates to the data cached at the mobile unit (if updates are allowed) will be deferred inside the mobile transaction workspace area. Clearly, maintaining the consistency of the cache and replicas at the three mentioned levels is the focus of replication management in mobile environments.

A pilot study on replication in mobile environments can be found in [13]. In this study, it is assumed that the mobile units may maintain copies of data, as do location servers (processors which reside in the fixed network). Moreover, it is also possible to have cached copies of the data. The study assumes a classification where some mobile units are dedicated as clients and some as servers. Within this framework, the study evaluates six different schemes for placement of data. Depending on the place where a copy of the data can reside, the following six schemes are proposed:

1. The server replicates the copy of the data at the mobile client. On each write, the server needs to write to the copy on the *mobile* client. Writing requires locating the mobile client. Reading is from a local copy on the mobile client.
2. The replicated copy resides at the location server of the client. Thus, the client reads from its own location server. Here, reads and writes are on *static* copies.
3. The server has a copy of the data at its home location server L_S. The client reads from L_S. Thus, reading and writing is on *static* remote copies.
4. The server has a cached copy at its home location server. The location server's copy is invalidated upon the first write since the last read request from the client. Reading an invalid cache will require locating the *mobile* server. However, if the cache is valid, then the read takes place from the copy at the location server L_S.
5. A cache of the server's copy exists at the client's location server. Location server copy is invalidated upon the first write since the last read request from the client. Reading an invalid cache will require locating the *mobile* server. However, if the cache is valid, then the read takes place from the copy at the location server L_C.
6. The cache is maintained at the mobile client. If there was a read since the last update, the server upon each write sends invalidation message to the *mobile* client. If the client wants to read and the cache is invalidated, then the *mobile* server is contacted to complete the read.

A recent research project at Xerox PARC has been investigating replication in the mobile environment. The Bayou system [68] is a replicated, weakly consistent storage system designed for a mobile computing environment that includes portable machines with less than ideal network connectivity. Bayou allows mobile users to share their appointment calendars, bibliographic databases, meeting notes, evolving design documents, news bulletin boards, and other types of data in spite of their intermittent network connectivity. To maximize availability, users can read and write any accessible replica. Bayou's design focuses on supporting application-specific mechanisms to detect and resolve the update conflicts that naturally arise in such a system, ensuring that replicas move towards eventual consistency. It includes novel methods for conflict detection, called dependency checks, and per-write conflict resolution based on client-provided merge procedures. To guarantee eventual consistency, Bayou servers must be able to roll-back the effects of previously executed writes and redo them according to a global serialization order. Furthermore, Bayou permits clients to observe the results of all writes received by a server, including tentative writes whose conflicts have not been ultimately resolved.

Wolfson *et al.* [217] describe research on view maintenance in mobile environments. The objective of the work is to facilitate disconnected operation, by letting the mobile client access and update the view during periods of disconnection. The paper discusses the needed divergence control, ranging from replica maintenance using one-copy serializability requirement, to unlimited divergence, which is the case of the virtual view.

6.3 Replication in Large-Scale Systems

This section describes some of the issues related to replication in systems consisting of large numbers of computing nodes. An example of a typical application involving large-scale replication is an electronic shopping network. Customers of such a system are able to order merchandise from sites scattered across the country; thus the merchandise database is massively distributed. Customer transactions mainly involve read operations, but data updates occur when orders are placed. Data replication must be used to meet the availability requirements of the commercial network. The Internet is another, obvious, example.

Classical data replication protocols—like those described in Chapter 2—perform poorly in such large-scale systems. This is partially due to the considerable

volume of message traffic that is generated to synchronize the large number of replicas placed on a widely-dispersed network. It is also due to the wide fluctuation in communication delays that are common in wide area networks. In addition, the massive number of components (sites, communication links, and software processes) impose a mode of *continuous partial failure* on the system.

A few successful scalable replication methods have been introduced in the literature. The common approach taken by these methods is one of relaxing consistency, in trade for higher performance and availability.

Grapevine [40] is one of the earliest systems that used the weak consistency approach. In this system (and its commercial version Clearinghouse [156]), replicated data is updated at one site, and the update is propagated to other sites in the background. Grapevine uses three methods to propagate updates: *direct mail, rumor mongering,* and *anti-entropy.* Of the three methods, only anti-entropy guarantees delivery to all sites. A variation of Grapevine's *anti-entropy* method called *timestamped anti-entropy* is introduced in [91]. The method guarantees reliable, eventual, but unordered delivery of propagated updates. The method uses a family of communication protocols as an associated group membership mechanism. The details of this method are presented in [91].

The Coda file system [184] addresses issues of site scalability. In this system, entire data files are cached at user sites for performance, and data can be updated in any network partition. A high-availability mechanism, *server replication,* is used to allow volumes to have read-write replicas at more than one server. The performance cost of server replication is kept low by callback-based caching at clients, and through the use of parallel access protocols. Coda also uses data replication for high availability, and employs an optimistic update algorithm. Ficus is another system that uses replication and that addresses site scalability [168]. The scalability in performance and availability that can be achieved in Coda and Ficus costs inconsistencies that are not tolerable in most transactional systems. The cost is much more affordable to a remote file system access by a computer user.

A simulation study on remote file system scalability using non-transactional consistency is described in [182]. The WANCE tool [220] (Wide Area Network Communication Emulation Tool), developed at Purdue, was used to study data replication for distributed database systems that are geographically widely distributed.

More recent research attempts to integrate strong and relaxed consistency in a single framework. Noha Adly's research [1, 2, 3] is one of the few such attempts.

The `Fast_read` and `Fast_write` operations support relaxed consistency, while the `Slow_read` and `Slow_write` operations support strong consistency. The slow/fast protocol also offers different levels of staleness in the hierarchical architecture. Consistency conflicts introduced by `Fast_writes` are resolved off–line and the applications should tolerate some inconsistency in this case.

Sean Liu [139] extends Adly's work of integrated strong/relaxed consistency into a framework that provides transactional access with strict, insular, relaxed, or no consistency. Insular consistency maintains consistent database states, but may allow read-only transactions to access stale but consistent data. In comparison with weak consistency, insular consistency is stronger than weak consistency, but weaker than strict consistency. In Liu's approach, closely-related sites are grouped into a hierarchical cluster architecture. Data replicas are organized into a corresponding hierarchical multi-view structure. Transactions are processed on-line against an abstracted meta-system which is much smaller than the original distributed database system. The multi-view structure allows for further abstraction of the meta-system if higher scalability is desired. The multi-view access protocol maintains multiple levels of consistent views of the data. In particular, it supports scalable user acceptable access through insular consistency. The protocol propagates updates level by level, and guarantees that the database always changes from one consistent state to another, although read-only transactions may access stale data.

7

The Future of Replication

The field of replication needs more experimental research, further study of replication methods for computation and communication, and more work to enable replication in large-scale and heterogeneous environments. The field ripened for systematic experimental research only in the late 1980s and early 1990s; consequently, much still needs to be done. It remains difficult to evaluate the claims that are made by algorithm and system designers, and we lack a consistent framework for comparing competing techniques. As for the replication of communication and computation, they go hand-in-hand with the replication of data in order to enhance availability and performance for systems in which interactions are not through shared data. Only data replication has been studied intensely; relatively little has been done in the replication of computation and communication.

We lack the definitive studies that will resolve conflicting claims made about the relative success of different methods in enhancing availability and performance and about the validity of various assumptions. One of the difficulties in evaluating replication techniques lies in the absence of commonly accepted failure incidence models. For example, Markov models that are sometimes used to analyze the availability achieved by replication protocols assume the statistical independence of individual failure events and the rarity of network partitions relative to site failures. We do not currently know that either of these assumptions is tenable, nor do we know how sensitive Markov models are to these assumptions. The validation of these Markov models by simulations cannot be trusted in the absence of empirical measurements since simulations often embody the same assumptions that underly the Markov analysis. Thus, the field needs empirical studies to monitor failure patterns in production systems

with the purpose of constructing a simple model of typical failure loads. An excellent, but rare, example of such a study is that of Gray [94].

A *failure load model* should resolve such issues as that of modeling the probability of sequences of failures: must joint probability distributions be employed or do correlation coefficients or even an independence assumption suffice? A failure model should include a variety of what might be called *failure incidence benchmarks* and availability metrics that are readily measurable. The development of commonly accepted failure models will clarify how reconfiguration algorithms should exploit information about failures that have been detected and diagnosed. Some methods (e.g., primary copy [9], available copies [24]) call for immediate reorganization after a failure before any further operations on the replicated data can be processed. On the other hand, voting methods [88, 109, 205] call for a reorganization only if the availability must (and can) be enhanced in the face of further failures that may occur before the first one is repaired.

Until failure loads are clarified and documented, validating the analytical and simulation approaches, the most credible alternative is to construct testbed systems for the specific purpose of evaluating the relative performance achieved by different replication techniques. These empirical studies must necessarily involve the tight integration of a number of different algorithms to maintain consistency while achieving replication, reconfiguration, caching, garbage collection, concurrent operation, replica failure recovery and security. This high degree of integration is required because of the tight and often subtle interaction between many of the different algorithms that form the components of the replication suite of algorithms [110, 214],

The second area of needed research is that of replicating computation and communication, including input and output. Replication of computation has been studied for a variety of purposes, including running synchronous duplicate processes [58] and processes implementing different versions of the same software to guard against human design errors [12]. Replication of communication messages has been studied in the context of providing reliable multicast message delivery [39], and a few papers report on the replication of input/output messages to enhance the availability of transactional systems [148]. However, more work needs to be done to study how these tools may be integrated together with data replication to support such applications as real time control systems, benefiting from all three kinds of replication. This work could be invaluable in guiding operating system and programming language designers towards the' proper set of tools to offer replicated systems.

Operating system and programming language mechanisms that support replication include RPCs [119], transactions [136, 80], and multicasts with various synchronization primitives such as those in Isis [38]. Definitive studies that help shape a consensus about the proper combination and features of such tools are much needed. For example, many researchers erroneously consider transactions and multicasts to be alternative support mechanisms for replication, whereas in reality they are complementary. Transactions (especially when available in the form of transactional RPCs with nesting as in Argus [136]) provide the programmer of the replication subsystem with highly flexible exception handling and clean aborts. When a (sub)transaction aborts as a result of failure, the programmer is notified at a lexically fixed spot in the program, and has the choice of either resubmitting the transaction, ignoring it, or performing an alternative transaction. By the same token, the programmer can abort ongoing transactions if they are no longer needed. Transaction properties of concurrency atomicity, failure atomicity, and permanence, have been specified in terms of, and implemented effectively for, processes that interact through shared data. However, in systems whose shared state is not explicitly encoded in shared objects, the vehicle of synchronization and fault-tolerance (hence replication) must necessarily be message passing, in which case a variety of multicast programming primitives [39] would be necessary.

Thirdly, the strong trend towards abstract data types in general and object-orientedness in particular, as the units of modularity in programs, has potentially dramatic effects on data replication. First, such systems often require the persistent storage of typed data that will be very costly to store in a traditional (replicated) file system and unwieldy if stored in a traditional (replicated) database. The reasons for that stem from the inappropriateness of the techniques for buffering, concurrency control, recovery, and disk allocation that are used in file systems and databases for object-oriented systems. While none of these techniques necessarily cause incorrect executions, they do hamper the response time, the degree of concurrency, and the throughput by being optimized for load characteristics that we cannot expect to hold for object-oriented systems. At the same time, the information hiding implicit in typed data offers an opportunity for the storage system (including the replication algorithms) for the systematic exploitation of type semantics.

An example of a type-specific approach is that of *event-based data replication* [100, 109], in which the unit of replication is a single operation execution and not the entire value of the object. Thus, a replica contains a (possibly partial) history or log of operation execution and checkpoint events, rather than a value or a set of versions. By contrast, traditional (value-based) data replication methods maintain copies of object values with an associated log

that is used exclusively for local recovery from site failures. Event-based data replication promises several advantages: more efficient utilization of disks because log appends eliminate disk seek times, higher concurrency by permitting type-specific concurrency control algorithms to operate directly on the log representation, and higher availability and faster response time for common update operations that are semantically blind-writes (log append). In addition, this approach forms a basis for the seamless integration of type-specific and type-independent algorithms for concurrency control, recovery, and replication, in the same system. This is because the event log degenerates into a value if the checkpoints are constrained to be always current.

A

Systems

A.1 Amoeba

Amoeba [201] is an object-based distributed operating system that does file management using two servers: a file server and a directory server. The directory server uses voting with ghosts as the replication algorithm to manage several replicated directories, with each directory providing a naming service by mapping ASCII names to object capabilities. File replication is supported using the regeneration algorithm.

A.2 Alphorn

Alphorn [11] is a remote procedure call environment (RPC) for fault-tolerant, heterogeneous, distributed systems. It is a software environment for programming distributed computer systems (possibly heterogeneous) communicating in a client-server relationship by means of RPC. It supports replication services transparently due to a *causal* broadcast protocol.

A.3 Andrew (AFS)

Andrew [149] is a distributed computing environment, a successful prototype demonstrating the symbiosis between personal computing and time-sharing. It consists of the *VICE* shared file system—a collection of semi-autonomous clusters connected together by a backbone LAN, and a large collection of work-

stations called *VIRTUE* provided with client software to provide the user with transparent access to system resources. The name space of files in each workstation is divided into two subspaces: local and shared. Shared files are handled by a local process called *Venus,* which makes requests to the relevant cluster server in VICE on behalf of the local file system. The hierarchical file name space is partitioned into disjoint subtrees, and each subtree served by a single server, called *custodian.* Replicated copies of such subtrees are supported in Andrew for read-only access. Copies are created by *cloning,* a service which requires manual intervention.

A.4 Arjuna

Arjuna [166] allows the use of passive and active replication. In the former, only one copy of the object is active. All operations from all transactions are directed to this primary copy. At commit time, updates to the primary are checkpointed to all the object stores where a copy of the object resides. Under active replication, operations on the replicated object can be performed at any object store while maintaining mutual consistency.

A.5 Avalon

Avalon [80] is built on top of *Camelot* and is a language for C++ programmers. It extends C++ to support transactions and highly concurrent access to shared data.

A.6 Birlix

BirliX [126] is a distributed operating system that provides distributed naming and location services using replicated directories.

A.7 Camelot

Camelot [80] is a distributed transaction facility built on top of Mach that demonstrates the successful use of the transaction concept for general-purpose applications. Camelot and its forerunner, the *TABS* project [197], use the principles of general quorum consensus and weighted voting for replicated directories in implementing a distributed log facility. A replicated log is an instance of an abstract data type that is an append-only sequence of records containing entries of the type <LSN, Epoch> pair, where LSN is the log sequence number, the key associated with Epoch, the record identifier, and having three operations *ReadLog, WriteLog,* and *EndLog* defined on it. Two server level operations—*ServerReadLog* and *ServerWriteLog* are also defined.

A.8 Coda

Coda [184, 123] is a replicated file system that helps incorporate failure resiliency into the Andrew File System. The basic units of replication are the subtrees we mentioned in the section on Andrew, called *volumes.* File replication is managed using a variant of read-one write all approach. The Venus process services a read request by accessing it from the site having the latest copy from a currently available subset of servers that store the replicas. An updated file is marked and transferred in parallel to all such servers if they are available or notified at a later time. Currency of the replicas is determined by an associated *store id,* a sequence of which forms an update history for use in recovery.

A.9 Deceit

Deceit [187] is a distributed file system implemented on top of the Isis distributed system (see Section 3.5), designed to experiment with a wide range of file system semantics and performance. It provides full NFS capability, file replication with concurrent reads and writes, a range of update propagation mechanisms, and multi-version control. It uses a *write token* protocol to serialize updates.

A.10 Echo

The Echo distributed file system [113] replicates entire arrays of logical disk blocks by virtue of its support for *RAID* (redundant array of inexpensive disks) management protocols. Consistency between replicas is maintained by a primary synchronization site which periodically verifies its quorum with *keep-alive messages.* When the primary synchronization fails, the election of a new primary is done based on ownership of a majority of disks. Witnesses are used to break ties. The potential primary must wait for disk ownership timeouts to expire. Hence, speedy regeneration of the new primary depends on short ownership timeouts and frequent keep-alives.

A.11 Eden

Eden [174] is a LAN-based experimental distributed system which supports user-defined, extensible, mobile objects called *Ejects* that are referenced by *capabilities.* A checkpointing facility is provided for recovery. The regeneration algorithm was proposed as part of the work done on replication of Ejects.

A.12 Ficus-Locus

Locus [170] is a commercial, UNIX compatible, distributed operating system that supports file replication and nested transactions. Files belonging to a logical file group can store their replicas in physical containers which are assigned to the file group and located at a subset of the group members. Each group has at least one of these containers containing all the replicas of files and directories, called the primary copy. Updates can be done only in a partition that has the primary copy. Reads access the most recent version of the file in a partition, but this optimistic strategy implies that stale copies may be read from partitions that do not have a primary copy. Version vectors are used to detect write-write conflicts between the copies.

Ficus [168] is a replicated file system for large-scale networks that supports replication using the above technique and provides automatic reconciliation of conflicting updates when detected and reported.

A.13 Galaxy

Galaxy [193] is a distributed operating system suitable for both local and wide-area networking. To accomplish this, no broadcast protocols, global locking, or time-stamping is required while maintaining consistency of replicated data. Decentralized management, naming schemes, view of the system as objects are part of the system. Interprocess communication is supported in a blocking and non-blocking mode.

A.14 Guide

Guide [15] is an object-oriented distributed system implemented as a tool to investigate mechanisms for supporting cooperative applications distributed on a network of possibly heterogeneous machines. Guide supports object replication via its Goofy server, which provides three main mechanisms: a pessimistic replication algorithm, client caching and lock invalidation.

A.15 Harp

Harp [137] is a replicated file system that is accessible via the virtual file system (VFS) [124]. Harp uses the primary copy replication technique. Clients access a file by requesting a *primary* server, which communicates with *backup* servers as and when required. Updates are done on all the servers using a two-phase protocol. In the event of failure, the server pool runs the virtual partition (view change) protocol to determine a new primary site.

A.16 Isis

A project at Cornell, Isis [39, 120] is a distributed computing system that provides several forms for fault-tolerant computing. It can transform non-distributed abstract type specifications into a fault-tolerant distributed implementation called *resilient objects.* Resilient objects achieve fault-tolerance by replicating the code and data managed by the object at more than one site. The resulting objects synchronize their actions to ensure consistency. We refer the reader to a much more detailed description of Isis in Sections 3.5 and 5.1.

A.17 Mariposa

Mariposa [192] is a distributed database management system that uses economical models to answer queries and to manage and control data replicas. Scaling up to 10,000 nodes is one of the design goals of Mariposa. Scalability is achieved through an asynchronous replication scheme. A transaction update is allowed to commit at one site. Updates are propagated to other copy holder sites within a time limit. Mutual consistency is therefore not guaranteed. Queries are presented to a resolver at each site. Each query specifies the maximum time delay, degree of staleness, and the maximum affordable cost to execute the query. The resolver arbitrates concurrent updates and queries according to their economical–cost models. To answer a query that requires up–to–date data, negotiation with other copy holders is initiated according to a well-defined protocol. Cost and delay is communicated in this protocol. Eventually, the query is answered within the time delay specified and stable results are returned.

A.18 Oracle 7

Oracle 7 [66] implements an *event-driven* replication, in which an application on the source database sends changes to target locations. Triggers on the source tables make copies of changes to data for replication purposes, storing the required change information in tables called queues. The replication catalogue is a set of tables identifying replication definitions and subscriptions—what, how, when, and where data are to be replicated. Changed data is sent out as transactions occur or on a scheduled basis.

Symmetric replication also supports asynchronous procedure calls, so users can update remote data asynchronously as well as the replication of data definitions and stored procedures. If appropriate, the replication environment can be set up so that users can update all data at all sites. Here, Oracle will provide update conflict detection and resolution, including time-based, commutative arithmetic and user-defined methods of conflict resolution.

A.19 Purdue Raid

Purdue Raid [32] is a distributed database system, developed on a LAN of Sun workstations, and provides an environment for experimenting with various protocols and adaptive schemes. Each site has a replication controller that may be configured to execute one of several replication control policies, ranging from *ROWA* to general quorum consensus. Mini-Raid is a smaller version of Raid that focuses on experimentation with network partitioning.

A.20 Rainbow

Rainbow [104] is a a toy distributed database system for class room education. It serves as an exercise to understand concepts of transactions and transaction management, concurrency control, atomic actions, data replication, and fault–tolerance. Rainbow provides experimentation infrastructure that is useful in studying and comparing various and alternate transaction processing protocols. It also facilitates simulation-based experimentation with various patterns of site and communication link failures. Rainbow is an extended version of Seth [106].

A.21 SDD–1

System for Distributed Database *SDD–1* [27] was developed by the Computer Corporation of America. SDD-1 allowed redundant storing of data at several sites to enhance reliability and responsiveness. It was designed between 1976 and 1978, implemented in 1979. This project made a significant contribution to the understanding of the important problems of distributed databases.

A.22 Sybase 10

Sybase 10's [65] Replication Server replicates transactions, not tables, across nodes in the network. Only the rows affected by a transaction at the primary site are sent to remote sites by the Replication Server. Whenever an update occurs to the primary copy of a replicated table, the Log Transfer Manager (LTM), a program that monitors log activity at the primary site, passes the changed records to the local (primary) Replication Server. If the rows are

in tables that are replicated, the Replication Server stores the rows and passes them on to the appropriate distributed Replication Servers. At the remote sites, these rows are then applied to the replicated copies. The Replication Server responsible for each remote site makes sure the transactions are executed in the right order on the copy and applies the appropriate transaction management.

Sybase also now supports an asynchronous stored procedure that can be used by remote sites to update data indirectly. The user at the remote site issues an update via a stored procedure. The stored procedure is passed to the local Replication Server by the LTM and then to the primary Replication Server, which executes the stored procedure at the primary data site. The change is then replicated back to the copy and the remote user. An example here would be a travel agent in a local office changing the address of a client. The travel agent generates the transaction from a copy of the client record, and the update is propagated through the primary data site and back to the copy. Depending on the network configuration, the change could be reflected in the copy in as little as a few seconds. If a component of the network is down, the stored procedure would be queued and then processed after the failed component is restored. The travel agent, in the meantime, can go on and do other tasks due to the asynchronous nature of the stored procedure. This is a new mechanism for Sybase, and it allows Replication Server to perform functions that many real-world customers need and have had to build their own systems to accomplish.

A.23 Yackos

Yackos [108] implements a process group paradigm with dynamic server squads. A squad is a collection of homogeneous processes that cooperate to provide a particular service. The squad grows and shrinks dynamically depending on workload in the system.

B

Further Readings

Each of the following sections presents a review and comparison of a small number of selected papers in each of five subfields of replication. The reviews have been written by invited experts in the respective domains.

B.1 Data Replication

Yelena Yesha, NASA Goddard Space Flight Center
Konstantinos Kalpak, University of Maryland Baltimore County

Reading List

1. Serializability Theory for Replicated Databases. *P. Bernstein, and N. Goodman.* ACM TODS, [25].
2. An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases. *P. Bernstein, and N. Goodman.* ACM TODS, [24].
3. Maintaining Availability in Partitioned Replicated Databases. *A. El Abbadi and S. Toueg.* ACM TODS, [76].
4. Dynamic Voting. *S. Jajodia and D. Mutchler.* ACM SIGMOD, [116].
5. Voting with Regenerate Volatile Witnesses. *J. F. Paris.* IEEE ICDE, [164].

6. Increasing Availability under Mutual Exclusion Constraints with Dynamic Voting Reassignment. *D. Barbara, H. Garcia-Molina, and A. Spauster.* ACM TOCS, [18].
7. A Randomized Voting Algorithm. *A. Kumar.* IEEE DCS, [129].
8. Multidimensional Voting. *S. Chueng, M. Ammar, and A. Ahamad.* ACM TOCS, [55].
9. Read Only Transactions in Partitioned Replicated Databases. *K. Brahmadathan and K. Ramarao.* IEEE ICDE, [47].
10. Managing Event-Based Replication. *A. Heddaya and M. Hsu and W.E. Weihl.* Information Sciences, [102].

One of the central issues in distributed database systems is that of fault–tolerance: how to ensure that data are accessible in the presence of failures. Among the many possible failures in a distributed database system, we usually focus on clean, fail-stop, detectable link and/or site failures. A classical and intuitive line of defense against failures in any physical system is to introduce redundancy and take advantage of such redundancy to combat component failures in the system. Employing this idea, one tries to improve the availability of data objects in a distributed database by replicating some data objects in multiple sites. This leads to the notion of a replicated distributed database system. However, we insist that the replication of data must be done in such a way that it is transparent to the user; a replicated distributed database should present itself to the user as a non–replicated centralized database. Assessing the correctness of a replicated database amounts to replica control (the multiple copies of an object should behave collectively as a single copy from the user's point of view) and concurrency control (a concurrent execution of any set of operations should be equivalent to a serial execution of those operations).

In this section, we annotate a comprehensive collection of papers for further readings in data replication. The papers in this collection are concerned with such issues as correctness criteria of algorithms for replicated databases, replication and concurrency control for replicated databases tolerating site, link, and/or network partitioning, vote assignment and reassignment schemes for replica control schemes, replica control algorithms for predominantly read–only transactions, and for large data objects.

Bernstein and Goodman, in "Serializability Theory for Replicated Databases," consider the problem of proving the correctness of algorithms for management

of replicated data with respect to replication and concurrency control. They extend the classical serializability theory to one-copy serializability theory: a concurrent execution of a set of transactions in a replicated database is one–copy serializable (1–SR) if it is equivalent to a serial execution of all those transactions in a non–replicated database. Bernstein and Goodman establish 1–SR as the common correctness criterion for replicated databases. They go on to prove that an execution is 1–SR if and only if it has an acyclic logical serialization graph. Bernstein and Goodman illustrate their- 1–SR theory by proving the correctness of two algorithms for replicated data: the majority consensus by Gifford [88], and the missing writes by Eager and Sevcik [73].

In "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases", Bernstein and Goodman introduce their available copies algorithm for concurrency control and recovery in replicated databases. Their algorithm keeps, for each data object, a directory of all available copies of that object. These directories are maintained by running special status transactions. Each site that wants to access an object has a copy of its directory of available copies. A site reads one or writes all available copies of an object by consulting that directory. Concurrency control is achieved via a two–phase locking mechanism. In the available copies algorithm, a site can read or write a data object as long as at least one copy is available; thus, it offers the highest availability. Correctness of this algorithm is proved using the 1–SR theory. Their algorithm is biased towards predictable data access patterns and small failure rates. They also assume clean detectable site failures. Moreover, Bernstein and Goodman assume that the failure patterns do not partition the network.[1]

Devising replica control algorithms that operate in the presence of network partitions is a significantly harder problem. The next two papers tackle this problem.

El Abbadi and Toueg present, in "Maintaining Availability in Partitioned Replicated Databases," a replica control algorithm that is resilient to site failures and network partitioning. To achieve this, they use the idea of multiple views. Views allow the system to closely follow changes in the network thus adapting the cost of operations while maintaining high availability. Another key feature of this algorithm is the greater degree of flexibility regarding the availability of data objects and the cost of accessing/updating them. El Abbadi and Toueg accomplish this via read/write accessibility thresholds and quorums. Accessibility thresholds determine the availability of data in all views while the quorums set

[1]The available copies algorithm may lead to inconsistencies among copies of a data object in the presence of network partitions.

in each view determine the cost of reading/writing a data object in each view. They too use 1–SR as the correctness criterion of their algorithm.

Jajodia and Mutchler describe, in "Dynamic Voting," an elegant, simple algorithm that preserves consistency while improving data availability in the presence of network partitions in a replicated database. They accomplish this by a slight but most effective modification of the classical voting algorithm. They suggest that one takes, when an update needs to be performed, the majority of votes from the copies that where updated during the most recent update to an object. Their algorithm has the same availability as a corresponding algorithm by Davcev and Burkhard [61], but it doesn't have the disadvantages of the latter. Using a certain stochastic model, Jajodia and Mutchler also prove that their dynamic voting algorithm has greater availability than the classic voting algorithm under some rather technical conditions.

In "Voting with Regenerable Volatile Witnesses," Paris and Long enhance the dynamic voting algorithm of Davcev and Burkhard by using witnesses and a regeneration mechanism. They observe that the key in improving the availability of dynamic voting is to employ a better tie–breaker. Their witnesses are volatile, require little storage, and incur minimal network and recovery overhead. Regeneration is used to replace failed witnesses. To prevent permanent unavailability of replicated data, witnesses are used as tie–breakers. Paris and Long analyze the availability of their algorithm and compare it with other algorithms. In particular, they find that, under heavy access rates, the availability of their algorithm is comparable to that of the available copies algorithm and significantly superior to that of the majority voting algorithm.

The voting based algorithms mentioned above do not concern themselves with the distribution of voting power of the participating sites. However, as one might expect, the distribution of voting power affects the availability of data objects and the performance. The following two papers are concerned with the question of how to distribute voting power to participating sites so that the availability of data objects is maximized.

Barbara, Garcia–Molina, and Spauster, in "Increasing Availability Under Mutual Exclusion Constraints with Dynamic Vote Reassignment," try to make a system, operating under mutual exclusion constraints, more resilient to future failures by redistributing voting power among sites. It's not hard to realize that mutual exclusion is the core issue in maintaining consistency in replicated databases. The redistribution of voting power can be done in two ways: via group consensus or autonomously. Barbara, Garcia–Molina, and Spauster focus in autonomous methods to perform dynamic vote reassignment. Reassignment

of votes can be done in a coordinated manner by electing a leader (coordinator) to do that task. Otherwise, each site independently decides on the number of votes it wants and requests approval from the majority. Barbara, Garcia–Molina, and Spauster describe policies to decide on how to redistribute voting power and present one–phase protocols that are used to reassign votes to sites. They experimentally compare their scheme with static vote assignment and coordinated vote reassignment. They do allow for site and link failures, and they do not require consistent and instantaneous global network state information at each site.

Kumar considers, in "A Randomized Voting Algorithm," the problem of statically assigning votes to a collection of sites holding a replicated object, given their reliabilities, so that the availability of a majority consensus among them is maximized. He presents a simulated annealing based heuristic, and he reports that the performance of that heuristic is close to optimal for a small number of sites.

So far, votes have been considered as one–dimensional entities. Ahamad, Ammar, and Cheung introduce, in "Multidimensional Voting," the idea of assigning to each site a ***k*–dimensional** vector of weights, and specifying the quorum requirement in another ***k*–dimensional** vector. In their scheme consensus is achieved if the sum of votes collected in each dimension dominates the quorum requirement for at least a given number of dimensions. They show how to find static vote and quorum assignments in their multidimensional voting scheme that can implement any coterie. Thus, they illustrate that multidimensional voting is more powerful than simple (one dimensional) voting schemes. Ahamad, Ammar, and Cheung extend their multidimensional voting to nested multidimensional voting; this is done by replacing the consensus requirement in a single multidimensional voting with another multidimensional voting assignment. The nested multidimensional voting scheme is shown to be at least as powerful as the coteries and quorums constructed by imposing a logical structure on the sites and also shown to be at least as powerful as hierarchical voting. Ahamad, Ammar, and Cheung provide example applications to mutual exclusion and consistency control for fully and partially replicated data items in distributed systems.

Up until now, there has been no preference to read over update transactions to data objects. In certain environments, however, such a preference does exist. Brahmadathan and Ramarao are concerned, in "Read-Only Transactions in Partitioned Replicated Databases," with the problem of increasing availability for read-only transactions in replicated databases in the presence of partitions by penalizing update transactions. Their algorithms use the notion of a group

view, as in El Abbadi and Toueg, such that its log together with one special transaction that only reads all available items in the view is one–copy serializable. The key idea is to use the copies that this special read–only transaction accesses to execute as many of the user's read–only transactions in that group (view). The intractability of finding groups is shown under some conditions, and approximate solutions, by penalizing update transactions or by predefined groups, are presented and used to execute the read/update transactions in the system.

There are cases where the object replicated is a large complex data object. As a result, algorithms that exchange the value of such an object may significantly increase network traffic leading to lower transaction throughput. Heddaya, Hsu, and Weihl, in "Two Phase Gossip: Managing Distributed Event Histories," take another approach; instead of managing the value of replicated object, they manage the history of events, the message exchanges in a network of object servers, for an object. The history of events is capable of representing the state of each object.[2] Replication, in this case, has the effect that each site has only partial knowledge of the event history of the object whose copy it holds. Heddaya, Hsu, and Weihl describe a fault–tolerant protocol whose objective is to reduce the size of event histories that need to be maintained by using checkpoints and by discarding old events. The idea is for each site to forward its checkpoint as far as it can determine an object's history to be complete and purge old events as far as it can determine that all other sites have their own histories complete. In their algorithm, exchange of messages is facilitated by employing a two–phase gossip mechanism.

[2]For example, think of an index structure as the object to be replicated. The value of such an object can be represented by the sequence of operations, together with their arguments and results, performed on the index.

B.2 Process, Object, and Message Replication

Jehan-François Pâris, University of Houston

Reading List

1. Replicated Transactions. *Tony Ng and Shepherd Shi.* IEEE DCS, [153].
2. A Replicated Object Server for a Distributed Object-Oriented System. *Pierre-Yves Chevalier.* IEEE RDS, [54].
3. Distributed Programming with Shared Data. *Henri Bal and Andrew Tanenbaum.* IEEE DCS, [14].
4. Reliable Communication in the Presence of Failures. *K. Birman and T. Joseph.* ACM TOCS, [39].
5. Quorum-Oriented Multicast Protocols for Data Replication. *R. Golding and D. D. Long.* IEEE ICDE, [92].

Processes, messages, and objects are replicated for the same reasons as files or databases: that is to increase the availability of the replicated entities or to distribute a workload that would otherwise be excessive. What is very different is the diversity of the approaches that have been followed. This diversity results from two important factors.

First, processes and objects are more complex entities than files and databases. Hence, many more choices have to be made in the selection of a replication strategy. Replication of processes can either follow a symmetrical approach in which all replicas are active and perform identical functions, or a primary/standby scheme where a single active replica performs the computation and one or more standby replicas remain inactive as long as the primary does not fail.

Second, the objectives being pursued can be quite different. This is in strong contrast to data replication where all extant replication control protocols share the common objective of providing a consistent view of the replicated data in the presence of partial failures. This is nowhere more apparent than in message

replication. Messages are, in some way, the glue that holds distributed systems together. Hence any distributed system incorporating replicated entities will have to use replicated messages whenever the state of the replicated entities is being updated. We can either rely on conventional message passing techniques and leave to the replicated entities the responsibility of handling lost messages and messages arriving out of sequence or use a group communication protocol that guarantees the proper distribution of messages and greatly simplifies the design of the replicated entities. That is to say that any replication control protocol can be at least partially reimplemented as a group communication protocol. Several examples of this duality appear in the literature. The Coda File System [184] uses an optimistic variant of the read- one/write-all protocol and is implemented on top of the MultiRPC parallel remote procedure call mechanism [185]. The Gemini replicated file test-bed which was developed for testing pessimistic voting protocols, included a PARPC [145] mechanism that handled all parallel remote procedure calls but was also responsible for the collection of quorums. Finally, the Deceit File System [187] was built on top of the Isis fault-tolerant group communication mechanism and offered a whole range of replication control protocols.

The five papers selected for further readings provide an excellent example of the variety of solutions proposed for replicating processes, objects, and messages. They also provide a good overview of the most recent works in the field.

The first paper, "Replicated transactions" by Ng and Shu, is an example of the symmetric approach to process replication. A major issue in this approach is the fact that the replicas will reach inconsistent states whenever they execute a non-deterministic computation. The Circus system [58] avoided this problem by requiring all computations to be deterministic while Clouds [59] committed only one replica of each replicated computation and copied its state into the other replicas. Ng and Shu propose a replicated transaction model where each replica has some buffer space where it maintains all non-committed transactions and the locks they own. Their model also allows the user to specify the level of resiliency of each transaction, the number of hardware failures the transaction will be guaranteed to survive.

The second paper, "A replicated object server for a distributed object-oriented system" by Chevalier, presents the Goofy replicated object server. Goofy was built to provide an experimental object-oriented distributed system with a reliable persistent storage that would guarantee object consistency in the presence of network partitions. Goofy utilizes a pessimistic quorum consensus protocol but includes two mechanisms to increase the availability of the replicated objects. First Goofy gives to its clients the ability to operate on their own

execution memory as soon as they have loaded all the objects that they need. Second, Goofy automatically places in the dead state all locks belonging to memory managers that have crashed and lets other memory managers load the object owned by the crashed memory manager.

The third paper, "Distributed programming with shared data" by Bal and Tanenbaum, presents an object-based model for a distributed shared memory and a new programming language, Orca, based on this model. Their shared data object model includes object replication because it allows the run time system to create a new replica of a data object whenever it detects that a processor frequently reads the state of a remote object. Changes made to the data objects are propagated to all replicas by an update protocol that guarantees that all processors will observe modifications to different data objects in the same order. One of the most interesting features of the paper is that it presents an application of object replication where the main objective is to reduce read access time rather than to increase object availability.

The fourth paper, "Reliable communication in the presence of failures" by Birman and Joseph, presents the three group communication primitives of the Isis system. These three primitives are atomic broadcast (*abcast*), causal broadcast (*cbcast*), and group broadcast (*gbcast*). Atomic broadcast guarantees that all destinations receiving *abcast* messages will receive them in the same order even in the presence of site failures. Causal broadcast allows the sender to specify delivery ordering constraints such as this message should always be delivered after that one. The last broadcast primitive, group broadcast, is used to inform members of a group of processes of any change in the membership of a group or the states of its members. It guarantees that: (a) the order in which *gbcasts* are delivered, relative to the delivery of all types of broadcasts, is the same at all destinations, and, (b) that the *gbcast* announcing the failure of a processor is always delivered after the last message sent by the failed process has been received.

The last paper, "Quorum-oriented multicast protocols for data replication" by Golding and Long, presents a set of four protocols providing efficient communication services for replicated data. The major distinctive feature of these quorum-oriented multicast protocols is that they attempt to minimize the cost of delivering a message to a given fraction of the number of replicas, say three out of five. They also address the problems associated with transient failures, and therefore, are particularly suited to the management of replicated data located on a wide-area network.

B.3 Replication in Heterogeneous, Mobile, and Large-Scale Systems

Mike W. Bright, University of Pittsburgh

Reading List

1. Primarily Disconnected Operation: Experiences with Ficus. *J. Heidemann, T. Page, R. Guy, and G. Popek.* Second IEEE Workshop on Management of Replicated Data, [169].
2. Weak Consistency Group Communication for Wide-Area Systems. *R. Golding.* Second IEEE Workshop on Management of Replicated Data, [91].
3. Managing Replicated Data in Heterogeneous Database Systems. *J. Tang.* IEEE RDS, [202].
4. Replication and Mobility. *B. Badrinath and T. Imielinski.* Second IEEE Workshop on Management of Replicated Data, [13].

Modern organizations tend towards large, heterogeneous computer systems. Many business disciplines are incorporating mobile workers in their online systems. These environments pose tremendous challenges to replication control theories and algorithms developed in more pristine research settings.

Even small organizations typically have (or plan to have) connections to larger networks such as the Internet or to external customers for Electronic Data Interchange (EDI). Larger organizations typically administer their own complex LANs (Local Area Networks) and WANs (Wide Area Networks), independent of their external connections. Large networks often imply that disconnected portions and network partitions are the norm, rather than a periodic inconvenience to deal with via specialized mechanisms. Algorithms that rely on connectivity or communication time guarantees may break down or degrade in a large-scale environment. Replication control data structures that work well in prototypes may fail when they scale up to systems several orders of magnitude larger. Complex topologies provide rich environments for replication.

For political and financial reasons, large distributed systems tend toward heterogeneous networks, computers, operating systems, databases, applications,

etc. Distributed control solutions must deal with pre-existing infrastructures, rather than assuming a top-down, total redesign. Everyone's drive towards new, open, client/server solutions must consider how to integrate legacy systems which will likely survive well into the next century. System heterogeneity means global control solutions must integrate local control components with differing assumptions, requirements, and solutions. Many heterogeneous systems support local autonomy. Local systems may not cooperate with global control functions or provide sufficient local functionality to satisfy global requirements. Local semantics may differ, even for seemingly similar data or functions. In addition to making replication allocation decisions, the control logic must search for previously unknown replications or partial replications in pre-existing systems.

Mobile and ubiquitous systems attempt to bring computing closer to the point of actual use. This frequently means there is no fixed network connection available at all possible service points. Hence the need for wireless networks or manual alternatives (e.g., laptop PCs that plug into fixed base stations). Mobile systems imply disconnected nodes are the norm, and limited communication bandwidth is available during connected periods. Mobile nodes must store all data and functions that may be used during the next disconnection period. However, these entry level system nodes are often relatively small systems such as laptop PCs, Personal Digital Assistants, or specialized data entry modules. This implies limited storage and capability to support global functions. Mobile computing in general is still a wide open field with many competing control theories and architecture paradigms. Replication control relies on many other functions such as concurrency control, addressing, search, time management, security, etc. Most of these are still unsettled in the mobile environment.

Selected Solutions

The papers that we recommend for further readings represent sample solutions to some of the problems related to large-scale, heterogeneous, and mobile systems. They illustrate the range of research, yet their focused attacks and limited number demonstrate that there are many open problems. This selection also illustrates the variety of system objects that may be replicated in a distributed system—files, processes, and data (frequently in distributed database systems).

Heidemann *et. al.* present a replicated file system in use at UCLA in "Primarily Disconnected Operation: Experiences with Ficus." Ficus assumes an environment that is primarily disconnected (partitioned) and applies an opti-

mistic concurrency control algorithm. Optimistic concurrency control is quite appropriate in this environment as it allows available copies to always be updateable. This paper illustrates how different assumptions about the underlying architecture affect the choice of control algorithms.

Traditional ideas about consistency and serializability are often inappropriate in a large, heterogeneous environment. For example, a variety of research is underway to explore alternatives to ACID (atomicity, consistency, integrity, durability) transactions in the multidatabase (heterogeneous, distributed databases) environment [115]. The analogue for replication control is to allow some form of weak consistency. This implies that all replicas are not updated synchronously; there are some forms of delayed propagation of updates. "Weak Consistency Group Communication for Wide-area Systems" by Golding discusses a taxonomy of weak consistency protocols and describes the timestamped anti-entropy family of protocols. The taxonomy illustrates the different variables to be considered, such as message delivery guarantees, delivery ordering, and time of delivery. The anti-entropy protocols show the affects of design trade-offs in this environment. The protocol is applied to replicated processes.

There are a variety of quorum consensus protocols and corresponding concurrency control mechanisms for dealing with replication in classical distributed database systems. Heterogeneous distributed databases often support local autonomy [115]. Each local database may run a different concurrency control mechanism. In addition, the global control module cannot guarantee local commits. Providing update guarantees in this environment is quite difficult. In "Managing Replicated Data in Heterogeneous Database Systems," Tang provides a method for guaranteeing appropriate replication control. Global copies of control information are introduced which correspond to local data copies. Since global data structures are under complete control of the global software, their consistency can be guaranteed. Appropriate management of the global copies allows modified quorum consensus protocols to provide traditional replication control.

Badrineth and Imielinski describe some of the difficulties associated with replication control of mobile copies in "Replication and Mobility." In a mobile environment, there are no guarantees of communication latency or bandwidth; costs will be asymmetric over time. There are also a variety of alternatives for locating the mobile node within the overall system. These factors complicate the decision of where to allocate replicas. The paper describes some alternatives and provides a preliminary evaluation of different strategies.

Open Issues

There are a variety of open issues in the large, heterogeneous, and mobile environments. Following are just a sample to motivate the interested researcher or practitioner.

One example is the need to negotiate weak consistency performance and guarantees [8]. Golding discusses the variety of weak consistency factors and possibilities. In any given environment, users and participating systems may want to choose different protocols to meet specific requirements. Mechanisms must be provided to allow selection of weak consistency guarantees dynamically.

Pre-existing systems may bring widely differing data semantics when joining a heterogeneous system. It can be quite difficult to determine what information is equivalent or similar in the face of syntactic differences [190]. In this case, replication control mechanisms must deal with existing copies, rather than allocating copies by design. The semantics of different replicas may be subtly different. How do you propagate updates to similar replicas, as opposed to exact replicas? In addition, the semantics of locally autonomous data may change over time (see Ventrone and Heiler, pp.16–20 in [190]). Replication control mechanisms must be able to recognize this evolution and change appropriately.

Metadata management is an important consideration in large, heterogeneous, and mobile environments. Every data item has associated information (collectively designated "metadata") about its semantics (e.g., data type or domain), precision, ownership, security attributes, content summary information (for searching), etc. Should metadata be replicated with each copy of the data item? How are updates to the metadata propagated?

The scale and practical requirements of large, heterogeneous, and mobile systems make them a fertile area for future research.

B.4 Availability and Performance

Waleed A. Muhanna, The Ohio State University

Reading List

1. Analyzing Availability of Replicated Database Systems. *B. Bhargava, A. Helal, and K. Friesen.* Journal of Computer Simulation [35].
2. Cost and Availability Tradeoffs in Replicated Data Concurrency Control. *A. Kumar and A. Segev.* ACM TODS, [131].
3. Performance Enhancement through Replication in an Object-Oriented DBMS *E. Shekita and M. Carey.* ACM SIGMOD, [189].
4. Bounds on the Effects of Replication on Availability. *L. Raab.* Second IEEE Workshop on Management of Replicated Data, [175].
5. The Location-Based Paradigm for Replication: Achieving Efficiency and Availability in Distributed Systems. *P. Triantafillou and D. Taylor.* IEEE TSE, [209].

Increased availability and enhanced performance are two important objectives of replication in distributed systems. But without a good understanding of the nature and potential magnitude of the costs of replication, it would be difficult to assess the conditions under which replication is cost–effective and the extent to which increased replication can be useful. Obtaining such a general understanding represents a major challenge—for two primary reasons.

First, the three factors (degree of replication, availability, and performance) are all related and interact in many complex and at times conflicting ways. If only read operations are allowed, the picture is relatively simple; increased replication has a positive impact on both availability and performance at the cost of additional storage requirements. In the presence of site and communication link failure, the availability of a data object for read operations is increased by redundantly storing duplicate copies of an object at different sites with independent failure modes. Replication also enhances the locality of reference by allowing more read requests to be satisfied locally, resulting in lower latency (response time) for read operations and reduced network traffic.

It is with write requests, however, that the picture becomes complex. An update control and propagation mechanism, with all the additional communication and processing costs it entails, must be employed when a replicated object is updated in order to resolve conflicting concurrent updates and ensure consistency among the various copies. Increased replication can therefore significantly impair the performance of update operations, a loss which must be traded off against performance and availability gains for read operations. The tradeoffs become even more complicated in the case of applications that require complete replication transparency (i.e., enforcing the one–copy serializability consistency constraint). Here, in addition to synchronizing update transactions, read transactions must also be globally synchronized with update operations to prevent conflicts and to ensure that each read operation is based on a current (up–to–date) copy. Thus, even for read operations, the potential performance gains due to replication must be balanced against the loss in performance due to the overhead required to enforce the consistency constraint. Additionally, a tradeoff now also exists between potential availability increases due to replication and availability decreases due to the need to enforce the consistency constraint.

Another issue that complicates the analysis of the impact of replication on performance and availability is the fact that the tradeoffs among these variables cannot be understood in isolation. Instead, they must be analyzed within the context of a multitude of other interacting factors. These factors include workload characteristics, application requirements, and the replication control algorithm used. Moreover, the overall performance and availability of a system are affected respectively by the performance and reliability characteristics of its constituent hardware and software subsystems. As such, system characteristics (e.g., individual host speed and reliability, multiprogramming level) and network characteristics (e.g., network topology, communication link bandwidth and degree of reliability) must also be taken into consideration when assessing the potential costs and benefits of increased replication with respect to performance and availability.

While considerable progress in understanding the tradeoffs involved has been made, there are significant gaps in our understanding and much remains to be done. The five papers that we recommend for further readings employ a variety of research methodologies and serve to illustrate the complexities involved analyzing various design tradeoffs. Each paper focuses on a somewhat different aspect of the problem and under certain assumptions. Additionally, as described below, the papers reveal the existence of varying conceptions of the term “availability” and different performance measures –all must be kept in mind when analyzing and comparing reported results.

The first paper by Bhargava, Helal, and Friesen helps put in perspective some of the issues involved in analyzing the impact of replication on availability and performance. The paper reports results from two distinct studies. The first study investigates, using simulated transaction workloads on the Purdue RAID (Reliable, Adaptable, and Interoperable Distributed System) experimental distributed database system, the relative performance of two replication control methods, both of which are special cases of the majority consensus protocol with weighted voting. Performance is assessed with respect to both response time and "algorithmic availability" (which they define as the probability that a transaction successfully starts and finishes execution). The experiments conducted measure the effect of the degree of replication on the performance of the two replication control methods under varying mixes of read–only and update transactions and varying degrees of workstation reliability. One result clearly shows increased replication does not always yield increased data availability. This result holds even under a transaction mix with low (20%) write–to–read transaction ratio, when a read–one–write–all replication control algorithm is used. The second study examines, via an experiment on the SETH replicated database prototype, the effect of the overhead of the read–same–as–write quorum consensus method on "operational availability," a metric which for the purposes of the study measures the maximum transaction load under which a system remains operable, even in the absence of site and communication–link failure. Also reported in the paper are the results of a simulation study that briefly highlight the interaction between some network characteristics (message queue length), replication, and performance.

An important property of the weighted voting replication control protocol is its flexibility; values for the parameters of the algorithm (the site vote assignments and quorum sizes) can be selected to fit the requirements of a particular application environment. The second paper in this readings list (By Kumar and Segev) considers some aspect of this voting assignment problem, under the assumption of full replication (i.e., a copy of each replicated object is stored at every site in the distributed system). Kumar and Segev formulated three models to study the tradeoffs between availability and communication cost. In each model, the objective is to find a voting assignment that minimizes communications cost. Voting assignments derived in the first model are unconstrained. Analysis of the model reveals that the read–one–write–all algorithm is optimal when the majority of operations applied to the replicated object are read only operations. Voting assignments derived in the other two model are subject to a failure resiliency constraint. The resiliency criterion is failure tolerance in the second model and availability in the third. Analysis shows that, when the fraction of writes is relatively small, a voting assignment that would satisfy a need for higher resiliency would also necessitate incurring significantly greater

communications cost. The paper shows that, under certain assumptions (e.g., equal votes), the three models can be solved optimally. Heuristics are proposed for solving other more general cases of the voting assignment problem.

While the notion of replication is almost exclusively discussed within the context of distributed systems, its applicability as a performance enhancement vehicle is not, and need not, be limited to such systems. Replication can also be cost–effective in single-site or centralized systems or within the same site in a distributed system, as the third paper in this readings list illustrates. In it, Shekita and Carey describe a simple, yet surprisingly effective, replication technique that can be used to speedup query processing in object–oriented database systems where reference attributes are used to represent relationships among object types.

The technique, which Shekita and Carey call field replication, allows the selective replication of individual data fields. Reference attributes are used to specify the data field or fields (in the referenced data set) that are to be replicated. Once replicated, values in a data field can be retrieved much faster because the expensive functional join operation (or sequence of operations) that would otherwise be required can be eliminated or made much less costly. Two replication strategies are proposed: in–place replication in which the replicated data values are stored (like any other field) in the referencing object itself and separate replication in which replicated values are stores in separate shared sets of objects. Shekita and Carey then show how updates can be efficiently propagated to replicated values. The scheme involves the use of a special storage structure called an inverted path –a set of link objects that implement a reverse mapping from base (referenced) objects to other objects that reference them. The authors also develop analytical cost model to evaluate the performance (measured in terms of expected I/O cost) of the two replication strategies relative to each other and relative to a situation involving no replication at all. Their results show, as one might expect, that in–place replication performs better than separate replication in situations where the update frequency is small and where the level of sharing (the number of references to an object) is very low. Of particular interest, however, is the finding that, at the cost of increased storage requirement, and except for update–intensive situations, substantial performance gains can be realized by appropriately adopting one of the two proposed replication strategies.

The application of Shekita and Carey's proposed replication technique is not limited to object–oriented database systems either. It, as they note, can be adapted to other data models that support reference attributes and referential integrity. One particular interesting avenue for further research is to explore

how replication might be used to eliminate or reduce the cost of value–based joins that would be normally required during query processing in traditional relational database systems. Normalization theory provides a rigorous set of rules for the design of a relational database schema. These rules are designed to prevent update anomalies and potential inconsistencies that arise when redundancies are present in the data. The normalization process can be seen as an exercise in eliminating such redundancies, ensuring, whenever possible, that each single fact is stored in only one place. Normalization may, therefore, be viewed as the antithesis of replication. This, however, does not mean that the two notions cannot happily coexist. They can, albeit at two different levels of abstraction. The soundness and desirability of normalization at the conceptual schema level is clear, but there is no obligation to honor normalization rules at the internal schema (storage) level. There, judicious use of replication may very well yield significant performance benefits for a modest cost.

Also included in this list is a short position paper by Larry Raab. The paper reports briefly on some simulation and theoretical results of research studies examining the impact of replication on site availability in distributed systems in which the enforcement of the mutual exclusion (one–copy serializability) consistency constraint is required. Raab uses an availability metric called site availability which he employs to refer to the probability that a data operation submitted to some site is allowed to succeed even in the presence of failures. Replication and the requirement for mutual exclusion drive the site availability metric in different directions, and the paper focuses on examining the tradeoff involved between the two factors. One important result reported is that the mutual exclusion requirement by itself, irrespective of the consistency protocol or network configuration used, is by far the most dominant and significant factor in the sense that it places a rather low cap on the potential benefits of replication with respect to availability. A theorem establishes that the upper bound for availability under any replication scheme equals the the square root of the availability provided by a "well–placed" nonreplicated data object. This, as the paper discusses, has implications for practice and suggests interesting and more focused directions for further research.

The final paper by Triantafillou and Taylor describes a new replication control algorithm, called the location–based paradigm, that is designed to provide virtually the same degree of flexibility and data availability that traditional quorum–based algorithms offer, but with lower transaction–execution delays. The protocol involves the use and maintenance, as part of normal transaction processing, of a replicated location table that contain information about the list of sites having an up–to–date copy of a requested object. Another component of the proposed protocol is an asynchronous concurrency control

mechanism wherein each lock is obtained at one (dynamically selected) object replica, called the leader, which in turn asynchronously (i.e., in the background) propagates the lock–acquisition request to the other replicas. One–copy serializability and transaction atomicity are enforced at commit time through a two–phase commit protocol. The paper demonstrates that it is possible to enhance data availability through replication without incurring significant performance penalties. Results from a simulation study reported in the paper clearly show that, with respect to average transaction latency, the proposed scheme outperforms both primary–copy and quorum–based methods under a workload consisting purely of small transactions. How the method performs in general is an interesting issue worthy of further investigation.

B.5 Implementations

John Riedl, University of Minnesota

Reading List

1. Coda: A Highly Available File System for a Distributed Workstation Environment. *M. Satyanarayana et al.* IEEE TOC, [184].
2. Efficient Availability Mechanisms in the RAID Distributed Database System. *B. Bhargava and A. Helal.* ACM CIKM, [34].
3. Replication in the Harp File System. *B. Liskov, et-al.* ACM SOSP, [137].
4. Accessing Replicated Data in an Internetwork. *A. Golding and D.D. Long.* Journal of Computer Simulation, [90].
5. Some Consequences of Excess Load on the Echo Replicated File System. *A. Hisgen et-all.* Second IEEE Workshop on Management of Replicated Data, [113].

There is a convergence of opinion among distributed systems researchers that effective use of distributed systems will be based on distributed replicas. The five papers in this section represent two independent traditions within distributed computing: distributed file systems for clusters of workstations and distributed database systems for transaction processing. All five propose similar models: replicated copies maintained in consistent versions on distributed nodes. The differences in approach are instructive. The file system papers emphasize availability, at the possible cost of consistency, with users being in control of the degree to which consistency can be traded off. Distributed file systems are commonly used in applications in which individual users own individual files, and can mediate access. In contrast, the database papers emphasize consistency with availability as high as possible within a framework of complete consistency. Databases are commonly used in applications in which multiple users simultaneously access files with the database system mediating access.

The three tools computer scientists use to understand computer systems are, in order of increasing acuity and increasing cost, mathematical analysis, simulation, and implementation. The distributed systems described in these papers

are too complex to yield to mathematical tools alone. Simulations are most useful for phenomena that are relatively well understood, so the crucial components can be modelled effectively. Implementations teach us first-hand about the difficulties of designing and building complex systems with appropriate transparencies. Further, implementations enable us to become users of distributed systems while we are also their designers. The interplay between the experience of building such a system and the experience of using it provides valuable feedback to the design process. As we learn from the systems we build, experimentation can serve as a vehicle for communicating our new knowledge to other scientists. Implementations of distributed systems enable us to study their complex phenomena directly, rather than from a distance. We always learn from such a study, just by being near to the system and thinking hard about its problems and features. Passing these hard-won lessons on to others is difficult because they have not been through the experience of implementing the system and often do not have experience interacting with it. In the physical sciences, formal experiments have long been the mechanism for capturing and passing on these lessons. Computer scientists have lagged in experimentation in the past [206] but are turning to experimentation increasingly. Every one of the papers in this section includes experimental results or careful descriptions of experience with a distributed system.

The first paper, "Coda: A highly available file system for a distributed workstation environment," by Satyanarayanan, Kistler, Kumar, Okasaki, Siegal, and Steere, describes a distributed file system designed to operate even during server and network failures. The insight in Coda is that the relatively large secondary store available on a modern workstation makes possible *disconnected operation,* in which all the files a user needs are available locally. By preparing for disconnected operation, a workstation insulates its user from temporary network or server failures. Disconnected operation has particular benefits for portable computers which are only occasionally connected to networks. Experiments with Coda quantify the performance effect of the Coda replication scheme on distributed file accesses.

Distributed file systems are built to improve availability, sometimes at the cost of performance. The second paper, "Replication in the Harp file system," by Liskov, Ghemawat, Gruber, Johnson, Shrira, and Williams, describes Harp, a new file system that seeks to use simple, elegant design to provide both high availability and good performance. Harp guarantees failure and concurrency atomicity using a primary copy method. Harp is currently implemented on top of NFS, but experiments show that its performance is still comparable to native distributed file systems because of the benefits of replication.

The third paper, "Some consequences of excess load on the Echo replicated file system," by Hisgen, Birrell, Jerian, Mann, and Swart, describes experience with a distributed file system under conditions of high load. Experience with Echo shows that high load can result in denial of service, even when the network and servers are functioning. The authors discuss deficiencies in primary-secondary design, communications, and their election algorithm under excess load. This paper is particularly valuable for its analysis of problems with a running system that can aid future designers. Election algorithms have been studied carefully under conditions of light load, but the present paper suggests that they may be inappropriately triggered under conditions of heavy load and may function poorly in those conditions. The paper reinforces the important lesson that distributed systems cannot assume that a site has failed when it does not respond quickly. The authors complete their paper with a discussion of possible solutions to the problems they have uncovered.

The fourth paper, "Efficient availability mechanisms in distributed database systems," by Bhargava and Helal, describes resiliency algorithms in the Purdue RAID distributed database system. Purdue RAID provides a set of flexible mechanisms for managing replication policies during the execution of distributed transactions. The paper describes the implementation of a surveillance mechanism that oversees the replication controllers in a RAID system. The surveillance mechanism provides the system administrator with powerful tools for redistributing responsibility among sites during failures, with the goal of better tolerating future failures. The paper includes performance measurements that show that flexible methods can be implemented at reasonable cost and suggests ways these mechanisms can be used to improve availability.

The fifth paper, "Accessing replicated data in an internetwork," by Golding and Long, describes a suite of replica-access algorithms that are appropriate for a wide-area environment. Most research in transaction processing has assumed the replicas are near each other, but this paper directly approaches the problem of replica management in the case that some of the replicas are much further than other replicas. Widely dispersed replicas are likely to be important in providing very high availability in systems that must be resilient to problems that span a geographical region such as hurricanes or earthquakes. This paper presents algorithms that choose the replicas to access based on the projected cost in time to access them. There is an intriguing tradeoff between the time required to access a replica and the number of messages sent. Experiments show that reducing the length of the retry timeout can decrease the latency, at the cost of increasing the number of messages sent. Using the new algorithms, a systems administrator could build a model relating the cost of slower reply

to the cost of additional messages to tune the cost-performance for the system as a whole.

This page intentionally left blank

C

Serializability Theory

This appendix tersely summarizes, the basic definitions and central theorem of serializability theory, then defines the most important correctness condition for replication control protocols, namely, *one-copy serializability.* This formalism captures our intuitive understanding of the flow of information between transactions through modifying and reading the shared data items. For a more detailed presentation of serializability theory, we refer the reader to Chapters two and three in Papadimitriou's book [159]. Our condensation below closely follows Chapters two and eight of Bernstein, Hadzilacos and Goodman's book [26], which gives an excellent treatment of the subject, without being overly formal.

We denote the execution of an operation o on a logical data item x, on behalf of transaction T_i, by $o_i[x]$. For our purposes in this condensed exposition, we limit consideration to read, $r_i[x]$, or write $w_i[x]$ operations only, *i.e.*, $o \in \{r, w\}$. A *transaction* T_i consists of a set of such operation executions—or, equivalently, *events*—and a termination event t_i that can be either a commit c_i, or an abort a_i. That is, $t_i \in \{c_i, a_i\}$. The program whose execution generates transaction T_i induces an irreflexive partial order $\prec_i$ on its events, which sequences conflicting events[1] belonging to the same transaction, and ensures that the termination event t_i follows all other events. Formally,

$$a_i \in T_i \text{ iff } c_i \notin T_i,$$
$$\forall e \in T_i : e \preceq_i t_i, \text{ and,}$$
$$\forall o_i[x], w_i[x] \in T_i : \; o_i[x] \prec_i w_i[x] \; \vee \; w_i[x] \prec_i o_i[x].$$

[1]Two events conflict if at least one of them is the execution of a write operation.

A *history* is a prefix[2] of a complete history. A *complete history H* over a set of transactions T = {T_1, T_2,..., T_n } consists of the set of all events belonging to transactions in *T,* together with an ordering relation $\prec_H$ that incorporates transaction orderings, and sequences conflicting events belonging to different transactions.

$$H = \bigcup_{i=1}^{n} T_i,$$
$$\prec_H \supseteq \bigcup_{i=1}^{n} \prec_i, \text{ and,}$$
$$\forall o_i[x], w_j[x] \in H : o_i[x] \prec_H w_j[x] \vee w_j[x] \prec_H o_i[x].$$

We say that a transaction T_j *reads-x-from* transaction T_i iff $\exists w_i[x] \prec_H r_j[x]$ and there does not exist a write $w_k[x]$ such that $w_i[x] \prec_H w_k[x] \prec_H r_j[x]$.

A *committed projection* $C(H)$ of a given history *H,* is the history obtained from *H* by deleting from it all operations that do not belong to committed transactions.[3] Two histories *H* and *H'* are *equivalent* ($H \equiv H'$) iff they have the same set of committed events, and induce identical reads-*x*-from relations on T.

$$C(H) = C(H'), \text{ and,}$$
$$T_j \textit{ reads-}x\textit{ -from } T_i \text{ in } C(H) \text{ iff } T_j \textit{ reads-}x\textit{ -from } T_i \text{ in } C(H').$$

If transaction programs are deterministic, this definition of equivalence means that all writes store the same values, and all reads return the same values in $C(H)$ and $C(H')$, and hence the two histories have no substantial differences.

We are now ready to define the basic notion of correctness of concurrent database systems, and it hinges on us assuming that transaction programs are correct in the absence of concurrency, *i.e.,* in a serial history. A history *H* is *serial,* if for every pair of transactions T_i and T_j that appear in *H,* either all events of T_i precede all events of T_j or vice versa. In other words, there can be no interleaving of operations of different transactions. A history *H* is *serializable* if it is equivalent to some serial history H_s.

This definition, as straightforward as it is, tends to be hard to prove directly from the statement of a concurrency control protocol. To aid in constructing such proofs, we define a more tractable object, called the serialization graph.

[2]A partial order $L' = (E', \prec')$ is a *restriction* of another partial order $L = (E, \prec)$ on domain E', if $E' \subseteq E$, and, $\forall a, b \in E', a \prec' b$ iff $a \prec b$. V is a ***prefix*** of L, written $L' \leq L$, if L' is a restriction of L, and, for each $a \in E'$, all predecessors of a in L (i.e., all $b \in E$ such that $b \prec a$) are also in E'.

[3]Formally, $C(H)$ is a restriction of H on domain $\cup_{c_i \in H} T_i$.

A *serialization graph* for *H,* denoted *SG(H),* is a directed graph whose nodes are the transactions in *T* that committed in *H.* An edge $T_i \rightarrow T_j$, $i \neq j$, is present in *SG(H)* iff T_i reads-from T_j.

Serializability Theorem. *A history H is serializable if and only if SG(H) is acyclic.* (The proof can be found in [26, 159].)

One-copy Serializability

The notion of serializability falls short when there is more than one copy of a logical item x, since individual events involving different copies can be serializable, even though operations executed on the logical item as a whole are not. We need to extend the notion of equivalence to a serial history, to that of equivalence to a serial history that has no replication, hence the name of this correctness condition. A *replicated data history* is one that contains, for every event involving a logical data item, a set of events involving some of its replicas. Every logical read event $r_i[x]$ is translated to a single physical read event $h(r_i[x]) = \{r_i[x_a]\}$ on one replica, and a logical write event $w_i[x]$ is mapped to a set of physical writes $h(w_i[x]) = \{w_i[x_{a_k}] : k = 1, 2, \ldots\}$ on a subset of the replicas. The resulting history must also respect basic well-formedness properties by: disallowing the reading of uninitialized copies, satisfying transaction program ordering, and sequencing every pair of conflicting events:

$$\forall r_j[x_a] \in H : \exists w_j[x_a] \prec_H r_j[x_a],$$
$$e_i \prec_i e_i' \Rightarrow \forall \varepsilon \in h(e_i), \varepsilon' \in h(e_i') : \varepsilon \prec_H \varepsilon',$$
$$\forall w_i[x] \prec_i r_i[x] : \exists w_i[x_a] \prec_H r_i[x_a], \text{ and,}$$
$$\forall o_i[x_a], w_j[x_a] \in H : o_i[x_a] \prec_H w_j[x_a] \vee w_j[x_a] \prec_H o_i[x_a].$$

It is quite straightforward to define the reads-from relation for replicated data histories. We say that a transaction T_j *reads-x-from* transaction T_i iff $\exists w_i[x_a] \prec_H r_j[x_a]$ and there does not exist a write $w_k[x_a]$ such that $w_i[x_a] \prec_H w_k[x_a] \prec_H r_j[x_a]$

For a replication control protocol to maintain consistency, it must be the case that, for every possible replicated data history H, there exists a one-copy serial history H_s that is equivalent to *H.* Our choice of definition for history equivalence above, serves us equally well in the replicated case, with a minor modification that maps events back from replicas to logical items. Two (possibly replicated) data histories *H* and *H'* are *equivalent* ($H \equiv H'$) iff they have the same set of committed *logical* events, and induce identical reads-*x*-from re-

lations on *T*. To arrive at logical events from those involving replicas, it suffices to remove the suffixes that identify the individual replicas.

Facilitating proofs of correctness of replication control protocols requires a construction similar to the serialization graph *SG(H)*. Unfortunately, *SG(H)* itself does not satisfy this purpose. One shortcoming in serialization graphs is that they do not necessarily rule out histories in which failures cause two conflicting events to operate on disjoint sets of copies. It is, however, possible to augment *SG*(*H*) to form a *replicated data serialization graph* whose acyclicity is both necessary and sufficient for one copy serializability of the original history. We refer the interested reader to the more detailed exposition in Chapter eight of [26].

References

[1] N. Adly. Performance Evaluation of HARP: A Hierarchical Asynchronous Replication Protocol for Large Scale Systems. Technical Report TR-378, Computer Laboratory, University of Cambridge, August 1995.

[2] N. Adly and A. Kumar. HPP: A Hierarchical Propagation Protocol for Large Scale Replication in Wide Area Networks. Technical Report TR-331, Computer Laboratory, University of Cambridge, March 1994.

[3] N. Adly, M. Nagi, and J. Bacon. A Hierarchical Asynchronous Replication Protocol for Large Scale Systems. In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems,* pages 152–157, Princeton, New Jersey, October 1993.

[4] D. Agrawal and A. El Abbadi. Integrating security with fault-tolerance in distributed databases. *The Computer Journal,* 33(1):71–78, 1990.

[5] D. Agrawal and A. El Abbadi. Storage efficient replicated databases. *IEEE Trans. on Knowledge and Data Engineering,* 2(3):342–352, Sep. 1990.

[6] D. Agrawal and A. El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems,* 9(1):1–20, Feb 1991.

[7] G. Agrawal and P. Jalote. Coding-based replication schemes for distributed systems. *IEEE Trans. on Parallel and Distributed Systems,* 6(3):240–251, Mar. 1995.

[8] R. Alonso and D. Barbara. Negotiating data access in federated database systems. In *5th International Conference on Data Engineering,* pages 56–65, 1989.

[9] P.A. Alsberg and J.D. Day. A principle for resilient sharing of distributed resources. In *Proc. 2nd Int'l Conf. on Software Engineering,* pages 627–644, 1976.

[10] T. Anderson and P.A. Lee. *Fault Tolerance: Principles and Practice.* Prentice-Hall, 1981.

[11] H.R. Aschmann, N. Giger, and E. Hoepli. Alphorn: a remote procedure call environment for fault-tolerant, heterogeneous, distributed systems. *IEEE Micro,* pages 16–19, October 1991.

[12] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE Trans. on Software Engineering,* SE-11(12):1491–1501, Dec. 1985.

[13] B.R. Badrinath and T. Imielinski. Replication and mobility. In *Proceedings of the 2nd IEEE Workshop on Management of Replicated Data,* pages 9–12. IEEE, November 1992.

[14] H. Bal and A. Tanenbaum. Distributed programming with shared data. In *Proc. 8th IEEE Intl. Conference on Distributed Computing Systems, San Jose, California,* pages 82–91, 1988.

[15] R. Balter *et al.* Architecture and implementation of Guide: An object oriented distributed system. *USENIX Computing Systems,* 4(1):31–67, 1991.

[16] D. Barbara and H. Garcia-Molina. Mutual exclusion in partitioned distributed systems. *Distributed Computing,* 1:119–132, 1986.

[17] D. Barbara, H. Garcia-Molina, and A. Spauster. Policies for dynamic vote reassignment. In *Proc. IEEE Int'l Conf. on Distributed Computing Systems,* pages 37–44, 1986.

[18] D. Barbara, H. Garcia-Molina, and A. Spauster. Increasing availability under mutual exclusion constraints with dynamic voting reassignment. *ACM Transactions on Computer Systems,* 7(4):394–426, November 1989.

[19] J. Bartlett. A nonstop operating system. *Proc. of Hawaii Intn'l Conference on System Sciences,* January 1978.

[20] J. Bartlett. A NonStop kernel. *Proc. 8th Symp. on Operating System Principles,* pages 22–29, December 1981.

[21] J.F. Bartlett. The use of triple modular redundancy to improve computer reliability. *ACM Operating Sytems Review,* 15(5):22–29, December 1981.

[22] C. Beeri, H. Schek, and G. Weikum. Multi-level transaction management, theoretical art or practical need? In *Proceedings of the International Conference on Extending Database Technology,* pages 134–154, March 1988.

[23] P.A. Bernstein and N. Goodman. The failure and recovery problem for replicated databases. *Proc. 2nd ACM Symp. on Principles of Distributed Computing,* pages 114–122, August 1983.

[24] P.A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems,* 9(4):596–615, December 1984.

[25] P.A. Bernstein and N. Goodman. Serializability theory for replicated databases. *Journal of Computer and System Sciences,* 31(3):355–374, December 1986.

[26] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[27] P.A. Bernstein, D.W. Shipman, and J.B. Rothnie. Concurrency control in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems,* 5(1):18–51, March 1980.

[28] E. Bertino and L. Martino. Object–oriented database management systems: Concepts and issues. *IEEE Computer,* April 1991.

[29] A. Bestavros and S. Braoudakis. Timeliness via speculation for real-time databases. In *Proceedings of RTSS'94: The 14th IEEE Real-Time System Symposium,* San Juan, Puerto Rico, December 1994.

[30] B. Bhargava and S. Browne. Adaptable recovery using dynamic quorum assignments. In *16th International Conference on Very Large Databases,* pages 231–242, Brisbane, Australia, August 1990.

[31] B. Bhargava, S. Browne, and J. Srinivasan. Composite object replication in distributed database systems. In *Proc. Information Systems and Management of Data,* pages 115–134, July 1992.

[32] B. Bhargava, K. Friesen, A. Helal, S. Jagannathan, and J. Riedl. Design and implementation of the RAID-V2 distributed database system. Technical Report CSE-TR-962, Purdue University, March 1990.

[33] B. Bhargava, K. Friesen, A. Helal, and J. Riedl. Adaptability experiments in the raid distributed database system. *Proceedings of the 9th IEEE Symposium on Reliability in Distributed Systems,* pages 76–85, October 1990.

[34] B. Bhargava and A. Helal. Efficient availability mechanisms in distributed database systems. In *Proc. of the International Conference on Information and Knowledge Management,* Washington, D.C., November 1993.

[35] B. Bhargava, A. Helal, and K. Friesen. Analyzing availability of replicated database systems. *Int'l Journal of Computer Simulation,* 1(4):393–418, December 1991. A special issue on distributed file systems and database simulation.

[36] B. Bhargava, P. Noll, and D. Sabo. An experimental analysis of replicated copy control during site failure and recovery. In *Proc. 4th IEEE Data Engineering Conference,* pages 82–91, Los Angeles, February 1988.

[37] B. Bhargava and J. Riedl. A model for adaptable systems for transaction processing. In *Proc. 4th IEEE International Conference on Data Engineering,* pages 40–50, February 1988.

[38] K. Birman and T.A. Joseph. Exploiting Virtual Synchrony in distributed systems. In *Proc. 11th ACM Symp. on Operating System Principles, Austin, Texas,* pages 123–138, Nov. 1987.

[39] K.P. Birman and T.A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems,* 5(1):47–76, February 1987.

[40] A. Birrell, R. Levin, R.M. Needham, , and M.D. Shroeder. Grapevine: An exercise in distributed computing. *Communication of the ACM,* 25(4):260–274, April 1982.

[41] A. Birrell and B.J. Nelson. Implementing remote procedural calls. *ACM Transactions on Computer Systems,* 2(1):39–59, February 1984.

[42] B.T. Blaustein. Enforcing database assertions: Techniques and applications. Technical Report TR-21-81, Harvard University, 1981.

[43] J.J. Bloch. *A Practical Approach to Replication of Abstract Data Objects.* PhD thesis, Tech. Report CMU-CS-90-133, Carnegie Mellon University, May 1990.

[44] J.J. Bloch, D.S. Daniels, and A.Z. Spector. A weighted voting algorithm for replicated directories. *J. ACM,* pages 859–909, October 1987.

[45] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. *ACM Operating Systems Review,* 17(5):90–99, October 1983.

[46] A.J. Borr. Transaction monitoring in ENCOMPASS: Reliable distributed transaction processing. In *Proc. 7th VLDB Conference,* September 1981.

[47] K. Brahmadathan and K.V.S. Ramarao. Read only transactions in partitioned replicated databases. In *Proc. 5th IEEE Int'l Conf. on Data Engineering,* pages 522–529, February 1989.

[48] Y. Breitbart, A. Silberschatz, and G. Thompson. An update mechanism for multidatabase systems. *IEEE Data Engineering Bulletin,* 10(3):12–18, September 1987.

[49] S. Bruso. A failure detection and notification protocol for distributed computing systems. *Proc. of IEEE 5th Intn'l Conference on Distributed Computing Systems,* pages 116–123, May 1985.

[50] Special issue on next-generation database systems. *Communications of the ACM,* 34(10), October 1991.

[51] D.R. Cheriton. The V distributed system. *Communications of the ACM,* 31(3):314–333, March 1988.

[52] S.Y. Cheung, M.H. Ammar, and A. Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. In *Proc. IEEE Sixth Int'l Conf. on Data Engineering,* pages 438–445, May 1990.

[53] P. Chevalier. A replicated object server for a distributed object-oriented system. In *Proc. 11th IEEE Symp. on Reliable Distributed Systems,* pages 4–11, October 1992.

[54] P. Chevalier. A replicated object server for a distributed object-oriented system. In *Proc. 11th IEEE Symp. on Reliable Distributed Systems,* pages 4–11, October 1992.

[55] S.Y. Chueng, M.H. Ammar, and A. Ahamad. Multidimensional voting. *ACM Transactions on Computer Systems,* 9(4):399–431, November 1991.

[56] B. Coan, B. Oki, and E. Kolodner. Limitations on database availability when networks partition. *Proceeding of the 5th ACM Symposium on Principles of Distributed Computing,* pages 63–72, August 1986.

[57] E.C. Cooper. Circus: A remote procedural call facility. In *Proc. 10th Symp. on Reliability in Distributed Software and Database Systems,* pages 11–24, October 1984.

[58] E.C. Cooper. Replicated distributed programs. *ACM Operating Systems Review,* 19(5):63–78, Dec. 1985. Proc. 10th ACM Symp. on Operating System Principles, Orcas Island, Washington.

[59] P. Dasgupta *et al.* The Clouds distributed operating system: Functional description implementation details and related work. In *Proc. 8th International Conference on Distributed Computing Systems,* pages 2–9, 1988.

[60] P. Dasgupta *et al.* The design and implementation of the Clouds distributed operating system. *USENIX Computing Systems,* 3(1):11–46, 1990.

[61] D. Davcev and W. Burkhard. Consistency and recovery control for replicated data. In *Proc. 10th Symp. on Operating Systems Principles,* December 1985.

[62] S. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Transactions on Database Systems,* 9(3):456–481, 1984.

[63] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys,* 17(3); 341–370, September 1985.

[64] C.T. Davies. Recovery semantics for a DB/DC system. In *Proc. ACM National Conference,* 1973.

[65] J. Davis. Sybase system 10. *Open Information Systems,* March 1993.

[66] J. Davis. Oracle delivers 7.1. *Open Information Systems,* July 1994.

[67] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proc. ACM SIGMOD Annual Conference,* pages 204–214, 1990.

[68] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The bayou architecture: Support for data sharing among mobile users. In *Proceedings of the Workshop on Mobile Computing and Applications,* December 1994.

[69] W. Du and A. Elmagarmid. Quasi serializability: A correctness criterion for global concurrency control in interbase. In *Proc. VLDB Conference,* Netherlands, Amsterdam, August 1989.

[70] W. Du and A. Elmagarmid. Quasi serializability: a correctness criterion for global concurrency control in interbase. In *Proceedings of the International Conference on Very Large Data Bases,* pages 347–355, Amsterdam, The Netherlands, August 1989.

[71] W. Du, A. Elmagarmid, W. Kim, and O. Bukhres. Supporting consistent updates in replicated multidatabase systems. *International Journal on Very Large Data Bases,* 2(2), 1993.

[72] M. Dunham and A. Helal. Mobile computing and databases: Anything new? *The ACM SIGMOD Record,* 24(4), December 1995.

[73] D. Eager and K.C. Sevcik. Achieving robustness in distributed database systems. *ACM Transactions on Database Systems,* 8(3):354–381, 1983.

[74] A. El Abbadi and P.C. Aristides. Fast read only transactions in replicated databases. In *Proc. IEEE Int'l Conf. on Knowledge and Data Engineering,* pages 246–253, 1992.

[75] A. El Abbadi, D. Skeen, and F. Christian. An efficient fault tolerant protocol for replicated data management. In *Proc, 4th ACM Symp. on Principles of Database Systems,* pages 215–229, Portland, Oregon, March 1985.

[76] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems,* 14(2):264–290, June 1989.

[77] A. Elmagarmid and A. Helal. Supporting updates in heterogeneous distributed database systems. In *Proc. IEEE Int'l Conf. on Data Engineering,* pages 564–569, 1988.

[78] A. Elmagarmid, Y. Leu, and S.D. Ostermann. Effects of local autonomy on global concurrency control in heterogeneous distributed database systems. In *Proc. 2nd IEEE Int'l Conf. on Data and Knowledge Systems for Manufacturing and Engineering,* pages 113–120, 1989.

[79] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* Benjamin Cummings, 1994. Second Edition.

[80] J.L. Eppinger, L.B. Mummert, and A.Z. Spector. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan–Kaufmann, 1991.

[81] K.P. Eswaran, J.N. Gray, R. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM,* 19(11):624–633, November 1976.

[82] A.J. Ferrier and C. Stangert. Heterogeneity in distributed database management system SIRIUS-DELTA. In *Proc. 8th VLDB Conference,* 1983.

[83] M.J. Fischer and A. Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proc. 1st ACM Symp. on Principles of Database Systems,* pages 70–75, May 1982.

[84] H. Garcia and G. Wiederhold. Read-only transactions in a distributed database. *ACM Transactions on Database Systems,* 7(2):209–234, 1982.

[85] H. Garcia-Molina. Global consistency constraints considered harmful for heterogeneous database systems. In *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems,* pages 248–250, Kobe, Japan, April 1991.

[86] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *J. ACM,* 32(4):841–860, Oct 1985.

[87] H. Garcia-Molina and J. Kent. Performance evaluation of reliable distributed systems. In B. Bhargava, editor, *Concurrency Control and Reliability in Distributed Systems,* pages 454–488. Van Nostrand Reinhold, 1987.

[88] D.K. Gifford. Weighted voting for replicated data. *Proc. 7th Symp. on Operating System Principles,* pages 150–162, December 1979.

[89] V.D. Gligor and R. Popescu-Zeletin. Concurrency control issues in heterogeneous database management systems. *Distributed Data Sharing Systems,* 1985. H.J. Schreiber and W. Litwin, ed. North Holland.

[90] R. Golding and D.D. E Long. Accessing replicated data in an internetwork. *Int'l Journal of Computer Simulation,* 1(4):347–372, December 1991. A special issue on distributed file systems and database simulation.

[91] R.A. Golding. Weak consistency group communication for wide-area systems. In *Proc. IEEE Second Workshop on Management of Data,* November 1992.

[92] R.A. Golding and D.D.E. Long. Quorum-oriented multicast protocols for data replication. In *Proc. IEEE 6th Int'l Conf. on Data Engineering,* pages 490–497, 1992.

[93] K.J. Goldman. Data replication in nested transaction systems. Master's thesis, M.I.T., Lab. for Computer Science, May 1987.

[94] J. Gray. A census of Tandem system availability between 1985 and 1990. *IEEE Trans, on Reliability,* 39(4):409–418, Oct. 1990.

[95] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[96] J.M. Gray. Notes on database operating systems. *Lecture Notes in Computer Science,* pages 393–481, 1979. Springer-Verlag.

[97] J.M. Gray. An approach to decentralized computer systems. *IEEE Trans. on Software Engineering,* 12(6):684–692, June 1986.

[98] P.M. Gray, K.G. Kulkarni, and N.W. Paton. *Object-Oriented Databases.* Prentice-Hall International (UK) Ltd., 1992.

[99] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys,* 15(4):287–317, December 1983.

[100] A.A. Heddaya. *Managing Event-based Replication for Abstract Data Types in Distributed Systems.* PhD thesis, TR-20-88, Harvard University, 1988.

[101] A.A. Heddaya. View-based reconfiguration of replicated data. Technical Report BU-CS-90-001, Boston University, 1990.

[102] A.A. Heddaya, M. Hsu, and W.E. Weihl. Two phase gossip: Managing distributed event histories. *Information Sciences,* 49(1,2,3):35–57, Oct./Nov./Dec. 1989.

[103] C. Hedrick. Routing information protocol. *Network Working Group, RFC 1058,* June 1988.

[104] A. Helal. The Rainbow distributed database system. Technical Report TR-CSE-94-002, University of Texas at Arlington, February 1994. Submitted for publication.

[105] A. Helal. Modeling database system availability under network partitioning. *Information Sciences,* 83(l–2):23–35, mar 1995.

[106] A. Helal, S. Jagannathan, and B. Bhargava. SETH: A quorum-based replicated database system for experimentation with failures. *Proceedings of the 5th IEEE Int'l Conference on Data Engineering,* pages 677–684, February 1989.

[107] A. Helal, Y. Zhang, and B. Bhargava. Surveillance for controlled performance degradation during failures. In *Proc. 25th Hawaii Int'l Conf. on System Sciences,* pages 202–210, Kauai, Hawaii, January 1992.

[108] D. Hensgen and R. Finkel. Dynamic server squads in Yackos. Technical Report 138–89, Dept. of Computer Science, University of Kentucky, Lexingotn, 1989.

[109] M. Herlihy. A quorum consensus replication method for abstract data types. *ACM Transactions on Computer Systems,* 4(1):32–53, February 1986.

[110] M. Herlihy. Concurrency versus availability: Atomicity mechanisms for replicated data. *ACM Transactions on Computer Systems,* 5(3):249–274, August 1987.

[111] M. Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Transactions on Database Systems,* 12(2):170–194, June 1987.

[112] M.P. Herlihy and J.D. Tygar. How to make replicated data secure. In Carl Pomerance, editor, *Advances in Cryptology: CRYPTO '87, Carl Pomerance, editor. Lecture Notes in Computer Science, volume 293,* pages 379–391. Springer-Verlag, 1988.

[113] A. Hisgen, A. Birrell, T. Mann, M. Schroeder, and G. Swart. Availability and consistency trade-offs in the Echo distributed file system. In *Proc. 2nd Workshop on Workstation Operating Systems,* pages 49–54, 1989.

[114] J.G. Hughes. *Object-Oriented Databases.* Prentice-Hall International (UK) Ltd., 1991.

[115] A.R. Hurson, M.W. Bright, and S. Pakzad. *Multidatabase Systems: An Advanced Solution for Global Information Sharing.* IEEE Computer Society Press, 1993.

[116] S. Jajodia and D. Mutchler. Dynamic voting. In *Proc. ACM SIGMOD Annual Conference,* pages 227–238, May 1987.

[117] A. Jhingran and M. Stonebraker. Alternatives in complex object representation: A performance perspective. In *Proc. IEEE 6th Int'l Conf. on Data Engineering,* pages 94–102, February 1990.

[118] J. Jing, W. Du, A. Elmagarmid, and oh. Bukhres. Maintaining consistency of replicated data in multidatabase systems. In *IEEE Distributed Computing Systems,* Poznan, Poland, 1994.

[119] M. Jones, R. Rashid, and M. Thompson. Matchmaker: An interface specification language for distributed processing. In *Proc. 12th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages.* ACM, January 1986.

[120] T.A. Joseph and K.P. Birman. Low cost management of replicated data in fault-tolerant distributed systems. *ACM Transactions on Computer Systems,* 4(1):54–70, February 1986.

[121] H. Kesten. *Percolation Theory for Mathematics.* Birkhäser, 1982.

[122] W. Kim. Auditor: A framework for highly available DB-DC systems. *Proc. of IEEE Second Symposium on Reliability in Distributed Software and Database Systems,* pages 116–123, July 1982.

[123] J.J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems,* 10(1):3–25, February 1992.

[124] S. Kleiman. Vnodes: An architecture for multiple file system types in sun UNIX. In *Proc. USENIX Summer '86 Conf.*, pages 238–247, 1986.

[125] H. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proc. 16th VLDB Conference,* pages 95–106, 1990.

[126] O.C. Kowalski and H. Hartig. Protection in the BirliX operating system. In *Proc. 10th Int'l Conf. on Distributed Computing Systems,* pages 160–166, May 1990.

[127] A. Kumar. Hierarchical quorum consensus: A new class of algorithms for replicated data. Technical report, Graduate School of Management, Cornell University, August 1990.

[128] A. Kumar. Performance analysis of a hierarchical quorum consensus algorithm for replicated objects. *Proc. IEEE 10th Int'l Conf. on Distributed Computing Systems,* pages 378–385, July 1990.

[129] A. Kumar. A randomized voting algorithm. *Proc. IEEE 11th Int'l Conf. on Distributed Computing Systems,* pages 412–419, 1991.

[130] A. Kumar and S.Y. Chueng. A high availability $\sqrt{N}$ hierarchical grid algorithm for replicated data. *Info. Proc. Lett.,* 40:311–316, 1991.

[131] A. Kumar and A. Segev. Optimizing voting-type algorithms for replicated data. *Lecture Notes in Computer Science,* pages 428–442, March 1988. J.W. Schmidt, S. Ceri and M. Missekoff, Eds. Springer-Verlag.

[132] H.T. Kung and J.T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems,* 6(2):213–226, 1981.

[133] L. Lamport. The implementation of reliable multiprocessor systems. *Computer Networks,* 2(5):95–114, May 1978.

[134] L. Lamport. Time, clocks and ordering of events in a distributed system. *Communications of the ACM,* 21(7):558–564, July 1978.

[135] G. Lelann. Error recovery. In *Lecture Notes in Computer Science, Vol. 105: Distributed Systems - Architecture and Implementation, An Advanced Course,* chapter 15, pages 371–376. Springer-Verlag, 1981.

[136] B. Liskov. Distributed pogramming in Argus. *Communications of the ACM,* 31(3):300–313, Mar. 1988.

[137] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams Replication in the Harp file system. In *Proc. 13th ACM Symp. on Operating System Principles, Asilomar, California,* pages 226–238, October 1991.

[138] N. Liskov and R. Ladin. Highly available services in distributed systems. In *Proc. 5th ACM Symp. on Principles of Distributed Computing,* pages 29–39, August 1986.

[139] X. Liu. *Data Communication and Replication Strategies in Large-Scale Distributed Databases.* PhD thesis, Purdue University, Dec 1995.

[140] D.D.E. Long and J.L. Carroll. The reliability of regeneration-based replica control protocols. In *Proc. IEEE 9th Int'l Conf. on Distributed Computing,* pages 465–473, 1989.

[141] R.E. Lyons and W. Vanderkulk. The use of triple modular redundancy to improve computer reliability. *IBM Journal of Research and Development,* 6(2):200–209, April 1962.

[142] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems,* 3(2):145–159, May 1985.

[143] S.R. Mahaney and F.B. Schneider. Inexact agreement: Accuracy, precision and graceful degradation. *Proc. 4th ACM Symp. on Principles of Distributed Computing,* pages 237–249, Aug 1985.

[144] G. Martella, B. Ronchetti, and F. Shreiber. Availability evaluation in distributed database systems. *Performance Evaluation,* pages 201–211, 1981.

[145] B.E. Martin, C.A. Bergan, and B. Russ. Parpc: A system for parallel procedure calls. In *The 1987 International Conference on Parallel Processing,* The Pennsylvania State University, 1987.

[146] B. Meyer. *Object Oriented Software Construction.* Prentice Hall, 1988.

[147] A. Milo. Placement of replicated items in distributed databases. *Lecture Notes in Computer Science,* pages 414–427, March 1988. Springer-Verlag.

[148] T. Minoura and G. Wiederhold. Resilient extended true-copy token, scheme for a distributed database system. *IEEE Trans. on Software Engineering,* 9(5):173–189, May 1982.

[149] J.H. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S.H. Rosenthal, and F.D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM,* 29(3):184–201, March 1986.

[150] R. Mukkamala. Measuring the effect of data distribution and replication policies on performance evaluation of distributed database systems. In *Proc. IEEE Intl. Conf. on Data Engineering,* February 1989.

[151] P. Muth, W. Klas, and E.J. Neuhold. How to handle global transactions in heterogeneous database systems. In *IEEE 2nd Int'l Wkshp. on Research Issues on Data Engg.: Transaction and Query Processing,* pages 192–198, February 1992.

[152] E.J. Neuhold and B. Walter. An overview of the architecture of the distributed database POREL. *Distributed Databases,* 1982. H.J. Schneider, ed. North Holland.

[153] T. Ng and S. Shi. Replicated transactions. In *Proc. IEEE Int'l Conf. on Distributed Computing Systems,* pages 474–480, 1989.

[154] J.D. Noe and A. Andreassian. Effectiveness of replication in distributed computing systems. In *Proc. IEEE 7th Int'l Conf. on Distributed Computing,* pages 508–513, 1987.

[155] B.M. Oki and B.H. Liskov. Viewstamped replication: A general primary copy method to support highly available distributed systems. In *Proc. 7th ACM Symp. on Principles of Distributed Computing,* Toronto, Canada, Aug. 1988.

[156] D. C. Oppen and Y. K. Dalal. The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment. Technical Report OPD-T8103, Xerox Research Center, Palo Alto, California, 1981.

[157] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[158] C. Papadimitriou. The serializability of concurrent database updates. *J. ACM,* 26(4):631–653, October 1979.

[159] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, Rockville, Maryland, 1986.

[160] J. F Paris and D. Long. Efficient dynamic voting. In *Proc. IEEE Sixth Int'l Conf. on Distributed Computing Systems,* May 1986.

[161] J.F. Paris. Efficient voting protocols with witnesses. In *Proc. 3rd Int'l Conf. on Database Theory,* pages 606–612, May 1986. Springer-Verlag.

[162] J.F. Paris. Voting with witnesses: A consistency scheme for replicated files. In *Proc. IEEE 6th Int'l Conf. on Distributed Computing Systems,* pages 606–612, May 1986.

[163] J.F. Paris. Efficient management of replicated data. In *Proc. Int'l Conf. on Database Theory,* 1988.

[164] J.F. Paris. Voting with regenerable volatile witnesses. In *Proc. IEEE 7th Int'l Conf. on Data Engineering,* pages 112–119, April 1991.

[165] D.S. Parker and R.A. Ramos. A distributed file system architecture supporting high availability. In *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks,* pages 161–183, February 1982.

[166] G. Parrington, A. Shrivastava, S. Wheater, and M. Little. The design and implementation of arjuna. Technical Report 509, Department of Computing Science, University of Newcastle upon Tyne, 1995.

[167] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM,* 27(2):228–234, Apr 1979.

[168] A. Popek, R. Guy, T. Page Jr., and J. Heidemann. Replication in the Ficus distributed file system. In *Proc. Workshop on Management of Distributed Data,* pages 5–10, November 1990.

[169] A. Popek, R. Guy, T. Page Jr., and J. Heidemann. Replication in the Ficus distributed file system. In *Proc. Workshop on Management of Distributed Data,* pages 5–10, November 1990.

[170] A. Popek, B. Walker, D. Chow J., Edwards, C. Kline, G. Rudisin, and G. Thiel. Locus: A network transparent, high reliability distributed system. In *Proc. Workshop on Management of Distributed Data,* pages 5–10, November 1990.

[171] G. Popek, B. Walker, D. Butterfield, R. English, C. Kline, G. Thiel, and T. Page. Functionality and architecture of the LOCUS distributed operating system. In *Concurrency Control and Reliability in Distributed Systems,* pages 154–186. Van Nostrand Reinhold, 1987.

[172] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proc. ACM SIGMOD Annual Conference,* pages 377–386, June 1991.

[173] C. Pu and A. Leff. Autonomous transaction execution with Epsilon serializability. *Proc. IEEE Int'l Conf. on Data Engineering,* pages 2–11, 1992.

[174] C. Pu, J.D. Noe, and A. Proudfoot. Regeneration of replicated objects: A technique and its Eden implementation. In *Proc. IEEE Int'l Conf. on Data Engineering,* pages 175–187, February 1986.

[175] L.J. Raab. Bounds on the effects of replication on availability. In *Proc. Second IEEE Workshop on Management of Data,* November 1992.

[176] L.J. Raab. *Effects of Replication on Availability in Distributed Database Systems.* PhD thesis, Darthmouth College, Dept. of Computer Science, May 1992.

[177] M.O. Rabin. Efficient dispersal of information for security, load balancing, and fault-tolerance. *J. ACM,* 36(2):335–348, April 1989.

[178] S. Rangarajan, A. Sethia, and S.K. Tripathi. A fault-tolerant algorithm for replicated data management. In *Proc. IEEE Int'l Conf. on Knowledge and Data Engineering,* pages 230–237, 1992.

[179] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems,* 7(1):61–77, Feb 1989.

[180] M. Rusinkiewicz, A. Sheth, and G. Karabatis. Specifying interdatabase dependencies in a multidatabaseenvironment. *IEEE Computer,* 24(12):46–53, December 1991. Special Issue on Heterogeneous Distributed Databases

[181] R. Sandberg. The Sun Network File System. In *Proc. Summer USENIX Conference,* 1987.

[182] H. S. Sandhu and S. Zhou. Cluster-Based File Replication in Large-Scale Distributed Systems. In *Proceedings of the ACM SIGMETRICS & Performance Conference,* pages 91–102, Rhode Island, June 1992.

[183] S. Sarin, R. Floyd, and N. Phadnis. A flexible algorithm for replicated directory management. In *Proc. IEEE Int'l Conf. on Data Engineering,* pages 456–464, 1989.

[184] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. on Computers,* 39(4):447–459, April 1990.

[185] M. Satyanarayanan and E.H. Siegel. Parallel communication in a large distributed environment. *IEEE Transactions on Computers,* 39(4):328–348, Apr 1990.

[186] R. Schlichting and F. Schneider. Fail-stop processors: An approach to designing fault-tolerant distributed computing systems. *ACM Transactions on Computer Systems,* 1(3):222–238, 1983.

[187] A. Seigel, K. Birman, and K. Marzullo. Deceit: A flexible distributed file system. In *Proc. Summer 1990 USENIX Conference,* June 1990. USENIX Association.

[188] D. Shasha and C. Lazere. *Out of their Minds: The Lives and Discoveries of 15 Great Computer Scientists.* Copernicus by Springer-Verlag, New York, NY, 1995.

[189] E.J. Shekita and M.J. Carey. Performance enhancement through replication in an object-oriented DBMS. In *Proc. ACM SIGMOD,* 1989.

[190] A. Sheth. Semantic issues in multidatabase systems. In *SIGMOD Record: Special Issue,* December 1991.

[191] A. Sheth, Y. Leu, and A. Elmagarmid. Maintaining consistency of interdependent data in multidatabase systems. Technical Report CSD-TR-91-016, Computer Sciences Dept., Purdue University, March 1991.

[192] J. Sidell, P. Aoki, A. Sah, C. Stael, M. Stonebraker, and A. Yu. Data Replication in Mariposa. In *Proc. IEEE Intl. Conf. on Data Engineering,* February 1996.

[193] P.K. Sinha *et al.* The Galaxy distributed operating system. *IEEE Computer,* August 1991.

[194] D. Skeen. Nonblocking commit protocols. In *Proc. ACM SIGMOD Annual Conference,* pages 133–144, 1981.

[195] D. Skeen. Achieving high availability in partitioned database systems. In *Proc. IEEE Conf. on Data Engineering,* volume 4, pages 159–166, 1985.

[196] P.K. Sloope, J.F. Paris, and D.D.E. Long. A simulation study of replication control protocols using volatile witnesses. In *Proc. Annual Simulation Symposium,* pages 1–10, 1991.

[197] A.Z. Spector, J. Butcher, D.S. Daniels, D. Duchamp, J.L. Eppinger, C.E. Fineman, A.A. Heddaya, and P.M. Schwarz. Support for distributed transactions in the TABS prototype. *IEEE Trans. on Software Engineering,* SE-11(6):520–530, June 1985.

[198] M. Stonebraker and E. Neuhold. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. on Software Engineering,* 3(3):188–194, May 1979.

[199] I. Suzuki and T. Kazami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems,* 3(4):344–349, Nov 1985.

[200] A.S. Tanenbaum and R. van Renesse. Voting with ghosts. In *Proc. IEEE 8'th Int'l Conf. on Distributed Computing systems,* pages 456–461, June 1988.

[201] A.S. Tanenbaum, R. van Renesse, H. van Streveren, G.J. Sharp, S. Mullender, J. Jensen, and G van Rossum. Experiences with the Ameoba distributed operating system. *Communications of the ACM,* 33(12):46–63, December 1990.

[202] J. Tang. Managing replicated data in heterogenous database systems. In *Proc. 11th IEEE Symp. on Reliable Distributed Systems,* pages 12–19, October 1992.

[203] J. Tang. Voting class—an approach to achieving high availability for replicated data. In *Proc. IEEE Int'l Conf. on Data Engineering,* pages 146–156, October 1992.

[204] Y.C. Tay. The reliability of *(k,* n)-resilient distributed systems. In *Proc. 3rd IEEE Symp. on Reliable Distributed Systems, Silver Spring, Maryland,* pages 119–122, Oct. 1984.

[205] R.H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems,* 4(2):180–209, June 1979.

[206] W.F. Tichy, P. Lukowicz, L. Prechelt, and E.A. Heinz. Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software,* 28(1):9–18, January 1995.

[207] I.L. Traiger, J. Gray, C.A. Galtieri, and B.G. Lindsay. Transactions and consistency in distributed database systems. *ACM Transactions on Database Systems,* 7(3):323–342, September 1982.

[208] P. Triantafillou and D. Taylor. Efficiently maintaining availability in the presence of partitioning in distributed systems. In *Proc. 7th IEEE Int'l Conf. on Data Engineering,* pages 34–41, April 1991.

[209] P. Triantafillou and D. Taylor. The location-based paradigm for replication: Achieving efficiency and availability in distributed systems. *IEEE Trans. on Software Engineering,* 21(1):1–18, January 1995.

[210] K.S. Trivedi. *Probability & Statistics with Reliability, Queuing, and Computer Science Applications.* Prentice-Hall, 1982.

[211] J. Von Neumann. Probabilistic logics and the synthesis of reliable organisms for unreliable components. *Automata Studies,* pages 43–98, 1956. ed. D.E. Shannon and J. McCarthy, Princeton University Press.

[212] B. Walter. Network partitioning and surveillance protocols. *Proc. of IEEE 5th International Conference on Distributed Computing Systems,* pages 124–129, May 1985.

[213] W.E. Weihl. Distributed version management of read-only actions. *IEEE Trans. on Software Engineering,* 13(2):55–64, January 1987.

[214] W.E. Weihl. The impact of recovery on concurrency control. In *Proc. 8th ACM Symp. on Principles of Database Systems,* pages 259–269, Mar. 1989.

[215] P. Weiss. Yellow Pages protocol specification. Technical report, Sun Microsystems Inc., August 1985.

[216] G. Wiederhold and X. Qian. Consistency control of replicated data in federated databases. In *Proc. Workshop on the Mgmt. of Replicated Data,* November 1990.

[217] O. Wolfson, P. Sistla, S. Dao, K. Narayanan, and R. Raj. View maintenance in mobile computing. *The ACM SIGMOD Record,* 24(4):22–27, December 1995. Special Section on Data Management Issues in Mobile Computing. (M. Dunham and A. Helal, Editors).

[218] C. Wu and G.G. Belford. The triangular lattice protocol: A highly fault tolerant and highly efficient protocol for replicated data. In *Proc. 11th IEEE Symp. on Reliable Distributed Systems,* pages 66–73, October 1992.

[219] G.T. Wuu and A.J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proc. 3rd ACM Symp. on Principles of Distributed Computing,* pages 233–242, August 1984.

[220] Y. Zhang and B. Bhargava. WANCE: A Wide Area Network Communication Emulation System. In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems,* pages 40–45, Princeton, New Jersey, October 1993.

Index